

Global Optimization Toolbox

User's Guide



MATLAB[®]

R2023a



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Global Optimization Toolbox User's Guide

© COPYRIGHT 2004–2023 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

January 2004	Online only	New for Version 1.0 (Release 13SP1+)
June 2004	First printing	Revised for Version 1.0.1 (Release 14)
October 2004	Online only	Revised for Version 1.0.2 (Release 14SP1)
March 2005	Online only	Revised for Version 1.0.3 (Release 14SP2)
September 2005	Second printing	Revised for Version 2.0 (Release 14SP3)
March 2006	Online only	Revised for Version 2.0.1 (Release 2006a)
September 2006	Online only	Revised for Version 2.0.2 (Release 2006b)
March 2007	Online only	Revised for Version 2.1 (Release 2007a)
September 2007	Third printing	Revised for Version 2.2 (Release 2007b)
March 2008	Online only	Revised for Version 2.3 (Release 2008a)
October 2008	Online only	Revised for Version 2.4 (Release 2008b)
March 2009	Online only	Revised for Version 2.4.1 (Release 2009a)
September 2009	Online only	Revised for Version 2.4.2 (Release 2009b)
March 2010	Online only	Revised for Version 3.0 (Release 2010a)
September 2010	Online only	Revised for Version 3.1 (Release 2010b)
April 2011	Online only	Revised for Version 3.1.1 (Release 2011a)
September 2011	Online only	Revised for Version 3.2 (Release 2011b)
March 2012	Online only	Revised for Version 3.2.1 (Release 2012a)
September 2012	Online only	Revised for Version 3.2.2 (Release 2012b)
March 2013	Online only	Revised for Version 3.2.3 (Release 2013a)
September 2013	Online only	Revised for Version 3.2.4 (Release 2013b)
March 2014	Online only	Revised for Version 3.2.5 (Release 2014a)
October 2014	Online only	Revised for Version 3.3 (Release 2014b)
March 2015	Online only	Revised for Version 3.3.1 (Release 2015a)
September 2015	Online only	Revised for Version 3.3.2 (Release 2015b)
March 2016	Online only	Revised for Version 3.4 (Release 2016a)
September 2016	Online only	Revised for Version 3.4.1 (Release 2016b)
March 2017	Online only	Revised for Version 3.4.2 (Release 2017a)
September 2017	Online only	Revised for Version 3.4.3 (Release 2017b)
March 2018	Online only	Revised for Version 3.4.4 (Release 2018a)
September 2018	Online only	Revised for Version 4.0 (Release 2018b)
March 2019	Online only	Revised for Version 4.1 (Release 2019a)
September 2019	Online only	Revised for Version 4.2 (Release 2019b)
March 2020	Online only	Revised for Version 4.3 (Release 2020a)
September 2020	Online only	Revised for Version 4.4 (Release 2020b)
March 2021	Online only	Revised for Version 4.5 (Release 2021a)
September 2021	Online only	Revised for Version 4.6 (Release 2021b)
March 2022	Online only	Revised for Version 4.7 (Release 2022a)
September 2022	Online only	Revised for Version 4.8 (Release 2022b)
March 2023	Online only	Revised for Version 4.8.1 (Release 2023a)

Getting Started

1	Introducing Global Optimization Toolbox Functions	
	Global Optimization Toolbox Product Description	1-2
	Key Features	1-2
	Compare Several Global Solvers, Problem-Based	1-3
	Comparison of Six Solvers	1-10
	Solver Behavior with a Nonsmooth Problem	1-17
	What Is Global Optimization?	1-24
	Local vs. Global Optima	1-24
	Basins of Attraction	1-25
	Optimization Workflow	1-28
	Table for Choosing a Solver	1-29
	Global Optimization Toolbox Solver Characteristics	1-30
	Solver Choices	1-30
	Explanation of “Desired Solution”	1-30
	Choosing Between Solvers for Smooth Problems	1-32
	Choosing Between Solvers for Nonsmooth Problems	1-32
	Solver Characteristics	1-33
	Why Are Some Solvers Objects?	1-35

2	Write Files for Optimization Functions	
	Compute Objective Functions	2-2
	Objective (Fitness) Functions	2-2
	Write a Function File	2-2
	Write a Vectorized Function	2-3
	Gradients and Hessians	2-4
	Maximizing vs. Minimizing	2-5

Write Constraints	2-6
Consult Optimization Toolbox Documentation	2-6
Set Bounds	2-6
Ensure ga Options Maintain Feasibility	2-6
Gradients and Hessians	2-7
Vectorized Constraints	2-7
Set and Change Options	2-9
View Options	2-10

Problem-Based Global Optimization

3

Decide Between Problem-Based and Solver-Based Approach	3-2
Global Optimization Toolbox Default Solvers and Problem Types	3-4
Initial Points for Global Optimization Toolbox Solvers	3-6
Integer Constraints in Nonlinear Problem-Based Optimization	3-8
Set Problem-Based Optimization Options for Global Optimization Toolbox Solvers	3-9

Using GlobalSearch and MultiStart

4

Problems That GlobalSearch and MultiStart Can Solve	4-2
Workflow for GlobalSearch and MultiStart	4-3
Create Problem Structure	4-4
About Problem Structures	4-4
Use the createOptimProblem Function	4-4
Example: Create a Problem Structure with createOptimProblem	4-5
Create Solver Object	4-7
What Is a Solver Object?	4-7
Properties (Global Options) of Solver Objects	4-7
Creating a Nondefault GlobalSearch Object	4-8
Creating a Nondefault MultiStart Object	4-9
Set Start Points for MultiStart	4-10
Four Ways to Set Start Points	4-10
Positive Integer for Start Points	4-10
RandomStartPointSet Object for Start Points	4-10
CustomStartPointSet Object for Start Points	4-11
Cell Array of Objects for Start Points	4-12

Run the Solver	4-13
Optimize by Calling run	4-13
Example of Run with GlobalSearch	4-13
Example of Run with MultiStart	4-14
Single Solution	4-16
Multiple Solutions	4-17
About Multiple Solutions	4-17
Change the Definition of Distinct Solutions	4-19
Iterative Display	4-21
Types of Iterative Display	4-21
Examine Types of Iterative Display	4-21
Global Output Structures	4-23
Visualize the Basins of Attraction	4-24
Output Functions for GlobalSearch and MultiStart	4-27
What Are Output Functions?	4-27
GlobalSearch Output Function	4-27
No Parallel Output Functions	4-28
Plot Functions for GlobalSearch and MultiStart	4-30
What Are Plot Functions?	4-30
MultiStart Plot Function	4-30
No Parallel Plot Functions	4-33
How GlobalSearch and MultiStart Work	4-34
Multiple Runs of a Local Solver	4-34
Differences Between the Solver Objects	4-34
GlobalSearch Algorithm	4-35
MultiStart Algorithm	4-38
Bibliography	4-40
Can You Certify That a Solution Is Global?	4-41
No Guarantees	4-41
Check if a Solution Is a Local Solution with patternsearch	4-41
Identify a Bounded Region That Contains a Global Solution	4-42
Use MultiStart with More Start Points	4-42
Refine Start Points	4-44
About Refining Start Points	4-44
Methods of Generating Start Points	4-44
Example: Searching for a Better Solution	4-46
Change Options	4-51
How to Determine Which Options to Change	4-51
Changing Local Solver Options	4-51
Changing Global Options	4-52
Reproduce Results	4-54
Identical Answers with Pseudorandom Numbers	4-54
Steps to Take in Reproducing Results	4-54

Example: Reproducing a GlobalSearch or MultiStart Result	4-54
Parallel Processing and Random Number Streams	4-55
Find Global or Multiple Local Minima	4-57
Maximizing Monochromatic Polarized Light Interference Patterns Using GlobalSearch and MultiStart	4-64
Optimize Using Only Feasible Start Points	4-76
MultiStart Using Isqcurvefit or Isqnonlin	4-79
Parallel MultiStart	4-83
Steps for Parallel MultiStart	4-83
Speedup with Parallel Computing	4-84
Isolated Global Minimum	4-86
Difficult-To-Locate Global Minimum	4-86
Default Settings Cannot Find the Global Minimum — Add Bounds	4-87
GlobalSearch with Bounds and More Start Points	4-88
MultiStart with Bounds and Many Start Points	4-88
MultiStart Without Bounds, Widely Dispersed Start Points	4-89
MultiStart with a Regular Grid of Start Points	4-89
MultiStart with Regular Grid and Promising Start Points	4-90

Problem-Based Multiple Start

5

Minimize Nonlinear Function Using Multiple-Start Solver, Problem-Based	5-2
Specify Start Points for MultiStart, Problem-Based	5-3
MultiStart with Isqnonlin, Problem-Based	5-6
Find Multiple Local Solutions Using MultiStart or GlobalSearch, Problem-Based	5-9

Using Direct Search

6

What Is Direct Search?	6-2
Optimize Using the GPS Algorithm	6-3
Coding and Minimizing an Objective Function Using Pattern Search ..	6-10
Constrained Minimization Using Pattern Search, Solver-Based	6-14

Effects of Pattern Search Options	6-18
Pattern Search Terminology	6-24
Patterns	6-24
Meshes	6-24
Polling	6-25
Expanding and Contracting	6-25
How Pattern Search Polling Works	6-27
Context	6-27
Successful Polls	6-27
An Unsuccessful Poll	6-29
Successful and Unsuccessful Polls in MADS	6-30
Displaying the Results at Each Iteration	6-31
More Iterations	6-31
Poll Method	6-32
Complete Poll	6-33
Stopping Conditions for the Pattern Search	6-33
Robustness of Pattern Search	6-34
Nonuniform Pattern Search (NUPS) Algorithm	6-35
Searching and Polling	6-37
Definition of Search	6-37
How to Use a Search Method	6-38
Built-In Search Types	6-39
When to Use Search	6-40
Setting Solver Tolerances	6-41
Search and Poll	6-42
Nonlinear Constraint Solver Algorithm for Pattern Search	6-46
Custom Plot Function	6-48
About Custom Plot Functions	6-48
Creating the Custom Plot Function	6-48
Set Up the Problem	6-49
Run the Optimization with Custom Plot Function	6-49
How the Plot Function Works	6-50
Pattern Search Climbs Mount Washington	6-52
Set Options	6-57
Polling Types	6-59
Using a Complete Poll in a Generalized Pattern Search	6-59
Compare the Efficiency of Poll Options	6-61
Set Mesh Options	6-66
Mesh Expansion and Contraction	6-66
Mesh Accelerator	6-69

Constrained Minimization Using patternsearch and Optimize Live Editor Task	6-71
Problem Description	6-71
Solve Using patternsearch in Optimize Live Editor Task	6-72
Solve Using patternsearch at the Command Line	6-78
Use Cache	6-80
Vectorize the Objective and Constraint Functions	6-83
Vectorize for Speed	6-83
Vectorized Objective Function	6-83
Vectorized Constraint Functions	6-85
Example of Vectorized Objective and Constraints	6-85
Optimize ODEs in Parallel	6-87
Optimization of Stochastic Objective Function	6-95
Explore patternsearch Algorithms	6-103
Explore patternsearch Algorithms in Optimize Live Editor Task	6-107

Problem-Based Direct Search

7

Optimize Nonsmooth Function Using patternsearch, Problem-Based . . .	7-2
Constrained Minimization Using Pattern Search, Problem-Based	7-4
Effects of Pattern Search Options, Problem-Based	7-10
Search and Poll, Problem-Based	7-16

Using the Genetic Algorithm

8

What Is the Genetic Algorithm?	8-2
Minimize Rastrigin's Function	8-4
Genetic Algorithm Terminology	8-13
Fitness Functions	8-13
Individuals	8-13
Populations and Generations	8-13
Diversity	8-13
Fitness Values and Best Fitness Values	8-14
Parents and Children	8-14

How the Genetic Algorithm Works	8-15
Outline of the Algorithm	8-15
Initial Population	8-15
Creating the Next Generation	8-16
Plots of Later Generations	8-18
Stopping Conditions for the Algorithm	8-18
Selection	8-19
Reproduction Options	8-19
Mutation and Crossover	8-20
Integer and Linear Constraints	8-20
 Coding and Minimizing a Fitness Function Using the Genetic Algorithm	 8-22
 Constrained Minimization Using the Genetic Algorithm	 8-27
 Effects of Genetic Algorithm Options	 8-32
 Mixed Integer ga Optimization	 8-40
Solving Mixed Integer Optimization Problems	8-40
Characteristics of the Integer ga Solver	8-41
Effective Integer ga	8-45
Integer ga Algorithm	8-45
 Solve a Mixed-Integer Engineering Design Problem Using the Genetic Algorithm	 8-47
 Nonlinear Constraint Solver Algorithms for Genetic Algorithm	 8-56
Augmented Lagrangian Genetic Algorithm	8-56
Penalty Algorithm	8-57
 Create Custom Plot Function	 8-59
About Custom Plot Functions	8-59
Creating the Custom Plot Function	8-59
Using the Custom Plot Function	8-59
How the Plot Function Works	8-60
 Resume ga	 8-62
 Options and Outputs	 8-64
Running ga with the Default Options	8-64
Setting Options at the Command Line	8-64
Additional Output Arguments	8-65
 Reproduce Results	 8-67
 Run ga from a File	 8-69
 Population Diversity	 8-72
Importance of Population Diversity	8-72
Set Initial Range	8-72
Custom Plot Function and Linear Constraints in ga	8-75
Setting the Population Size	8-79

Fitness Scaling	8-80
Scaling the Fitness Scores	8-80
Comparing Rank and Top Scaling	8-81
Vary Mutation and Crossover	8-83
Global vs. Local Optimization Using ga	8-91
Hybrid Scheme in the Genetic Algorithm	8-95
Set Maximum Number of Generations and Stall Generations	8-101
Vectorize the Fitness Function	8-103
Vectorize for Speed	8-103
Vectorized Constraints	8-104
Custom Output Function for Genetic Algorithm	8-105
Custom Data Type Optimization Using the Genetic Algorithm	8-109
When to Use a Hybrid Function	8-116

Problem-Based Genetic Algorithm

9

Minimize Rastrigins' Function Using ga, Problem-Based	9-2
Solve a Mixed-Integer Engineering Design Problem Using the Genetic Algorithm, Problem-Based	9-5
Set Options in Problem-Based Approach Using varindex	9-17
Constrained Minimization Using ga, Problem-Based	9-19

Particle Swarm Optimization

10

What Is Particle Swarm Optimization?	10-2
Optimize Function Using particleswarm, Problem-Based	10-3
Optimize Using Particle Swarm	10-5
Particle Swarm Output Function	10-8
Particle Swarm Optimization Algorithm	10-11
Algorithm Outline	10-11
Initialization	10-11

Iteration Steps	10-12
Stopping Criteria	10-13
Tune Particle Swarm Optimization Process	10-14

Surrogate Optimization

11

What Is Surrogate Optimization?	11-2
Surrogate Optimization Algorithm	11-3
Serial surrogateopt Algorithm	11-3
Mixed-Integer surrogateopt Algorithm	11-8
Linear Constraint Handling	11-9
surrogateopt Algorithm with Nonlinear Constraints	11-9
Parallel surrogateopt Algorithm	11-10
Parallel Mixed-Integer surrogateopt Algorithm	11-10
Surrogate Optimization of Multidimensional Function	11-12
Modify surrogateopt Options	11-19
Interpret surrogateoptplot	11-25
Compare Surrogate Optimization with Other Solvers	11-31
Surrogate Optimization with Nonlinear Constraint	11-41
Surrogate Optimization of Six-Element Yagi-Uda Antenna	11-47
Work with Checkpoint Files	11-56
Checkpoint for Restarting	11-56
Change Options to Extend or Monitor Optimization	11-58
Code for Robust Surrogate Optimization	11-60
Mixed-Integer Surrogate Optimization	11-62
Fix Variables in surrogateopt	11-64
Optimal Component Choice Using surrogateopt	11-67
Convert Nonlinear Constraints Between surrogateopt Form and Other Solver Forms	11-74
Why Convert Constraint Forms?	11-74
Convert from surrogateopt Structure Form to Other Solvers	11-74
Convert from Other Solvers to surrogateopt Structure Form	11-76
Solve Feasibility Problem	11-78
Solve Nonlinear Problem with Integer and Nonlinear Constraints	11-82

Improve surrogateopt Solution or Process	11-90
surrogateopt Stalls	11-90
No Feasible Point Found	11-90
Solution Might Not Be Optimal	11-91
Vectorized Surrogate Optimization for Custom Parallel Simulation ...	11-92

Problem-Based Surrogate Optimization

12

Optimize Multidimensional Function Using surrogateopt, Problem-Based	12-2
Solve Feasibility Problem Using surrogateopt, Problem-Based	12-6
Feasibility Using Problem-Based Optimize Live Editor Task	12-11
Mixed-Integer Surrogate Optimization, Problem-Based	12-20
Specify Starting Points and Values for surrogateopt, Problem-Based .	12-24

Using Simulated Annealing

13

What Is Simulated Annealing?	13-2
Optimize Function Using simulannealbnd, Problem-Based	13-3
Minimize Function with Many Local Minima	13-5
Simulated Annealing Terminology	13-12
Objective Function	13-12
Temperature	13-12
Annealing Parameter	13-12
Reannealing	13-12
How Simulated Annealing Works	13-14
Outline of the Algorithm	13-14
Stopping Conditions for the Algorithm	13-15
Bibliography	13-16
Reproduce Your Results	13-17
Minimization Using Simulated Annealing Algorithm	13-19
Simulated Annealing Options	13-22

Multiprocessor Scheduling Using Simulated Annealing with a Custom Data Type	13-28
--	--------------

14

Multiobjective Optimization

What Is Multiobjective Optimization?	14-2
gamultiobj Algorithm	14-5
Introduction	14-5
Multiobjective Terminology	14-5
Initialization	14-7
Iterations	14-7
Stopping Conditions	14-7
Integer and Linear Constraints	14-8
Bibliography	14-8
paretosearch Algorithm	14-10
paretosearch Algorithm Overview	14-10
Definitions for paretosearch Algorithm	14-10
Sketch of paretosearch Algorithm	14-13
Initialize Search	14-14
Create Archive and Incumbents	14-14
Poll to Find Better Points	14-14
Update archive and iterates Structures	14-15
Stopping Conditions	14-15
Returned Values	14-16
Modifications for Parallel Computation and Vectorized Function Evaluation	14-16
Run paretosearch Quickly	14-16
gamultiobj Options and Syntax: Differences from ga	14-18
Pareto Front for Two Objectives	14-19
Multiobjective Optimization with Two Objectives	14-19
Find Pareto Set Using Optimize Live Editor Task	14-19
Find Pareto Set at the Command Line	14-24
Alternate Views	14-25
Compare paretosearch and gamultiobj	14-27
Plot 3-D Pareto Front	14-38
Performing a Multiobjective Optimization Using the Genetic Algorithm	14-48
Effects of Multiobjective Genetic Algorithm Options	14-53
Design Optimization of a Welded Beam	14-62

Problem-Based Multiobjective Optimization

15

Steps for Problem-Based Multiobjective Optimization	15-2
Specify Multiple Objective Functions	15-2
Specify Multiple Objective Senses (Maximize or Minimize)	15-2
Data Format of Multiobjective Solutions	15-3
Supply Initial Points for Multiobjective Problem	15-3
Hybrid Function	15-3
View Pareto Set	15-3
Pareto Front for Multiobjective Optimization, Problem-Based	15-5
Plan Nuclear Fuel Disposal Using Multiobjective Optimization	15-10

Parallel Processing

16

How Solvers Compute in Parallel	16-2
Parallel Processing Types in Global Optimization Toolbox	16-2
How Toolbox Functions Distribute Processes	16-3
How to Use Parallel Processing in Global Optimization Toolbox	16-11
Multicore Processors	16-11
Processor Network	16-12
Parallel Search Functions or Hybrid Functions	16-14
Testing Parallel Optimization	16-15
Minimizing an Expensive Optimization Problem Using Parallel Computing Toolbox	16-16

Options Reference

17

GlobalSearch and MultiStart Properties (Options)	17-2
How to Set Properties	17-2
Properties of Both Objects	17-2
GlobalSearch Properties	17-5
MultiStart Properties	17-6
Pattern Search Options	17-7
Options for Pattern Search	17-7
Algorithm Options	17-7
Plot Options	17-8
Poll Options	17-10
Multiobjective Options	17-11
Search Options	17-12
Mesh Options	17-15

Constraint Parameters	17-16
Cache Options	17-16
Stopping Criteria	17-17
Output Function Options	17-17
Display to Command Window Options	17-19
Vectorized and Parallel Options	17-19
Options Table for Pattern Search Algorithms	17-21
Genetic Algorithm Options	17-23
Options for Genetic Algorithm	17-23
Plot Options	17-23
Population Options	17-26
Fitness Scaling Options	17-29
Selection Options	17-30
Reproduction Options	17-31
Mutation Options	17-31
Crossover Options	17-34
Migration Options	17-37
Constraint Parameters	17-37
Multiobjective Options	17-38
Hybrid Function Options	17-39
Stopping Criteria Options	17-40
Output Function Options	17-41
Display to Command Window Options	17-42
Vectorize and Parallel Options (User Function Evaluation)	17-43
Particle Swarm Options	17-45
Specifying Options for particleswarm	17-45
Swarm Creation	17-45
Display Settings	17-46
Algorithm Settings	17-46
Hybrid Function	17-47
Output Function and Plot Function	17-48
Parallel or Vectorized Function Evaluation	17-49
Stopping Criteria	17-50
Surrogate Optimization Options	17-51
Algorithm Control	17-51
Stopping Criteria	17-52
Command-Line Display	17-52
Output Function	17-53
Plot Function	17-55
Parallel Computing	17-56
Vectorized Computing	17-56
Checkpoint File	17-56
Simulated Annealing Options	17-58
Set Simulated Annealing Options at the Command Line	17-58
Plot Options	17-58
Temperature Options	17-59
Algorithm Settings	17-60
Hybrid Function Options	17-61
Stopping Criteria Options	17-62
Output Function Options	17-62
Display Options	17-64

Options Changes in R2016a	17-65
Use optimoptions to Set Options	17-65
Options that optimoptions Hides	17-65
Table of Option Names in Legacy Order	17-67
Table of Option Names in Current Order	17-70

Functions

18 |

Getting Started

Introducing Global Optimization Toolbox Functions

- “Global Optimization Toolbox Product Description” on page 1-2
- “Compare Several Global Solvers, Problem-Based” on page 1-3
- “Comparison of Six Solvers” on page 1-10
- “Solver Behavior with a Nonsmooth Problem” on page 1-17
- “What Is Global Optimization?” on page 1-24
- “Optimization Workflow” on page 1-28
- “Table for Choosing a Solver” on page 1-29
- “Global Optimization Toolbox Solver Characteristics” on page 1-30

Global Optimization Toolbox Product Description

Solve multiple maxima, multiple minima, and nonsmooth optimization problems

Global Optimization Toolbox provides functions that search for global solutions to problems that contain multiple maxima or minima. Toolbox solvers include surrogate, pattern search, genetic algorithm, particle swarm, simulated annealing, multistart, and global search. You can use these solvers for optimization problems where the objective or constraint function is continuous, discontinuous, stochastic, does not possess derivatives, or includes simulations or black-box functions. For problems with multiple objectives, you can identify a Pareto front using genetic algorithm or pattern search solvers.

You can improve solver effectiveness by adjusting options and, for applicable solvers, customizing creation, update, and search functions. You can use custom data types with the genetic algorithm and simulated annealing solvers to represent problems not easily expressed with standard data types. The hybrid function option lets you improve a solution by applying a second solver after the first.

Key Features

- Surrogate solver for problems with lengthy objective function execution times and bound constraints
- Pattern search solvers for single and multiple objective problems with linear, nonlinear, and bound constraints
- Genetic algorithm for problems with linear, nonlinear, bound, and integer constraints
- Multiobjective genetic algorithm for problems with linear, nonlinear, and bound constraints
- Particle swarm solver for bound constraints
- Simulated annealing solver for bound constraints
- Multistart and global search solvers for smooth problems with linear, nonlinear, and bound constraints

Compare Several Global Solvers, Problem-Based

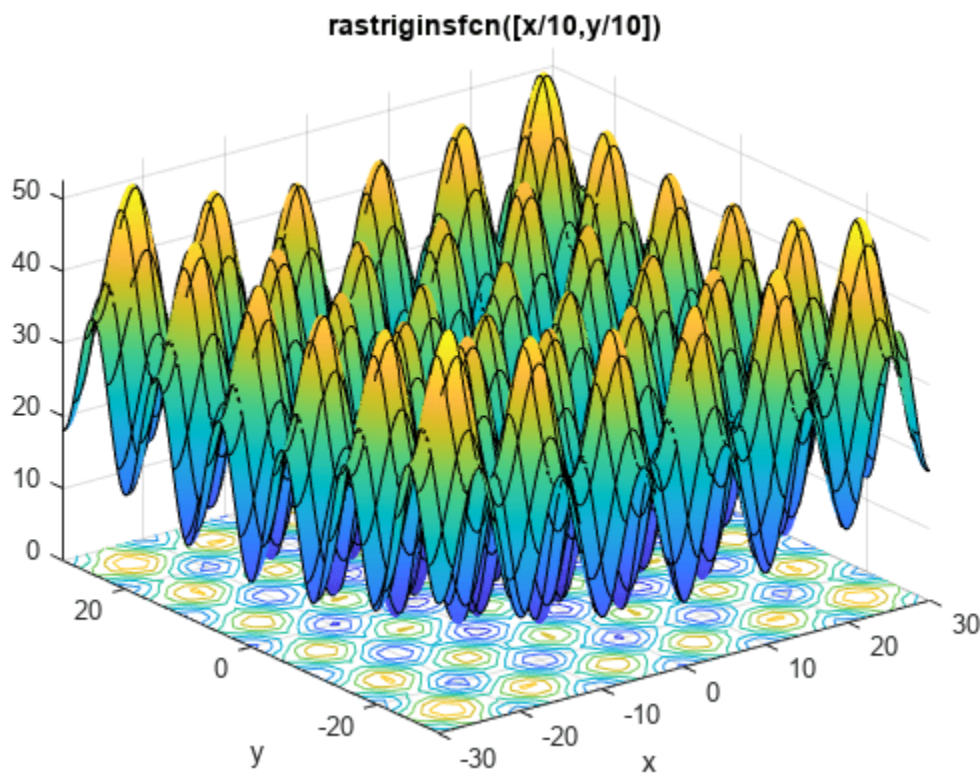
This example shows how to minimize Rastrigin's function with several solvers. Each solver has its own characteristics. The characteristics lead to different solutions and run times. The results, summarized in Compare Solvers and Solutions on page 1-8, can help you choose an appropriate solver for your own problems.

Rastrigin's function has many local minima, with a global minimum at (0,0):

```
ras = @(x, y) 20 + x.^2 + y.^2 - 10*(cos(2*pi*x) + cos(2*pi*y));
```

Plot the function scaled by 10 in each direction.

```
rf3 = @(x, y) ras(x/10, y/10);
fsurf(rf3, [-30 30], "ShowContours", "on")
title("rastriginsfcn([x/10,y/10])")
xlabel("x")
ylabel("y")
```



Usually you don't know the location of the global minimum of your objective function. To show how the solvers look for a global solution, this example starts all the solvers around the point [20,30], which is far from the global minimum.

fminunc Solver

To solve the optimization problem using the default fminunc Optimization Toolbox™ solver, enter:

```
x = optimvar("x");
y = optimvar("y");
prob = optimproblem("Objective", rf3(x,y));
x0.x = 20;
x0.y = 30;
[solf,fvalf,eflagf,outputf] = solve(prob,x0)
```

Solving problem using fminunc.

Local minimum found.

Optimization completed because the size of the gradient is less than the value of the optimality tolerance.

```
solf = struct with fields:
    x: 19.8991
    y: 29.8486
```

```
fvalf = 12.9344
```

```
eflagf =
    OptimalSolution
```

```
outputf = struct with fields:
    iterations: 3
    funcCount: 5
    stepsize: 1.7773e-06
    lssteplength: 1
    firstorderopt: 2.0461e-09
    algorithm: 'quasi-newton'
    message: 'Local minimum found....'
    objectivederivative: "reverse-AD"
    solver: 'fminunc'
```

`fminunc` solves the problem in very few function evaluations (only five, as shown in the `outputf` structure), and reaches a local minimum near the start point. The exit flag indicates that the solution is a local minimum.

patternsearch Solver

To solve the optimization problem using the `patternsearch` Global Optimization Toolbox solver, enter:

```
x0.x = 20;
x0.y = 30;
[solp,fvalp,eflagp,outputp] = solve(prob,x0,"Solver","patternsearch")
```

Solving problem using `patternsearch`.

Optimization terminated: mesh size less than options.MeshTolerance.

```
solp = struct with fields:
    x: 19.8991
    y: -9.9496
```

```
fvalp = 4.9748
```

```

eflagp =
    SolverConvergedSuccessfully

outputp = struct with fields:
    function: @(x)fun(x,extraParams)
    problemtype: 'unconstrained'
    pollmethod: 'gpspositivebasis2n'
    maxconstraint: []
    searchmethod: []
    iterations: 48
    funccount: 174
    meshsize: 9.5367e-07
    rngstate: [1x1 struct]
    message: 'Optimization terminated: mesh size less than options.MeshTolerance.'
    solver: 'patternsearch'

```

Like `fminunc`, `patternsearch` reaches a local optimum, as shown in the exit flag `exitflagp`. The solution is better than the `fminunc` solution, because it has a lower objective function value. However, `patternsearch` takes many more function evaluations, as shown in the output structure.

ga Solver

To solve the optimization problem using the `ga` Global Optimization Toolbox solver, enter:

```

rng default % For reproducibility
x0.x = 10*randn(20) + 20;
x0.y = 10*randn(20) + 30; % Random start population near [20,30];
[solg,fvalg,eflagg,outputg] = solve(prob,"Solver","ga")

```

```

Solving problem using ga.
Optimization terminated: maximum number of generations exceeded.

```

```

solg = struct with fields:
    x: 0.0064
    y: 7.7057e-04

```

```
fvalg = 8.1608e-05
```

```

eflagg =
    SolverLimitExceeded

```

```

outputg = struct with fields:
    problemtype: 'unconstrained'
    rngstate: [1x1 struct]
    generations: 200
    funccount: 9453
    message: 'Optimization terminated: maximum number of generations exceeded.'
    maxconstraint: []
    hybridflag: []
    solver: 'ga'

```

`ga` takes many more function evaluations than the previous solvers, and arrives at a solution near the global minimum. The solver is stochastic, and can reach a suboptimal solution.

particleswarm Solver

To solve the optimization problem using the `particleswarm` Global Optimization Toolbox solver, enter:

```
rng default % For reproducibility
[solpso,fvalpso,eflagpso,outputpso] = solve(prob,"Solver","particleswarm")

Solving problem using particleswarm.
Optimization ended: relative change in the objective value
over the last OPTIONS.MaxStallIterations iterations is less than OPTIONS.FunctionTolerance.

solpso = struct with fields:
  x: 7.1467e-07
  y: 1.4113e-06

fvalpso = 4.9631e-12

eflagpso =
  SolverConvergedSuccessfully

outputpso = struct with fields:
  rngstate: [1x1 struct]
  iterations: 120
  funccount: 2420
  message: 'Optimization ended: relative change in the objective value ...'
  hybridflag: []
  solver: 'particleswarm'
```

The solver takes fewer function evaluations than `ga`, and arrives at an even more accurate solution. Again, the solver is stochastic and can fail to reach a global solution.

simulannealbnd Solver

To solve the optimization problem using the `simulannealbnd` Global Optimization Toolbox solver, enter:

```
rng default % For reproducibility
x0.x = 20;
x0.y = 30;
[solsim,fvalsim,eflagsim,outputsim] = solve(prob,x0,"Solver","simulannealbnd")

Solving problem using simulannealbnd.
Optimization terminated: change in best function value less than options.FunctionTolerance.

solsim = struct with fields:
  x: 0.0025
  y: 0.0018

fvalsim = 1.8311e-05

eflagsim =
  SolverConvergedSuccessfully
```

```

outputsim = struct with fields:
  iterations: 1967
  funccount: 1986
  message: 'Optimization terminated: change in best function value less than options.Funct.
  rngstate: [1x1 struct]
  problemtype: 'unconstrained'
  temperature: [2x1 double]
  totaltime: 1.1838
  solver: 'simulannealbnd'

```

The solver takes about the same number of function evaluations as `particleswarm`, and reaches a good solution. This solver, too, is stochastic.

surrogateopt Solver

`surrogateopt` does not require a start point, but does require finite bounds. Set bounds of -70 to 130 in each component. To have the same sort of output as the other solvers, disable the default plot function.

```

rng default % For reproducibility
x = optimvar("x", "LowerBound", -70, "UpperBound", 130);
y = optimvar("y", "LowerBound", -70, "UpperBound", 130);
prob = optimproblem("Objective", rf3(x,y));
options = optimoptions("surrogateopt", "PlotFcn", []);
[solsur, fvalsur, eflagsur, outputsur] = solve(prob, ...
    "Solver", "surrogateopt", ...
    "Options", options)

```

```

Solving problem using surrogateopt.
surrogateopt stopped because it exceeded the function evaluation limit set by
'options.MaxFunctionEvaluations'.

```

```

solsur = struct with fields:
  x: -1.3383
  y: -0.3022

```

```
fvalsur = 3.5305
```

```

eflagsur =
  SolverLimitExceeded

```

```

outputsur = struct with fields:
  elapsedtime: 4.5763
  funccount: 200
  constrviolation: 0
  ineq: [1x1 struct]
  rngstate: [1x1 struct]
  message: 'surrogateopt stopped because it exceeded the function evaluation limit set
  solver: 'surrogateopt'

```

The solver takes relatively few function evaluations to reach a solution near the global optimum. However, each function evaluation takes much more time than those of the other solvers.

Compare Solvers and Solutions

One solution is better than another if its objective function value is smaller than the other. The following table summarizes the results.

```
sols = [solf.x solf.y;
        solp.x solp.y;
        solg.x solg.y;
        solpso.x solpso.y;
        solsim.x solsim.y;
        solsur.x solsur.y];
fvals = [fvalf;
         fvalp;
         fvalg;
         fvalpso;
         fvalsim;
         fvalsur];
fevals = [outputf.funcCount;
          outputp.funccount;
          outputg.funccount;
          outputpso.funccount;
          outputsim.funccount;
          outputsur.funccount];
stats = table(sols,fvals,fevals);
stats.Properties.RowNames = ["fminunc" "patternsearch" "ga" "particleswarm" "simulannealbnd" "surrogateopt"];
stats.Properties.VariableNames = ["Solution" "Objective" "# Fevals"];
disp(stats)
```

	Solution		Objective	# Fevals
fminunc	19.899	29.849	12.934	5
patternsearch	19.899	-9.9496	4.9748	174
ga	0.0063672	0.00077057	8.1608e-05	9453
particleswarm	7.1467e-07	1.4113e-06	4.9631e-12	2420
simulannealbnd	0.002453	0.0017923	1.8311e-05	1986
surrogateopt	-1.3383	-0.30217	3.5305	200

These results are typical:

- `fminunc` quickly reaches the local solution within its starting basin, but does not explore outside this basin at all. Because the objective function has analytic derivatives, `fminunc` uses automatic differentiation and takes very few function evaluations to reach an accurate local minimum.
- `patternsearch` takes more function evaluations than `fminunc`, and searches through several basins, arriving at a better solution than `fminunc`.
- `ga` takes many more function evaluations than `patternsearch`. By chance it arrives at a better solution. In this case, `ga` finds a point near the global optimum. `ga` is stochastic, so its results change with every run. `ga` requires extra steps to have an initial population near [20,30].
- `particleswarm` takes fewer function evaluations than `ga`, but more than `patternsearch`. In this case, `particleswarm` finds a point with lower objective function value than `patternsearch` or `ga`. Because `particleswarm` is stochastic, its results change with every run. `particleswarm` requires extra steps to have an initial population near [20,30].
- `simulannealbnd` takes about the same number of function evaluations as `particleswarm`. In this case, `simulannealbnd` finds a good solution, but not as good as `particleswarm`. The solver is stochastic and can arrive at a suboptimal solution.

- `surrogateopt` stops when it reaches a function evaluation limit, which by default is 200 for a two-variable problem. `surrogateopt` requires finite bounds. `surrogateopt` attempts to find a global solution, and in this case succeeds. Each function evaluation in `surrogateopt` takes a longer time than in most other solvers, because `surrogateopt` performs many auxiliary computations as part of its algorithm.

See Also

`solve` | `patternsearch` | `ga` | `particleswarm` | `simulannealbnd` | `surrogateopt`

Related Examples

- “Comparison of Six Solvers” on page 1-10

Comparison of Six Solvers

Function to Optimize

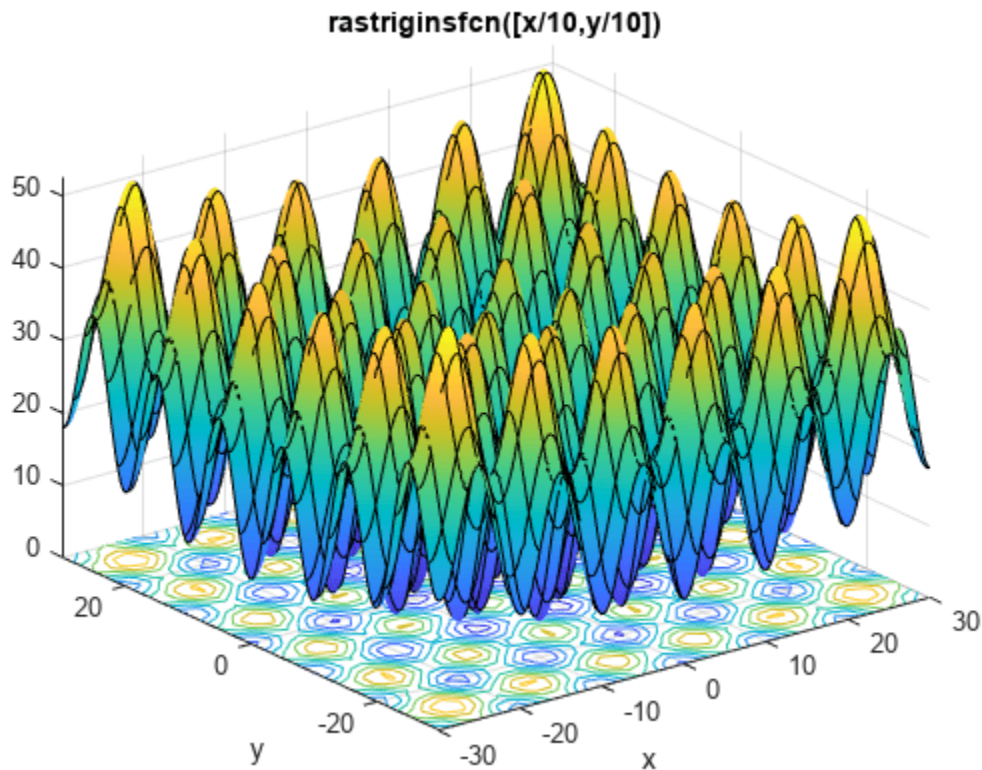
This example shows how to minimize Rastrigin's function with six solvers. Each solver has its own characteristics. The characteristics lead to different solutions and run times. The results, examined in , can help you choose an appropriate solver for your own problems.

Rastrigin's function has many local minima, with a global minimum at (0,0). The function is defined as $Ras(x)$:

$$Ras(x) = 20 + x_1^2 + x_2^2 - 10(\cos 2\pi x_1 + \cos 2\pi x_2).$$

The `rastriginsfcn.m` file, which computes the values of Rastrigin's function, is available when you run this example. This example employs a scaled version of Rastrigin's function with larger basins of attraction. For information, see "Basins of Attraction" on page 1-25. Create a surface plot of the scaled function.

```
rf2 = @(x)rastriginsfcn(x/10);
rf3 = @(x,y)reshape(rastriginsfcn([x(:)/10,y(:)/10]),size(x));
fsurf(rf3,[-30 30],'ShowContours','on')
title('rastriginsfcn([x/10,y/10])')
xlabel('x')
ylabel('y')
```



Usually, you don't know the location of the global minimum of your objective function. To show how the solvers look for a global solution, this example starts all the solvers around the point [20,30], which is far from the global minimum.

This example minimizes `rf2` using the default settings of `fminunc` (an Optimization Toolbox™ solver), `patternsearch`, and `GlobalSearch`. The example also uses `ga` and `particleswarm` with nondefault options to start with an initial population around the point [20,30]. Because `surrogateopt` requires finite bounds, the example uses `surrogateopt` with lower bounds of -70 and upper bounds of 130 in each variable.

Six Solution Methods

`fminunc`

To solve the optimization problem using the `fminunc` Optimization Toolbox solver, enter:

```
rf2 = @(x)rastriginsfcn(x/10); % objective
x0 = [20,30]; % start point away from the minimum
[xf,ff,flf,of] = fminunc(rf2,x0)
```

Local minimum found.

Optimization completed because the size of the gradient is less than the value of the optimality tolerance.

```
xf = 1×2
```

```
    19.8991    29.8486
```

```
ff = 12.9344
```

```
flf = 1
```

```
of = struct with fields:
    iterations: 3
    funcCount: 15
    stepsize: 1.7776e-06
    lssteplength: 1
    firstorderopt: 5.9907e-09
    algorithm: 'quasi-newton'
    message: 'Local minimum found....'
```

- `xf` is the minimizing point.
- `ff` is the value of the objective, `rf2`, at `xf`.
- `flf` is the exit flag. An exit flag of 1 indicates `xf` is a local minimum.
- `of` is the output structure, which describes the `fminunc` calculations leading to the solution.

`patternsearch`

To solve the optimization problem using the `patternsearch` Global Optimization Toolbox solver, enter:

```
rf2 = @(x)rastriginsfcn(x/10); % objective
x0 = [20,30]; % start point away from the minimum
[xp,fp,flp,op] = patternsearch(rf2,x0)
```

```
Optimization terminated: mesh size less than options.MeshTolerance.
xp = 1×2
    19.8991    -9.9496

fp = 4.9748
flp = 1
op = struct with fields:
    function: @(x)rastriginsfcn(x/10)
    problemtype: 'unconstrained'
    pollmethod: 'gpspositivebasis2n'
    maxconstraint: []
    searchmethod: []
    iterations: 48
    funccount: 174
    meshsize: 9.5367e-07
    rngstate: [1x1 struct]
    message: 'Optimization terminated: mesh size less than options.MeshTolerance.'
```

- `xp` is the minimizing point.
- `fp` is the value of the objective, `rf2`, at `xp`.
- `flp` is the exit flag. An exit flag of 1 indicates `xp` is a local minimum.
- `op` is the output structure, which describes the patternsearch calculations leading to the solution.

ga

To solve the optimization problem using the `ga` Global Optimization Toolbox solver, enter:

```
rng default % for reproducibility
rf2 = @(x)rastriginsfcn(x/10); % objective
x0 = [20,30]; % start point away from the minimum
initpop = 10*randn(20,2) + repmat(x0,20,1);
opts = optimoptions('ga','InitialPopulationMatrix',initpop);
```

- `initpop` is a 20-by-2 matrix. Each row of `initpop` has mean `[20,30]`, and each element is normally distributed with standard deviation 10. The rows of `initpop` form an initial population matrix for the `ga` solver.
- `opts` is the options that set `initpop` as the initial population.
- `ga` uses random numbers, and produces a random result.
- The next line calls `ga`, using the options.

```
[xga,fga,flga,oga] = ga(rf2,2,[],[],[],[],[],[],[],[],opts)
```

```
Optimization terminated: maximum number of generations exceeded.
```

```
xga = 1×2
    -0.0042    -0.0024

fga = 4.7054e-05
```

```

flga = 0
oga = struct with fields:
    problemtype: 'unconstrained'
    rngstate: [1x1 struct]
    generations: 200
    funccount: 9453
    message: 'Optimization terminated: maximum number of generations exceeded.'
    maxconstraint: []
    hybridflag: []

```

- `xga` is the minimizing point.
- `fga` is the value of the objective, `rf2`, at `xga`.
- `flga` is the exit flag. An exit flag of 0 indicates that `ga` reaches a function evaluation limit or an iteration limit. In this case, `ga` reaches an iteration limit.
- `oga` is the output structure, which describes the `ga` calculations leading to the solution.

particleswarm

Like `ga`, `particleswarm` is a population-based algorithm. So for a fair comparison of solvers, initialize the particle swarm to the same population as `ga`.

```

rng default % for reproducibility
rf2 = @(x)rastriginsfcn(x/10); % objective
opts = optimoptions('particleswarm','InitialSwarmMatrix',initpop);
[xpso,fpso,flgpso,opso] = particleswarm(rf2,2,[],[],opts)

```

```

Optimization ended: relative change in the objective value
over the last OPTIONS.MaxStallIterations iterations is less than OPTIONS.FunctionTolerance.

```

```

xpso = 1x2
      9.9496      0.0000

```

```

fpso = 0.9950

```

```

flgpso = 1

```

```

opso = struct with fields:
    rngstate: [1x1 struct]
    iterations: 56
    funccount: 1140
    message: 'Optimization ended: relative change in the objective value ...'
    hybridflag: []

```

surrogateopt

`surrogateopt` does not require a start point, but does require finite bounds. Set bounds of -70 to 130 in each component. To have the same sort of output as the other solvers, disable the default plot function.

```

rng default % for reproducibility
lb = [-70,-70];
ub = [130,130];
rf2 = @(x)rastriginsfcn(x/10); % objective

```

```
opts = optimoptions('surrogateopt','PlotFcn',[]);
[xsur,fsur,flgsur,osur] = surrogateopt(rf2,lb,ub,opts)

surrogateopt stopped because it exceeded the function evaluation limit set by
'options.MaxFunctionEvaluations'.

xsur = 1x2
    -1.3383    -0.3022

fsur = 3.5305

flgsur = 0

osur = struct with fields:
    elapsedtime: 4.4570
    funccount: 200
    constrviolation: 0
    ineq: [1x0 double]
    rngstate: [1x1 struct]
    message: 'surrogateopt stopped because it exceeded the function evaluation limit set
```

- `xsur` is the minimizing point.
- `fsur` is the value of the objective, `rf2`, at `xsur`.
- `flgsur` is the exit flag. An exit flag of 0 indicates that `surrogateopt` halted because it ran out of function evaluations or time.
- `osur` is the output structure, which describes the `surrogateopt` calculations leading to the solution.

GlobalSearch

To solve the optimization problem using the `GlobalSearch` solver, enter:

```
rf2 = @(x)rastriginsfcn(x/10); % objective
x0 = [20,30]; % start point away from the minimum
problem = createOptimProblem('fmincon','objective',rf2,...
    'x0',x0);
gs = GlobalSearch;
```

`problem` is an optimization problem structure. `problem` specifies the `fmincon` solver, the `rf2` objective function, and `x0=[20,30]`. For more information on using `createOptimProblem`, see “Create Problem Structure” on page 4-4.

Note: You must specify `fmincon` as the solver for `GlobalSearch`, even for unconstrained problems.

`gs` is a default `GlobalSearch` object. The object contains options for solving the problem. Calling `run(gs,problem)` runs `problem` from multiple start points. The start points are random, so the following result is also random.

```
[xg,fg,flg,og] = run(gs,problem)
```

```
GlobalSearch stopped because it analyzed all the trial points.
```

```
All 6 local solver runs converged with a positive local solver exit flag.
```

```

xg = 1×2
10-7 ×

    -0.1405    -0.1405

fg = 0

flg = 1

og = struct with fields:
    funcCount: 2157
    localSolverTotal: 6
    localSolverSuccess: 6
    localSolverIncomplete: 0
    localSolverNoSolution: 0
    message: 'GlobalSearch stopped because it analyzed all the trial points....'

```

- `xg` is the minimizing point.
- `fg` is the value of the objective, `rf2`, at `xg`.
- `flg` is the exit flag. An exit flag of 1 indicates all `fmincon` runs converged properly.
- `og` is the output structure, which describes the `GlobalSearch` calculations leading to the solution.

Compare Syntax and Solutions

One solution is better than another if its objective function value is smaller than the other. The following table summarizes the results.

```

sols = [xf;
    xp;
    xga;
    xps0;
    xsur;
    xg];
fvals = [ff;
    fp;
    fga;
    fps0;
    fsur;
    fg];
fevals = [of.funcCount;
    op.funccount;
    oga.funccount;
    opso.funccount;
    osur.funccount;
    og.funcCount];
stats = table(sols,fvals,fevals);
stats.Properties.RowNames = ["fminunc" "patternsearch" "ga" "particleswarm" "surrogateopt" "GlobalSearch"];
stats.Properties.VariableNames = ["Solution" "Objective" "# Fevals"];
disp(stats)

```

	Solution		Objective	# Fevals
	_____	_____	_____	_____
fminunc	19.899	29.849	12.934	15

patternsearch	19.899	-9.9496	4.9748	174
ga	-0.0042178	-0.0024347	4.7054e-05	9453
particleswarm	9.9496	6.75e-07	0.99496	1140
surrogateopt	-1.3383	-0.30217	3.5305	200
GlobalSearch	-1.4046e-08	-1.4046e-08	0	2157

These results are typical:

- `fminunc` quickly reaches the local solution within its starting basin, but does not explore outside this basin at all. `fminunc` has a simple calling syntax.
- `patternsearch` takes more function evaluations than `fminunc`, and searches through several basins, arriving at a better solution than `fminunc`. The `patternsearch` calling syntax is the same as that of `fminunc`.
- `ga` takes many more function evaluations than `patternsearch`. By chance it arrives at a better solution. In this case, `ga` finds a point near the global optimum. `ga` is stochastic, so its results change with every run. `ga` has a simple calling syntax, but there are extra steps to have an initial population near `[20, 30]`.
- `particleswarm` takes fewer function evaluations than `ga`, but more than `patternsearch`. In this case, `particleswarm` finds a point with lower objective function value than `patternsearch`, but higher than `ga`. Because `particleswarm` is stochastic, its results change with every run. `particleswarm` has a simple calling syntax, but there are extra steps to have an initial population near `[20, 30]`.
- `surrogateopt` stops when it reaches a function evaluation limit, which by default is 200 for a two-variable problem. `surrogateopt` has a simple calling syntax, but requires finite bounds. `surrogateopt` attempts to find a global solution, but in this case does not succeed. Each function evaluation in `surrogateopt` takes a longer time than in most other solvers, because `surrogateopt` performs many auxiliary computations as part of its algorithm.
- `GlobalSearch` run takes the same order of magnitude of function evaluations as `ga` and `particleswarm`, searches many basins, and arrives at a good solution. In this case, `GlobalSearch` finds the global optimum. Setting up `GlobalSearch` is more involved than setting up the other solvers. As the example shows, before calling `GlobalSearch`, you must create both a `GlobalSearch` object (`gs` in the example), and a problem structure (`problem`). Then, you call the `run` method with `gs` and `problem`. For more details on how to run `GlobalSearch`, see “Workflow for `GlobalSearch` and `MultiStart`” on page 4-3.

See Also

`GlobalSearch` | `patternsearch` | `ga` | `surrogateopt` | `particleswarm` | `fminunc`

More About

- “Compare Several Global Solvers, Problem-Based” on page 1-3
- “Solver-Based Optimization Problem Setup”
- “Solver Behavior with a Nonsmooth Problem” on page 1-17

Solver Behavior with a Nonsmooth Problem

This example shows the importance of choosing an appropriate solver for optimization problems. It also shows that a single point of non-smoothness can cause problems for Optimization Toolbox™ solvers.

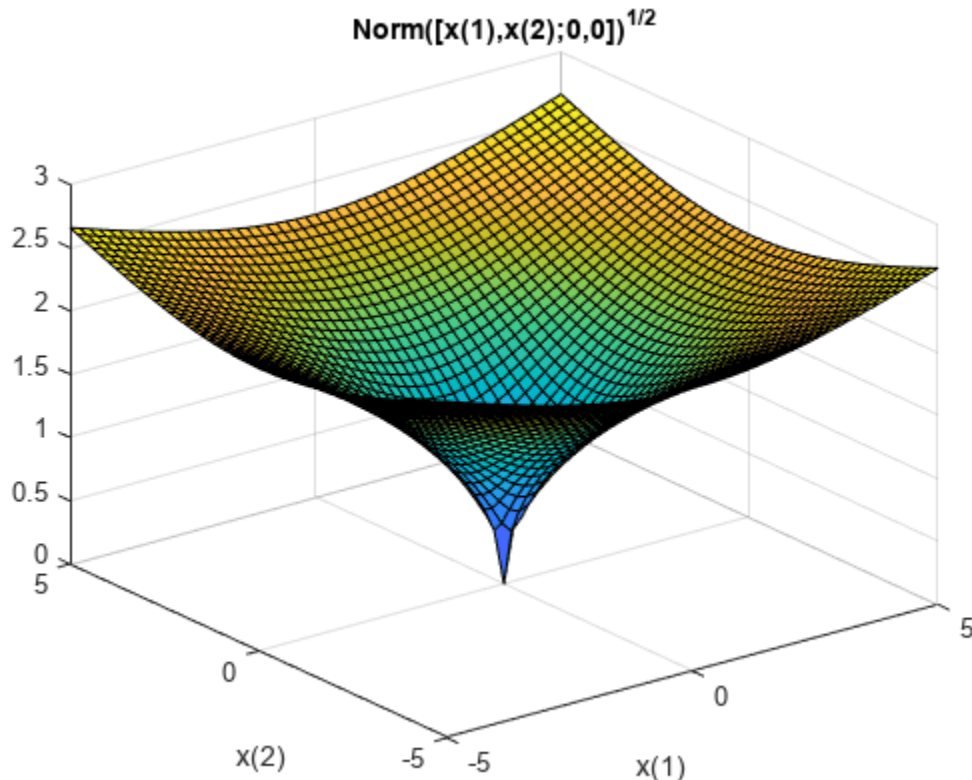
In general, the solver decision tables provide guidance on which solver is likely to work best for your problem. For smooth problems, see “Optimization Decision Table”. For nonsmooth problems, see “Table for Choosing a Solver” on page 1-29 first, and for more information consult “Global Optimization Toolbox Solver Characteristics” on page 1-30.

A Function with a Single Nonsmooth Point

The function $f(x) = ||x||^{1/2}$ is nonsmooth at the point 0, which is the minimizing point. Here is a 2-D plot using the *matrix norm* for the 4-D point $\begin{bmatrix} x(1) & x(2) \\ 0 & 0 \end{bmatrix}$.

```
figure
x = linspace(-5,5,51);
[xx,yy] = meshgrid(x);
zz = zeros(size(xx));
for ii = 1:length(x)
    for jj = 1:length(x)
        zz(ii,jj) = sqrt(norm([xx(ii,jj),yy(ii,jj);0,0]));
    end
end

surf(xx,yy,zz)
xlabel('x(1)')
ylabel('x(2)')
title('Norm([x(1),x(2);0,0])^{1/2}')
```



This example uses matrix norm for a 2-by-6 matrix x . The matrix norm relates to the singular value decomposition, which is not as smooth as the Euclidean norm. See “2-Norm of Matrix”.

Minimize Using patternsearch

`patternsearch` is the recommended first solver to try for nonsmooth problems. See “Table for Choosing a Solver” on page 1-29. Start `patternsearch` from a nonzero 2-by-6 matrix x_0 , and attempt to locate the minimum of f . For this attempt, and all others, use the default solver options.

Return the solution, which should be near zero, the objective function value, which should likewise be near zero, and the number of function evaluations taken.

```
fun = @(x)norm([x(1:6);x(7:12)])^(1/2);
x0 = [1:6;7:12];
rng default
x0 = x0 + rand(size(x0))
```

```
x0 = 2×6
```

```
    1.8147    2.1270    3.6324    4.2785    5.9575    6.1576
    7.9058    8.9134    9.0975   10.5469   11.9649   12.9706
```

```
[xps,fvalps,eflagps,outputps] = patternsearch(fun,x0);
```

```
Optimization terminated: mesh size less than options.MeshTolerance.
```

```
xps,fvalps,eflagps,outputps.funccount
```

```
xps = 2×6
10-4 ×

    0.1116   -0.1209    0.3503   -0.0520   -0.1270    0.2031
   -0.3082   -0.1526    0.0623    0.0652    0.4479    0.1173
```

```
fvalps = 0.0073
```

```
eflagps = 1
```

```
ans = 10780
```

`patternsearch` reaches a good solution, as evinced by exit flag 1. However, it takes over 10,000 function evaluations to converge.

Minimize Using `fminsearch`

The documentation states that `fminsearch` sometimes can handle discontinuities, so this is a reasonable option.

```
[xfms,fvalfms,eflagfms,outputfms] = fminsearch(fun,x0);
```

```
Exiting: Maximum number of function evaluations has been exceeded
        - increase MaxFunEvals option.
        Current function value: 3.197063
```

```
xfms,fvalfms,eflagfms,outputfms.funcCount
```

```
xfms = 2×6

    2.2640    1.1747    9.0693    8.1652    1.7367   -1.2958
    3.7456    1.2694    0.2714   -3.7942    3.8714    1.9290
```

```
fvalfms = 3.1971
```

```
eflagfms = 0
```

```
ans = 2401
```

Using default options, `fminsearch` runs out of function evaluations before it converges to a solution. Exit flag 0 indicates this lack of convergence. The reported solution is poor.

Use `particleswarm`

`particleswarm` is recommended as the next solver to try. See “Choosing Between Solvers for Nonsmooth Problems” on page 1-32.

```
[xpsw,fvalpsw,eflagpsw,outputpsw] = particleswarm(fun,12);
```

```
Optimization ended: relative change in the objective value
over the last OPTIONS.MaxStallIterations iterations is less than OPTIONS.FunctionTolerance.
```

```
xpsw,fvalpsw,eflagpsw,outputpsw.funccount
```

```
xpsw = 1×12
10-12 ×
```

```
-0.0386 -0.1282 -0.0560 0.0904 0.0771 -0.0541 -0.1189 0.1290 -0.0032 0.0000  
fvalpsw = 4.5222e-07  
eflagpsw = 1  
ans = 37200
```

`particleswarm` finds an even more accurate solution than `patternsearch`, but takes over 35,000 function evaluations. Exit flag 1 indicates that the solution is good.

Use `ga`

`ga` is a popular solver, but is not recommended as the first solver to try. See how well it works on this problem.

```
[xga,fvalga,eflagga,outputga] = ga(fun,12);  
Optimization terminated: average change in the fitness value less than options.FunctionTolerance  
xga,fvalga,eflagga,outputga.funcCount  
xga = 1×12  
-0.0061 -0.0904 0.0816 -0.0484 0.0799 -0.1925 0.0048 0.3581 0.0848 0.0000  
fvalga = 0.6257  
eflagga = 1  
ans = 65190
```

`ga` does not find as good a solution as `patternsearch` or `particleswarm`, and takes about twice as many function evaluations as `particleswarm`. Exit flag 1 is misleading in this case.

Use `fminunc` from Optimization Toolbox

`fminunc` is not recommended for nonsmooth functions. See how it performs on this one.

```
[xfmu,fvalfmu,eflagfmu,outputfmu] = fminunc(fun,x0);  
Local minimum possible.  
fminunc stopped because the size of the current step is less than  
the value of the step size tolerance.  
xfmu,fvalfmu,eflagfmu,outputfmu.funcCount  
xfmu = 2×6  
-0.5844 -0.9726 -0.4356 0.1467 0.3263 -0.1002  
-0.0769 -0.1092 -0.3429 -0.6856 -0.7609 -0.6524  
fvalfmu = 1.1269  
eflagfmu = 2  
ans = 429
```

The `fminunc` solution is not as good as the `ga` solution. However, `fminunc` reaches the rather poor solution in relatively few function evaluations. Exit flag 2 means you should take care, the first-order optimality conditions are not met at the reported solution.

Use `fmincon` from Optimization Toolbox

`fmincon` can sometimes minimize nonsmooth functions. See how it performs on this one.

```
[xfmc,fvalfmc,eflagfmc,outputfmc] = fmincon(fun,x0);
```

Local minimum possible. Constraints satisfied.

`fmincon` stopped because the size of the current step is less than the value of the step size tolerance and constraints are satisfied to within the value of the constraint tolerance.

```
xfmc,fvalfmc,eflagfmc,outputfmc.funcCount
```

```
xfmc = 2×6  
10-10 ×
```

```
    0.2066    -0.3225     0.0684     0.2739    -0.2820    -0.2964  
    0.4100     0.3742    -0.0238     0.5316     0.0990    -0.0954
```

```
fvalfmc = 9.7096e-06
```

```
eflagfmc = 2
```

```
ans = 1020
```

`fmincon` with default options produces an accurate solution after fewer than 1000 function evaluations. Exit flag 2 does not mean that the solution is inaccurate, but that the first-order optimality conditions are not met. This is because the gradient of the objective function is not zero at the solution.

Summary of Results

Choosing the appropriate solver leads to better, faster results. This summary shows how disparate the results can be. The solution quality is 'Poor' if the objective function value is greater than 0.1, 'Good' if the value is smaller than 0.01, and 'Mediocre' otherwise.

```
Solver = {'patternsearch';'fminsearch';'particleswarm';'ga';'fminunc';'fmincon'};  
SolutionQuality = {'Good';'Poor';'Good';'Poor';'Poor';'Good'};  
FVal = [fvalps,fvalfms,fvalpsw,fvalga,fvalfmu,fvalfmc]';  
NumEval = [outputps.funcCount,outputfms.funcCount,outputpsw.funcCount,...  
           outputga.funcCount,outputfmu.funcCount,outputfmc.funcCount]';  
results = table(Solver,SolutionQuality,FVal,NumEval)
```

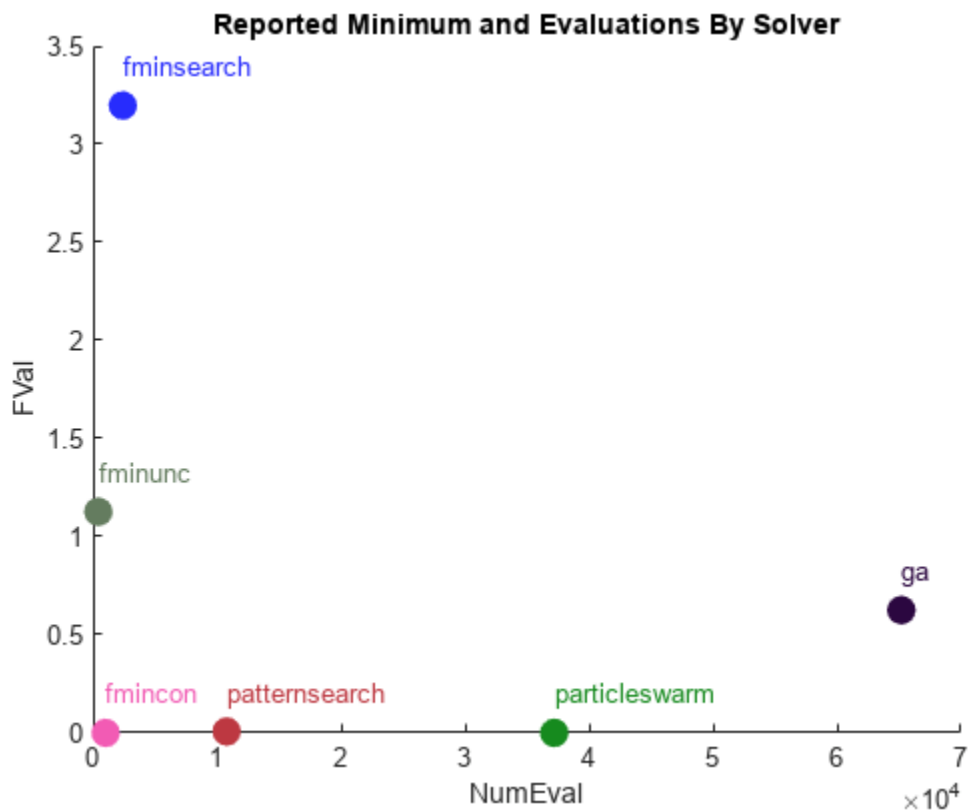
```
results=6×4 table
```

Solver	SolutionQuality	FVal	NumEval
{'patternsearch'}	{'Good'}	0.0072656	10780
{'fminsearch' }	{'Poor'}	3.1971	2401
{'particleswarm'}	{'Good'}	4.5222e-07	37200
{'ga' }	{'Poor'}	0.62572	65190
{'fminunc' }	{'Poor'}	1.1269	429

```
{'fmincon' } {'Good'} 9.7096e-06 1020
```

Another view of the results.

```
figure
hold on
for ii = 1:length(FVal)
    clr = rand(1,3);
    plot(NumEval(ii),FVal(ii),'o','MarkerSize',10,'MarkerEdgeColor',clr,'MarkerFaceColor',clr)
    text(NumEval(ii),FVal(ii)+0.2,Solver{ii},'Color',clr);
end
ylabel('FVal')
xlabel('NumEval')
title('Reported Minimum and Evaluations By Solver')
hold off
```



While particleswarm achieves the lowest objective function value, it does so by taking over three times as many function evaluations as patternsearch, and over 30 times as many as fmincon.

fmincon is not generally recommended for nonsmooth problems. It is effective in this case, but this case has just one nonsmooth point.

See Also

More About

- “Comparison of Six Solvers” on page 1-10
- “Table for Choosing a Solver” on page 1-29
- “Global Optimization Toolbox Solver Characteristics” on page 1-30

What Is Global Optimization?

In this section...

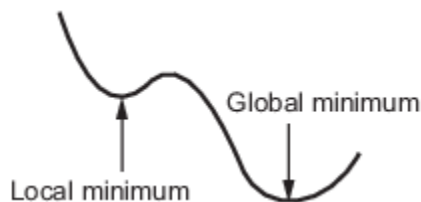
“Local vs. Global Optima” on page 1-24

“Basins of Attraction” on page 1-25

Local vs. Global Optima

Optimization is the process of finding the point that minimizes a function. More specifically:

- A local minimum of a function is a point where the function value is smaller than or equal to the value at nearby points, but possibly greater than at a distant point.
- A global minimum is a point where the function value is smaller than or equal to the value at all other feasible points.



Generally, Optimization Toolbox™ solvers find a local optimum. (This local optimum can be a global optimum.) They find the optimum in the basin of attraction of the starting point. For more information, see “Basins of Attraction” on page 1-25.

In contrast, Global Optimization Toolbox solvers are designed to search through more than one basin of attraction. They search in various ways:

- `GlobalSearch` and `MultiStart` generate a number of starting points. They then use a local solver to find the optima in the basins of attraction of the starting points.
- `ga` uses a set of starting points (called the population) and iteratively generates better points from the population. As long as the initial population covers several basins, `ga` can examine several basins.
- `particleswarm`, like `ga`, uses a set of starting points. `particleswarm` can examine several basins at once because of its diverse population.
- `simulannealbnd` performs a random search. Generally, `simulannealbnd` accepts a point if it is better than the previous point. `simulannealbnd` occasionally accepts a worse point, in order to reach a different basin.
- `patternsearch` looks at a number of neighboring points before accepting one of them. If some neighboring points belong to different basins, `patternsearch` in essence looks in a number of basins at once.
- `surrogateopt` begins by quasirandom sampling within bounds, looking for a small objective function value. `surrogateopt` uses a merit function that, in part, gives preference to points that are far from evaluated points, which is an attempt to reach a global solution. After it cannot improve the current point, `surrogateopt` resets, causing it to sample widely within bounds again. Resetting is another way `surrogateopt` searches for a global solution.

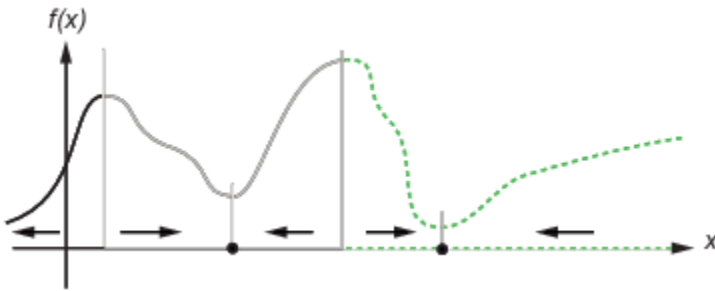
Basins of Attraction

If an objective function $f(x)$ is smooth, the vector $-\nabla f(x)$ points in the direction where $f(x)$ decreases most quickly. The equation of steepest descent, namely

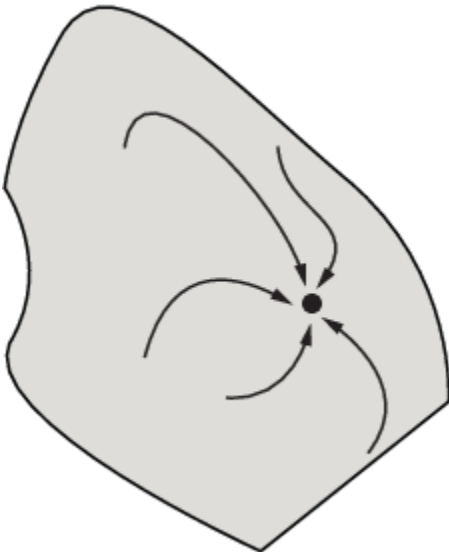
$$\frac{d}{dt}x(t) = -\nabla f(x(t)),$$

yields a path $x(t)$ that goes to a local minimum as t gets large. Generally, initial values $x(0)$ that are close to each other give steepest descent paths that tend to the same minimum point. The basin of attraction for steepest descent is the set of initial values leading to the same local minimum.

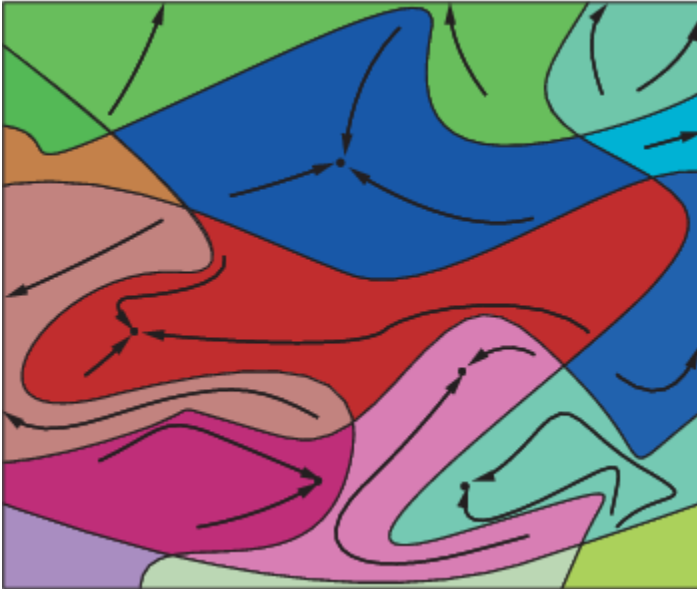
The following figure shows two one-dimensional minima. The figure shows different basins of attraction with different line styles, and it shows directions of steepest descent with arrows. For this and subsequent figures, black dots represent local minima. Every steepest descent path, starting at a point $x(0)$, goes to the black dot in the basin containing $x(0)$.



The following figure shows how steepest descent paths can be more complicated in more dimensions.



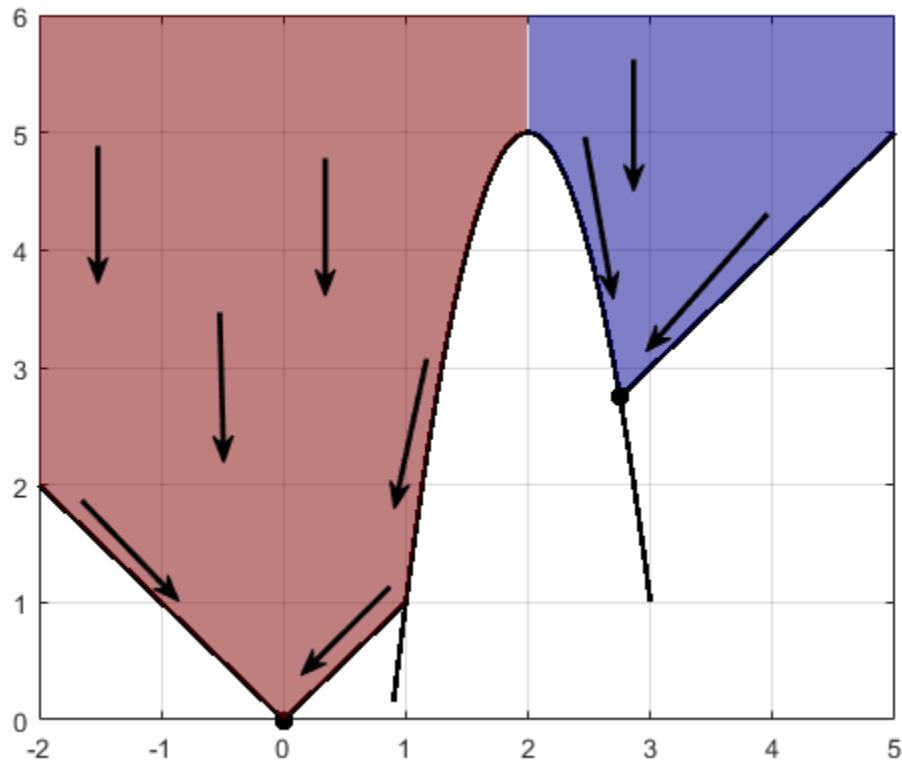
The following figure shows even more complicated paths and basins of attraction.



Constraints can break up one basin of attraction into several pieces. For example, consider minimizing y subject to:

- $y \geq |x|$
- $y \geq 5 - 4(x-2)^2$.

The figure shows the two basins of attraction with the final points.



The steepest descent paths are straight lines down to the constraint boundaries. From the constraint boundaries, the steepest descent paths travel down along the boundaries. The final point is either $(0,0)$ or $(11/4,11/4)$, depending on whether the initial x -value is above or below 2.

See Also

More About

- “Visualize the Basins of Attraction” on page 4-24
- “Comparison of Six Solvers” on page 1-10

Optimization Workflow

To solve an optimization problem:

- 1** Decide what type of problem you have, and whether you want a local or global solution (see “Local vs. Global Optima” on page 1-24). Choose a solver per the recommendations in “Table for Choosing a Solver” on page 1-29.
- 2** Write your objective function and, if applicable, constraint functions per the syntax in “Compute Objective Functions” on page 2-2 and “Write Constraints” on page 2-6.
- 3** Set appropriate options using `optimoptions`, or prepare a `GlobalSearch` or `MultiStart` problem as described in “Workflow for GlobalSearch and MultiStart” on page 4-3. For details, see “Pattern Search Options” on page 17-7, “Particle Swarm Options” on page 17-45, “Genetic Algorithm Options” on page 17-23, “Simulated Annealing Options” on page 17-58, or “Surrogate Optimization Options” on page 17-51.
- 4** Run the solver.
- 5** Examine the result. For information on the result, see “Solver Outputs and Iterative Display” or Examine Results for `GlobalSearch` or `MultiStart`.
- 6** If the result is unsatisfactory, change options or start points or otherwise update your optimization and rerun it. For information, see “Global Optimization Toolbox Solver Characteristics” on page 1-30 or Improve Results. For information on improving solutions that applies mainly to smooth problems, see “When the Solver Fails”, “When the Solver Might Have Succeeded”, or “When the Solver Succeeds”.

See Also

More About

- “Solver-Based Optimization Problem Setup”
- “What Is Global Optimization?” on page 1-24

Table for Choosing a Solver

Choose a solver based on problem characteristics and on the type of solution you want. “Solver Characteristics” on page 1-33 contains more information to help you decide which solver is likely to be most suitable. This table gives recommendations that are suitable for most problems.

Problem Type	Recommended Solver
Smooth (objective twice differentiable), and you want a local solution	An appropriate Optimization Toolbox solver; see “Optimization Decision Table”
Smooth (objective twice differentiable), and you want a global solution or multiple local solutions	GlobalSearch or MultiStart
Nonsmooth, and you want a local solution	patternsearch
Nonsmooth, and you want a global solution or multiple local solutions	surrogateopt or patternsearch with several initial points x_0

To start `patternsearch` at multiple points when you have finite bounds `lb` and `ub` on every component, try:

```
x0 = lb + rand(size(lb)).*(ub - lb);
```

Many other solvers provide different solution algorithms, including the genetic algorithm solver `ga` and the `particleswarm` solver. Try some of them if the recommended solvers do not perform well on your problem. For details, see “Global Optimization Toolbox Solver Characteristics” on page 1-30.

See Also

Related Examples

- “Solver Behavior with a Nonsmooth Problem” on page 1-17

More About

- “Optimization Workflow” on page 1-28
- “Global Optimization Toolbox Solver Characteristics” on page 1-30

Global Optimization Toolbox Solver Characteristics

In this section...
“Solver Choices” on page 1-30
“Explanation of “Desired Solution”” on page 1-30
“Choosing Between Solvers for Smooth Problems” on page 1-32
“Choosing Between Solvers for Nonsmooth Problems” on page 1-32
“Solver Characteristics” on page 1-33
“Why Are Some Solvers Objects?” on page 1-35

Solver Choices

This section describes Global Optimization Toolbox solver characteristics. The section includes recommendations for obtaining results more effectively.

To achieve better or faster solutions, first try tuning the recommended solvers on page 1-29 by setting appropriate options or bounds. If the results are unsatisfactory, try other solvers.

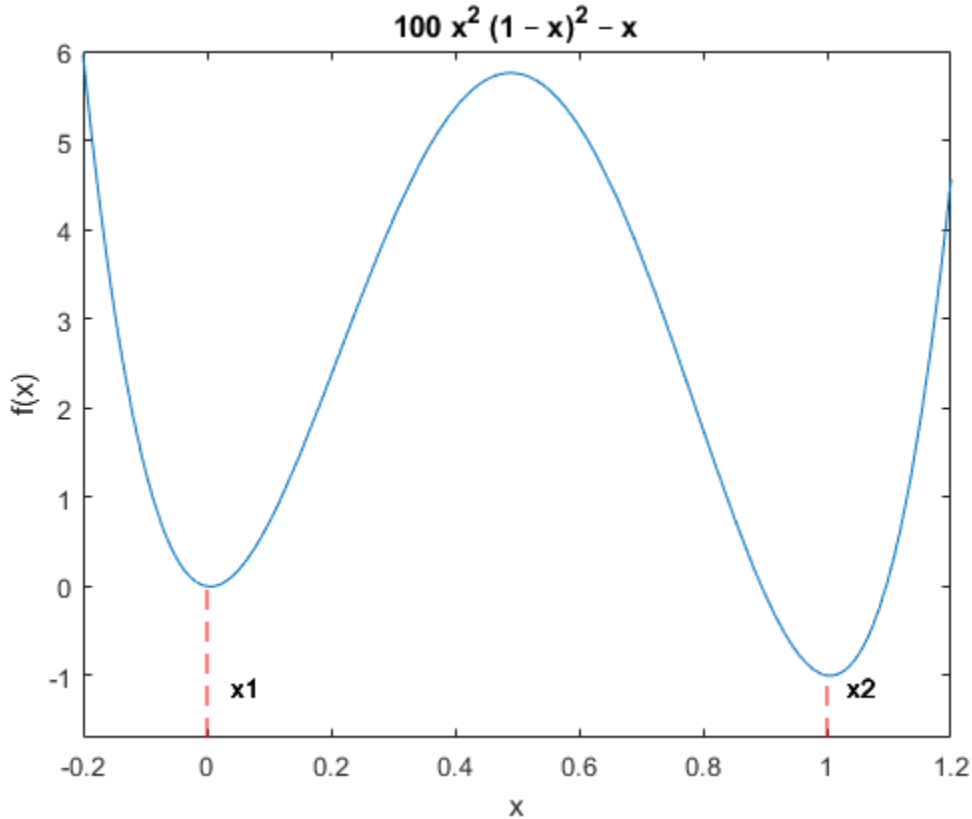
Desired Solution	Smooth Objective and Constraints	Nonsmooth Objective or Constraints
“Explanation of “Desired Solution”” on page 1-30	“Choosing Between Solvers for Smooth Problems” on page 1-32	“Choosing Between Solvers for Nonsmooth Problems” on page 1-32
Single local solution	Optimization Toolbox functions; see “Optimization Decision Table”	fminbnd, patternsearch, fminsearch, ga, particleswarm, simulannealbnd, surrogateopt
Multiple local solutions	GlobalSearch, MultiStart	patternsearch, ga, particleswarm, simulannealbnd, or surrogateopt started from multiple initial points x_0 or from multiple initial populations
Single global solution	GlobalSearch, MultiStart, patternsearch, particleswarm, ga, simulannealbnd, surrogateopt	patternsearch, ga, particleswarm, simulannealbnd, surrogateopt
Single local solution using parallel processing	MultiStart, Optimization Toolbox functions	patternsearch, ga, particleswarm, surrogateopt
Multiple local solutions using parallel processing	MultiStart	patternsearch, ga, or particleswarm started from multiple initial points x_0 or from multiple initial populations
Single global solution using parallel processing	MultiStart	patternsearch, ga, particleswarm, surrogateopt

Explanation of “Desired Solution”

To understand the meaning of the terms in “Desired Solution,” consider the example

$$f(x) = 100x^2(1-x)^2 - x,$$

which has local minima x_1 near 0 and x_2 near 1:



The minima are located at:

```
fun = @(x)(100*x^2*(x - 1)^2 - x);
x1 = fminbnd(fun,-0.1,0.1)
x1 =
    0.0051
```

```
x2 = fminbnd(fun,0.9,1.1)
x2 =
    1.0049
```

Description of the Terms

Term	Meaning
Single local solution	Find one local solution, a point x where the objective function $f(x)$ is a local minimum. For more details, see “Local vs. Global Optima” on page 1-24. In the example, both x_1 and x_2 are local solutions.
Multiple local solutions	Find a set of local solutions. In the example, the complete set of local solutions is $\{x_1, x_2\}$.
Single global solution	Find the point x where the objective function $f(x)$ is a global minimum. In the example, the global solution is x_2 .

Choosing Between Solvers for Smooth Problems

- “Single Global Solution” on page 1-32
- “Multiple Local Solutions” on page 1-32

Single Global Solution

- 1 Try `GlobalSearch` first. It is most focused on finding a global solution, and has an efficient local solver, `fmincon`.
- 2 Try `MultiStart` next. It has efficient local solvers, and can search a wide variety of start points.
- 3 Try `patternsearch` next. It is less efficient, since it does not use gradients. However, `patternsearch` is robust and is more efficient than the remaining local solvers. To search for a global solution, start `patternsearch` from a variety of start points.
- 4 Try `surrogateopt` next. `surrogateopt` attempts to find a global solution using the fewest objective function evaluations. `surrogateopt` has more overhead per function evaluation than most other solvers. `surrogateopt` requires finite bounds, and accepts integer constraints, linear constraints, and nonlinear inequality constraints.
- 5 Try `particleswarm` next, if your problem is unconstrained or has only bound constraints. Usually, `particleswarm` is more efficient than the remaining solvers, and can be more efficient than `patternsearch`.
- 6 Try `ga` next. It can handle all types of constraints, and is usually more efficient than `simulannealbnd`.
- 7 Try `simulannealbnd` last. It can handle problems with no constraints or bound constraints. `simulannealbnd` is usually the least efficient solver. However, given a slow enough cooling schedule, it can find a global solution.

Multiple Local Solutions

`GlobalSearch` and `MultiStart` both provide multiple local solutions. For the syntax to obtain multiple solutions, see “Multiple Solutions” on page 4-17. `GlobalSearch` and `MultiStart` differ in the following characteristics:

- `MultiStart` can find more local minima. This is because `GlobalSearch` rejects many generated start points (initial points for local solution). Essentially, `GlobalSearch` accepts a start point only when it determines that the point has a good chance of obtaining a global minimum. In contrast, `MultiStart` passes all generated start points to a local solver. For more information, see “GlobalSearch Algorithm” on page 4-35.
- `MultiStart` offers a choice of local solver: `fmincon`, `fminunc`, `lsqcurvefit`, or `lsqnonlin`. The `GlobalSearch` solver uses only `fmincon` as its local solver.
- `GlobalSearch` uses a scatter-search algorithm for generating start points. In contrast, `MultiStart` generates points uniformly at random within bounds, or allows you to provide your own points.
- `MultiStart` can run in parallel. See “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11.

Choosing Between Solvers for Nonsmooth Problems

Choose the applicable solver with the lowest number. For problems with integer constraints, use `ga`.

- 1 Use `fminbnd` first on one-dimensional bounded problems only. `fminbnd` provably converges quickly in one dimension.
- 2 Use `patternsearch` on any other type of problem. `patternsearch` provably converges, and handles all types of constraints.
- 3 Try `surrogateopt` for problems that have time-consuming objective functions. `surrogateopt` searches for a global solution. `surrogateopt` requires finite bounds, and accepts integer constraints, linear constraints, and nonlinear inequality constraints.
- 4 Try `fminsearch` next for low-dimensional unbounded problems. `fminsearch` is not as general as `patternsearch` and can fail to converge. For low-dimensional problems, `fminsearch` is simple to use, since it has few tuning options.
- 5 Try `particleswarm` next on unbounded or bound-constrained problems. `particleswarm` has little supporting theory, but is often an efficient algorithm.
- 6 Try `ga` next. `ga` has little supporting theory and is often less efficient than `patternsearch` or `particleswarm`. `ga` handles all types of constraints. `ga` and `surrogateopt` are the only Global Optimization Toolbox solvers that accept integer constraints.
- 7 Try `simulannealbnd` last for unbounded problems, or for problems with bounds. `simulannealbnd` provably converges only for a logarithmic cooling schedule, which is extremely slow. `simulannealbnd` takes only bound constraints, and is often less efficient than `ga`.

Solver Characteristics

Solver	Convergence	Characteristics
GlobalSearch	Fast convergence to local optima for smooth problems	Deterministic iterates
		Gradient-based
		Automatic stochastic start points
		Removes many start points heuristically
MultiStart	Fast convergence to local optima for smooth problems	Deterministic iterates
		Can run in parallel; see “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11
		Gradient-based
		Stochastic or deterministic start points, or combination of both
		Automatic stochastic start points
		Runs all start points
		Choice of local solver: <code>fmincon</code> , <code>fminunc</code> , <code>lsqcurvefit</code> , or <code>lsqnonlin</code>
patternsearch	Proven convergence to local optimum; slower than gradient-based solvers	Deterministic iterates
		Can run in parallel; see “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11
		No gradients

Solver	Convergence	Characteristics
surrogateopt	Proven convergence to global optimum for bounded problems; slower than gradient-based solvers; generally stops by reaching a function evaluation limit or other limit	User-supplied start point
		Stochastic iterates
		Can run in parallel; see “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11
		Best used for time-consuming objective functions
		Requires bound constraints, accepts linear constraints and nonlinear inequality constraints
		Allows integer constraints; see “Mixed-Integer Surrogate Optimization” on page 11-62
		No gradients Automatic start points or user-supplied points, or a combination of both
particleswarm	No convergence proof	Stochastic iterates
		Can run in parallel; see “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11
		Population-based
		No gradients
		Automatic start population or user-supplied population, or a combination of both
		Only bound constraints
ga	No convergence proof	Stochastic iterates
		Can run in parallel; see “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11
		Population-based
		No gradients
		Allows integer constraints; see “Mixed Integer ga Optimization” on page 8-40
		Automatic start population or user-supplied population, or a combination of both
simulannealbnd	Proven to converge to global optimum for bounded problems with very slow cooling schedule	Stochastic iterates
		No gradients
		User-supplied start point
		Only bound constraints

Explanation of some characteristics:

- **Convergence** — Solvers can fail to converge to any solution when started far from a local minimum. When started near a local minimum, gradient-based solvers converge to a local minimum quickly for smooth problems. `patternsearch` provably converges for a wide range of problems, but the convergence is slower than gradient-based solvers. Both `ga` and `simulannealbnd` can fail to converge in a reasonable amount of time for some problems, although they are often effective.
- **Iterates** — Solvers iterate to find solutions. The steps in the iteration are iterates. Some solvers have deterministic iterates. Others use random numbers and have stochastic iterates.
- **Gradients** — Some solvers use estimated or user-supplied derivatives in calculating the iterates. Other solvers do not use or estimate derivatives, but use only objective and constraint function values.
- **Start points** — Most solvers require you to provide a starting point for the optimization in order to obtain the dimension of the decision variables. `ga` and `surrogateopt` do not require any starting points, because they take the dimension of the decision variables as an input or infer dimensions from bounds. These solvers generate a start point or population automatically, or they accept a point or points that you supply.

Compare the characteristics of Global Optimization Toolbox solvers to Optimization Toolbox solvers.

Solver	Convergence	Characteristics
fmincon, fminunc, fseminf, lsqcurvefit, lsqnonlin	Proven quadratic convergence to local optima for smooth problems	Deterministic iterates
		Gradient-based
		User-supplied starting point
fminsearch	No convergence proof — counterexamples exist.	Deterministic iterates
		No gradients
		User-supplied start point
		No constraints
fminbnd	Proven convergence to local optima for smooth problems, slower than quadratic.	Deterministic iterates
		No gradients
		User-supplied start interval
		Only one-dimensional problems

All these Optimization Toolbox solvers:

- Have deterministic iterates
- Require a start point or interval
- Search just one basin of attraction

Why Are Some Solvers Objects?

`GlobalSearch` and `MultiStart` are objects. What does this mean for you?

- You create a `GlobalSearch` or `MultiStart` object before running your problem.
- You can reuse the object for running multiple problems.

- `GlobalSearch` and `MultiStart` objects are containers for algorithms and global options. You use these objects to run a local solver multiple times. The local solver has its own options.

For more information, see the “Classes” documentation.

See Also

Related Examples

- “Solver Behavior with a Nonsmooth Problem” on page 1-17

More About

- “Optimization Workflow” on page 1-28
- “Table for Choosing a Solver” on page 1-29

Write Files for Optimization Functions

- “Compute Objective Functions” on page 2-2
- “Maximizing vs. Minimizing” on page 2-5
- “Write Constraints” on page 2-6
- “Set and Change Options” on page 2-9
- “View Options” on page 2-10

Compute Objective Functions

In this section...

“Objective (Fitness) Functions” on page 2-2

“Write a Function File” on page 2-2

“Write a Vectorized Function” on page 2-3

“Gradients and Hessians” on page 2-4

Objective (Fitness) Functions

To use Global Optimization Toolbox functions, first write a file (or an anonymous function) that computes the function you want to optimize. This is called an objective function for most solvers, or fitness function for `ga`. The function should accept a vector, whose length is the number of independent variables, and return a scalar. For `gamultiobj`, the function should return a row vector of objective function values. For vectorized solvers, the function should accept a matrix, where each row represents one input vector, and return a vector of objective function values. This section shows how to write the file.

Write a Function File

This example shows how to write a file for the function you want to optimize. Suppose that you want to minimize the function

$$f(x) = \exp(-(x_1^2 + x_2^2))(x_1^2 - 2x_1x_2 + 6x_1 + 4x_2^2 - 3x_2).$$

The file that computes this function must accept a vector `x` of length 2, corresponding to the variables `x1` and `x2`, and return a scalar equal to the value of the function at `x`.

- 1 Select **New > Script (Ctrl+N)** from the MATLAB® **File** menu. A new file opens in the editor.
- 2 Enter the following two lines of code:

```
function z = my_fun(x)
z = x(1)^2 - 2*x(1)*x(2) + 6*x(1) + 4*x(2)^2 - 3*x(2);
```

- 3 Save the file in a folder on the MATLAB path.

Check that the file returns the correct value.

```
my_fun([2 3])
```

```
ans =
    31
```

For `gamultiobj`, suppose you have three objectives. Your objective function returns a three-element vector consisting of the three objective function values:

```
function z = my_fun(x)
z = zeros(1,3); % allocate output
z(1) = x(1)^2 - 2*x(1)*x(2) + 6*x(1) + 4*x(2)^2 - 3*x(2);
z(2) = x(1)*x(2) + cos(3*x(2)/(2+x(1)));
z(3) = tanh(x(1) + x(2));
```

Write a Vectorized Function

The `ga`, `gamultiobj`, `paretosearch`, `particleswarm`, and `patternsearch` solvers optionally compute the objective functions of a collection of vectors in one function call. This method can take less time than computing the objective functions of the vectors serially. This method is called a vectorized function call.

To compute in vectorized fashion:

- Write your objective function to:
 - Accept a matrix with an arbitrary number of rows.
 - Return the vector of function values of each row.
 - For `gamultiobj` or `paretosearch`, return a matrix, where each row contains the objective function values of the corresponding input matrix row.
- If you have a nonlinear constraint, be sure to write the constraint in a vectorized fashion. For details, see “Vectorized Constraints” on page 2-7.
- Set the `UseVectorized` option to `true` using `optimoptions`. For `patternsearch` or `paretosearch`, also set `UseCompletePoll` to `true`. Be sure to pass the options to the solver.

For example, to write the objective function of “Write a Function File” on page 2-2 in a vectorized fashion,

```
function z = my_fun(x)
z = x(:,1).^2 - 2*x(:,1).*x(:,2) + 6*x(:,1) + ...
    4*x(:,2).^2 - 3*x(:,2);
```

To use `my_fun` as a vectorized objective function for `patternsearch`:

```
options = optimoptions('patternsearch','UseCompletePoll',true,'UseVectorized',true);
[x fval] = patternsearch(@my_fun,[1 1],[[],[],[],[],[],[],[],...
    [],options);
```

To use `my_fun` as a vectorized objective function for `ga`:

```
options = optimoptions('ga','UseVectorized',true);
[x fval] = ga(@my_fun,2,[],[],[],[],[],[],[],options);
```

For `gamultiobj` or `paretosearch`,

```
function z = my_fun(x)
z = zeros(size(x,1),3); % allocate output
z(:,1) = x(:,1).^2 - 2*x(:,1).*x(:,2) + 6*x(:,1) + ...
    4*x(:,2).^2 - 3*x(:,2);
z(:,2) = x(:,1).*x(:,2) + cos(3*x(:,2)./(2+x(:,1)));
z(:,3) = tanh(x(:,1) + x(:,2));
```

To use `my_fun` as a vectorized objective function for `gamultiobj`:

```
options = optimoptions('ga','UseVectorized',true);
[x fval] = gamultiobj(@my_fun,2,[],[],[],[],[],[],options);
```

For more information on writing vectorized functions for `patternsearch`, see “Vectorize the Objective and Constraint Functions” on page 6-83. For more information on writing vectorized functions for `ga`, see “Vectorize the Fitness Function” on page 8-103.

Gradients and Hessians

If you use `GlobalSearch` or `MultiStart`, your objective function can return derivatives (gradient, Jacobian, or Hessian). For details on how to include this syntax in your objective function, see “Including Gradients and Hessians”. Use `optimoptions` to set options so that your solver uses the derivative information:

Local Solver = `fmincon`, `fminunc`

Condition	Option Setting
Objective function contains gradient	'SpecifyObjectiveGradient' = true; see “How to Include Gradients”
Objective function contains Hessian	'HessianFcn' = 'objective' or a function handle; see “Including Hessians”
Constraint function contains gradient	'SpecifyConstraintGradient' = true; see “Including Gradients in Constraint Functions”

Local Solver = `lsqcurvefit`, `lsqnonlin`

Condition	Option Setting
Objective function contains Jacobian	'SpecifyObjectiveGradient' = true

See Also

Related Examples

- “Vectorize the Objective and Constraint Functions” on page 6-83
- “Vectorize the Fitness Function” on page 8-103
- “Maximizing vs. Minimizing” on page 2-5

Maximizing vs. Minimizing

Global Optimization Toolbox optimization functions minimize the objective (or fitness) function. That is, they solve problems of the form

$$\min_x f(x).$$

If you want to maximize $f(x)$, minimize $-f(x)$, because the point at which the minimum of $-f(x)$ occurs is the same as the point at which the maximum of $f(x)$ occurs.

For example, suppose you want to maximize the function

$$f(x) = \exp(-(x_1^2 + x_2^2))(x_1^2 - 2x_1x_2 + 6x_1 + 4x_2^2 - 3x_2).$$

Write a function to compute

$$g(x) = -f(x) = -\exp(-(x_1^2 + x_2^2))(x_1^2 - 2x_1x_2 + 6x_1 + 4x_2^2 - 3x_2),$$

and then minimize $g(x)$. Start from the point $x_0 = [0 \ 0]$.

```
f = @(x)exp(-(x(1)^2 + x(2)^2))*(x(1)^2 - 2*x(1)*x(2) + 6*x(1) + 4*x(2)^2 - 3*x(2));
g = @(x)-f(x);
x0 = [0 0];
[xmin,gmin] = fminsearch(g,x0)
```

```
xmin =
```

```
0.5550 -0.5919
```

```
gmin =
```

```
-3.8683
```

The maximum of f is the value of $f(x_{\min})$, which is $-g_{\min}$.

```
f(xmin)
```

```
ans =
```

```
3.8683
```

See Also

Related Examples

- “Compute Objective Functions” on page 2-2

Write Constraints

In this section...
“Consult Optimization Toolbox Documentation” on page 2-6
“Set Bounds” on page 2-6
“Ensure ga Options Maintain Feasibility” on page 2-6
“Gradients and Hessians” on page 2-7
“Vectorized Constraints” on page 2-7

Consult Optimization Toolbox Documentation

Many Global Optimization Toolbox functions accept bounds, linear constraints, or nonlinear constraints. To see how to include these constraints in your problem, see “Write Constraints”. Try consulting these pertinent links to sections:

- “Bound Constraints”
- “Linear Constraints”
- “Nonlinear Constraints”

Note The `surrogateopt` solver uses a different syntax for nonlinear constraints than other solvers, and requires finite bounds on all components. For details, see the function reference page and “Convert Nonlinear Constraints Between `surrogateopt` Form and Other Solver Forms” on page 11-74.

Set Bounds

It is more important to set bounds for global solvers than for local solvers. Global solvers use bounds in a variety of ways:

- `GlobalSearch` requires bounds for its scatter-search point generation. If you do not provide bounds, `GlobalSearch` bounds each component below by `-9999` and above by `10001`. However, these bounds can easily be inappropriate.
- If you do not provide bounds and do not provide custom start points, `MultiStart` bounds each component below by `-1000` and above by `1000`. However, these bounds can easily be inappropriate.
- `ga` uses bounds and linear constraints for its initial population generation. For unbounded problems, `ga` uses a default of `0` as the lower bound and `1` as the upper bound for each dimension for initial point generation. For bounded problems, and problems with linear constraints, `ga` uses the bounds and constraints to make the initial population.
- `simulannealbnd` and `patternsearch` do not require bounds, although they can use bounds.

Ensure ga Options Maintain Feasibility

The `ga` solver generally maintains strict feasibility with respect to bounds and linear constraints. This means that, at every iteration, all members of a population satisfy the bounds and linear constraints.

However, you can set options that cause this feasibility to fail. For example if you set `MutationFcn` to `@mutationgaussian` or `@mutationuniform`, the mutation function does not respect constraints, and your population can become infeasible. Similarly, some crossover functions can cause infeasible populations, although the default `gacreationlinearfeasible` does respect bounds and linear constraints. Also, `ga` can have infeasible points when using custom mutation or crossover functions.

To ensure feasibility, use the default crossover and mutation functions for `ga`. Be especially careful that any custom functions maintain feasibility with respect to bounds and linear constraints.

Note When a problem has integer constraints, `ga` ensures that all operators (mutation, crossover, and creation) return feasible populations with respect to bounds, linear constraints, and integer constraints at each iteration. This feasibility holds to within a small tolerance.

Gradients and Hessians

If you use `GlobalSearch` or `MultiStart` with `fmincon`, your nonlinear constraint functions can return derivatives (gradient or Hessian). For details, see “Gradients and Hessians” on page 2-4.

Vectorized Constraints

The `ga` and `patternsearch` solvers optionally compute the nonlinear constraint functions of a collection of vectors in one function call. This method can take less time than computing the objective functions of the vectors serially. This method is called a vectorized function call.

For the solver to compute in a vectorized manner, you must vectorize both your objective (fitness) function and nonlinear constraint function. For details, see “Vectorize the Objective and Constraint Functions” on page 6-83.

As an example, suppose your nonlinear constraints for a three-dimensional problem are

$$\frac{x_1^2}{4} + \frac{x_2^2}{9} + \frac{x_3^2}{25} \leq 6$$

$$x_3 \geq \cosh(x_1 + x_2)$$

$$x_1 x_2 x_3 = 2.$$

The following code gives these nonlinear constraints in a vectorized fashion, assuming that the rows of your input matrix `x` are your population or input vectors:

```
function [c ceq] = nlinconst(x)

c(:,1) = x(:,1).^2/4 + x(:,2).^2/9 + x(:,3).^2/25 - 6;
c(:,2) = cosh(x(:,1) + x(:,2)) - x(:,3);
ceq = x(:,1).*x(:,2).*x(:,3) - 2;
```

For example, minimize the vectorized quadratic function

```
function y = vfun(x)
y = -x(:,1).^2 - x(:,2).^2 - x(:,3).^2;
```

over the region with constraints `nlinconst` using `patternsearch`:

```
options = optimoptions('patternsearch','UseCompletePoll',true,'UseVectorized',true);
[x fval] = patternsearch(@vfun,[1,1,2],[[],[],[],[],[],[],[],...)
```

```
@nlinconst,options)
Optimization terminated: mesh size less than options.MeshTolerance
and constraint violation is less than options.ConstraintTolerance.
```

```
x =
    0.2191    0.7500   12.1712
```

```
fval =
   -148.7480
```

Using ga:

```
options = optimoptions('ga','UseVectorized',true);
[x fval] = ga(@vfun,3,[],[],[],[],[],[],[],@nlinconst,options)
Optimization terminated: maximum number of generations exceeded.
```

```
x =
   -1.4098   -0.1216   11.6664
```

```
fval =
   -138.1066
```

For this problem `patternsearch` computes the solution far more quickly and accurately.

See Also

More About

- “Write Constraints”
- “Vectorize the Objective and Constraint Functions” on page 6-83

Set and Change Options

For all Global Optimization Toolbox solvers except `GlobalSearch` and `MultiStart`, the recommended way to set options is to use the `optimoptions` function. Set `GlobalSearch` and `MultiStart` options using their name-value pairs; see “Changing Global Options” on page 4-52.

For example, to set the `ga` maximum time to 300 seconds and set iterative display:

```
options = optimoptions('ga','MaxTime',300,'Display','iter');
```

Change options as follows:

- Dot notation. For example,
`options.MaxTime = 5e3;`
- `optimoptions`. For example,
`options = optimoptions(options,'MaxTime',5e3);`

Ensure that you pass `options` in your solver call. For example,

```
[x,fval] = ga(@objfun,2,[],[],[],[],lb,ub,@nonlcon,options);
```

To see the options you can change, consult the solver function reference pages. For option details, see the options reference sections.

See Also

[patternsearch](#) | [particleswarm](#) | [ga](#) | [simulannealbnd](#) | [surrogateopt](#) | [paretosearch](#) | [gamultiobj](#)

More About

- “Genetic Algorithm Options” on page 17-23
- “Particle Swarm Options” on page 17-45
- “Pattern Search Options” on page 17-7
- “Simulated Annealing Options” on page 17-58
- “Surrogate Optimization Options” on page 17-51

View Options

optimoptions “hides” some options, meaning it does not display their values. For example, it hides the patternsearch MaxMeshSize option.

```
options = optimoptions('patternsearch','MaxMeshSize',1e2)
```

```
options =
```

```
patternsearch options:
```

```
Set properties:
```

```
No options set.
```

```
Default properties:
```

```
AccelerateMesh: 0
ConstraintTolerance: 1.0000e-06
Display: 'final'
FunctionTolerance: 1.0000e-06
InitialMeshSize: 1
MaxFunctionEvaluations: '2000*numberOfVariables'
MaxIterations: '100*numberOfVariables'
MaxTime: Inf
MeshContractionFactor: 0.5000
MeshExpansionFactor: 2
MeshTolerance: 1.0000e-06
OutputFcn: []
PlotFcn: []
PollMethod: 'GPSPositiveBasis2N'
PollOrderAlgorithm: 'consecutive'
ScaleMesh: 1
SearchFcn: []
StepTolerance: 1.0000e-06
UseCompletePoll: 0
UseCompleteSearch: 0
UseParallel: 0
UseVectorized: 0
```

You can view the value of any option, including “hidden” options, by using dot notation. For example,

```
options.MaxMeshSize
```

```
ans =
```

```
100
```

Solver reference pages list “hidden” options in italics.

There are two reason that some options are “hidden”:

- They are not useful. For example, the `ga StallTest` option allows you to choose a stall test that does not work well. Therefore, this option is “hidden”.
- They are rarely used, or it is hard to know when to use them. For example, the `patternsearch MaxMeshSize` option is hard to choose, and so is “hidden”.

For details, see “Options that optimoptions Hides” on page 17-65.

See Also

More About

- “Set and Change Options” on page 2-9

Problem-Based Global Optimization

- “Decide Between Problem-Based and Solver-Based Approach” on page 3-2
- “Global Optimization Toolbox Default Solvers and Problem Types” on page 3-4
- “Initial Points for Global Optimization Toolbox Solvers” on page 3-6
- “Integer Constraints in Nonlinear Problem-Based Optimization” on page 3-8
- “Set Problem-Based Optimization Options for Global Optimization Toolbox Solvers” on page 3-9

Decide Between Problem-Based and Solver-Based Approach

Use a Global Optimization Toolbox solver to optimize a nonsmooth function, search for a global solution, or solve a multiobjective problem. Use the problem-based approach for a simpler way to set up and solve problems.

Problem-Based Characteristics

Advantages	Limitations
Easier to set up and debug	No equation problems
Easier to run different solvers on the same problem	No custom data types
Obtain an appropriate solver automatically	No checkpoint file for <code>surrogateopt</code>
Automatically speed the solution of problems where the objective and nonlinear constraints are calculated in the same time-consuming function (typically simulations)	No vectorization You must convert variables for options that relate to the solver-based approach

Advantages:

- **Easier to set up and debug.** In the problem-based approach, you use symbolic-style variables to create optimization expressions and constraints. See “Problem-Based Global Optimization Setup”. In the solver-based approach, you must place all variables into a single vector, which can be awkward, especially with variables of large or differing dimensions.
- **Easier to run different solvers on the same problem.** Some solvers have different calling syntaxes. For example, the syntax for nonlinear constraints in `surrogateopt` is different from the syntax in all other solvers. To run a problem using both `surrogateopt` and another solver in the solver-based approach, you have to create different versions of the objective function. In contrast, the problem-based approach takes care of translating syntaxes, so you need to change only the solver name and possibly some options.
- **Obtain an appropriate solver automatically.** The `solve` function automatically chooses a solver that can handle your objective and constraints. In the solver-based approach you must choose an appropriate solver.
- **Automatically speed the solution of problems where the objective and nonlinear constraints are calculated in the same time-consuming function (typically simulations).** Frequently, a simulation or ODE solver calculates the objective and nonlinear constraints in the same function. When you convert the time-consuming function to an optimization expression using `fcn2optimexpr`, you can save solution time by setting the `'ReuseEvaluation'` argument to `true`. This setting causes the solver to avoid recalculating the time-consuming function when evaluating the objective and nonlinear constraints. Achieving this time savings in the solver-based approach can require extra programming, as shown in the example “Objective and Nonlinear Constraints in the Same Function”.

Limitations:

- **No equation problems.** You cannot use a Global Optimization Toolbox solver to solve an equation problem of type `EquationProblem`. However, you can solve a feasibility problem by specifying a zero objective function and any constraints accepted by the solver. For an example, see “Solve Feasibility Problem Using `surrogateopt`, Problem-Based” on page 12-6.
- **No custom data types.** To use a custom data type with `ga` or `simulannealbnd`, you must use the solver-based approach. For examples, see “Custom Data Type Optimization Using the Genetic

Algorithm” on page 8-109 and “Multiprocessor Scheduling Using Simulated Annealing with a Custom Data Type” on page 13-28.

- **No checkpoint file for surrogateopt.** Use the solver-based approach for checkpoint files in surrogateopt. For details, see “Work with Checkpoint Files” on page 11-56.
- **No vectorization** (see “Using Vectorization”). If your objective function and any nonlinear constraint functions are written in a vectorized fashion, you must use the solver-based workflow to gain the benefits of vectorization. If you set the UseVectorized option in the problem-based approach, you get a warning, not improved performance.
- **You must convert variables for options that relate to the solver-based approach.** For example, custom output functions use solver-based syntax. Use varindex to convert problem-based variables to solver-based indices. For an example, see “Set Options in Problem-Based Approach Using varindex” on page 9-17.

See Also

solve

Related Examples

- “Problem-Based Global Optimization Setup”
- “Compare Several Global Solvers, Problem-Based” on page 1-3
- “Direct Search”
- “Genetic Algorithm”
- “Surrogate Optimization”

Global Optimization Toolbox Default Solvers and Problem Types

This topic identifies the types of problems handled by Global Optimization Toolbox solvers, and the default solver selected by `solve` or `prob2struct` for each type.

Problem Type	Default Solver
Linear Programming (LP)	<code>linprog</code>
Mixed-Integer Linear Programming (MILP)	<code>intlinprog</code>
Quadratic Programming (QP)	<code>quadprog</code>
Second-Order Cone Programming (SOCP)	<code>coneprog</code>
Linear Least Squares	<code>lsqlin</code>
Nonlinear Least Squares	<code>lsqnonlin</code>
Nonlinear Programming (NLP)	<code>fminunc</code> for problems with no constraints, otherwise <code>fmincon</code>
Mixed-Integer Nonlinear Programming (MINLP)	<code>ga</code>
Multiobjective	<code>gamultiobj</code>

Note The call `optimoptions(prob)` creates options for the default solver of the problem type of `prob`.

In this table, a check mark ✓ means the solver is available for the problem type, and an **x** means the solver is not available.

Problem Type	LP	MILP	QP	SOCP	Linear Least Squares	Nonlinear Least Squares	NLP	MINLP
Solver								
<code>linprog</code>	✓	x	x	x	x	x	x	x
<code>intlinprog</code>	✓	✓	x	x	x	x	x	x
<code>quadprog</code>	✓	x	✓	✓	✓	x	x	x
<code>coneprog</code>	✓	x	x	✓	x	x	x	x
<code>lsqlin</code>	x	x	x	x	✓	x	x	x
<code>lsqnonneg</code>	x	x	x	x	✓	x	x	x
<code>lsqnonlin</code>	x	x	x	x	✓	✓	x	x
<code>fminunc</code>	✓	x	✓	x	✓	✓	✓	x

Problem Type	LP	MILP	QP	SOCP	Linear Least Squares	Nonlinear Least Squares	NLP	MINLP
fmincon	✓	x	✓	✓	✓	✓	✓	x
fminbnd	x	x	x	x	✓	✓	✓	x
fminsearch	x	x	x	x	✓	✓	✓	x
patternsearch	✓	x	✓	✓	✓	✓	✓	x
ga	✓	✓	✓	✓	✓	✓	✓	✓
particleswarm	✓	x	✓	x	✓	✓	✓	x
simulannealbnd	✓	x	✓	x	✓	✓	✓	x
surrogateopt	✓	✓	✓	✓	✓	✓	✓	✓
gamultiobj	✓	✓	✓	✓	✓	✓	✓	✓
paretosearch	✓	x	✓	✓	✓	✓	✓	x

See Also

prob2struct | solve | optimoptions

Related Examples

- “Problem-Based Global Optimization Setup”

Initial Points for Global Optimization Toolbox Solvers

Some Global Optimization Toolbox solvers require an initial point `x0`: `patternsearch`, `simulannealbnd`, `GlobalSearch`, and `MultiStart`. When solving optimization problems using the problem-based approach, you specify `x0` in the second argument for `solve` and for `prob2struct`. To specify an initial point, create a structure with the variable names as fields and variable values as structure values. For example, for a scalar variable `x` and a 2-by-2 matrix `y` for the `patternsearch` solver, enter the following code.

```
x0.x = 5;
x0.y = eye(2) + 0.1*randn(2);
[sol,fval] = solve(prob,x0,"Solver","patternsearch")
```

You can also specify an initial point for these solvers using `optimvalues`, as shown next.

Other Global Optimization Toolbox solvers do not require an initial point, but can accept an initial point or set of initial points: `ga`, `gamultiobj`, `paretosearch`, and `surrogateopt`. To pass initial points to these solvers, create the points using `optimvalues`.

Note When using the problem-based approach, you cannot pass an initial point or initial population using options such as:

- `InitialPopulationMatrix` for `ga`
 - `InitialSwarmMatrix` for `particleswarm`
 - `InitialPoints` for `surrogateopt`
-

For example, take a 2-D variable `x` and a 2-by-2 matrix `y` for the `ga` solver.

```
x = optimvar('x',2,"LowerBound",-1,"UpperBound",1);
y = optimvar('y',2,2,"LowerBound",-1,"UpperBound",1);
prob = optimproblem("Objective",...
    cosh(dot(y*x,[2;-1])) - sinh(dot(y*x,[1;-2])));
prob.Constraints = y(1,2) == y(2,1);
% Set initial population: x0x for x, x0y for y
rng default
x0x = [1;1/2];
x0y = eye(2)/2 + 0.1*randn(2);
x0 = optimvalues(prob,'x',x0x,'y',x0y);
% Solve problem
[sol,fval] = solve(problem,Solver="ga")

Optimization terminated: average change in the fitness value less than options.FunctionTolerance.

sol =

    1.0000   -1.0000    0.3080   -1.0000   -0.9990    1.0000

fval =

   -50.4209
```

The solution satisfies the constraint $y(1,2) == y(2,1)$ only to within the constraint tolerance $1e-3$: `sol(4) = -1.0000`, but `sol(5) = -0.9990`.

See Also

`optimvalues` | `solve`

Related Examples

- “Problem-Based Global Optimization Setup”
- “Specify Start Points for MultiStart, Problem-Based” on page 5-3
- “Specify Starting Points and Values for `surrogateopt`, Problem-Based” on page 12-24
- “Pareto Front for Multiobjective Optimization, Problem-Based” on page 15-5

Integer Constraints in Nonlinear Problem-Based Optimization

To solve a nonlinear optimization problem with integer constraints using the problem-based approach, follow one of these processes:

- If you have a Global Optimization Toolbox license, formulate the problem as usual for the problem-based approach. `ga` is the default solver for a nonlinear problem with integer constraints. You can also specify `surrogateopt` as the solver in the `Solver` argument of `solve`.
- Use the solver-based approach with `ga` or `surrogateopt` as the solver. The solver-based approach requires you to modify the objective function and nonlinear constraint function when switching between these solvers.
- Convert the problem to a structure using `prob2struct`, and then use an external solver.
- Sometimes, you can iteratively approximate a nonlinear integer problem using `intlinprog`. For an example of this approach, see “Mixed-Integer Quadratic Programming Portfolio Optimization: Problem-Based”.

When you use an external solver and call `prob2struct`, you might need to specify the `Solver` name-value argument.

Note For a nonlinear problem with integer constraints, if you do not have a Global Optimization Toolbox license, you must include the `Solver` argument.

Even if you have a Global Optimization Toolbox license, you still might need to specify the `Solver` name-value argument. An external solver can expect the problem structure to be in a form that corresponds to a particular solver. For example, for a problem with linear and integer constraints and a quadratic objective function, an external solver might require the objective function to be expressed as matrices H and f in the expression $\frac{1}{2}x^T H x + f^T x$. To obtain these matrices, specify the 'quadprog' solver by using the `Solver` name-value argument.

```
problem = prob2struct(prob, "Solver", "quadprog");
```

If you do not specify the `quadprog` solver, the resulting problem structure can contain a function handle for the objective function rather than matrices. In either case, the resulting problem structure contains the integer variables in the `intcon` field.

Note For a nonlinear problem with integer constraints, when you specify a solver that does not handle integer constraints, `prob2struct` issues a warning that the solver cannot solve the resulting structure. If you then try to solve the problem by calling the solver on the problem structure, the solver ignores the integer constraints. In this case, the solution is not the solution to the original problem, but is instead the solution to the problem without integer constraints.

See Also

`prob2struct` | `solve` | `ga` | `surrogateopt`

Set Problem-Based Optimization Options for Global Optimization Toolbox Solvers

To tune your optimization solution process in the problem-based approach, set options using `optimoptions` and pass the options to `solve`. Set the solver for `solve` to match the options.

```
options = optimoptions("patternsearch",PlotFcn="psplotbestf");
[sol,fval] = solve(prob,x0,Options=options,Solver="patternsearch");
```

To find all available solvers for a problem, view the second output of `solvers`.

```
[~,validsolvers] = solvers(prob)

validsolvers = 1x10 string
    "lsqnonlin"    "lsqcurvefit"    "fmincon"    "ga"    "patternsearch"    "surrogateopt"    "pa
```

Some Global Optimization Toolbox solver options require you to map the option values to their solver equivalents. For example, to set an initial population for `ga` or `particleswarm`, you need to map all problem variables to a single matrix. Perform this mapping using the `varindex` function. For a discussion and examples, see “Set Options in Problem-Based Approach Using `varindex`” on page 9-17.

The remaining considerations for setting options are largely the same as in the solver-based approach. For details, see “Set Optimization Options”.

See Also

`optimoptions` | `solve` | `solvers` | `varindex`

Related Examples

- “Set Optimization Options”

Using GlobalSearch and MultiStart

- “Problems That GlobalSearch and MultiStart Can Solve” on page 4-2
- “Workflow for GlobalSearch and MultiStart” on page 4-3
- “Create Problem Structure” on page 4-4
- “Create Solver Object” on page 4-7
- “Set Start Points for MultiStart” on page 4-10
- “Run the Solver” on page 4-13
- “Single Solution” on page 4-16
- “Multiple Solutions” on page 4-17
- “Iterative Display” on page 4-21
- “Global Output Structures” on page 4-23
- “Visualize the Basins of Attraction” on page 4-24
- “Output Functions for GlobalSearch and MultiStart” on page 4-27
- “Plot Functions for GlobalSearch and MultiStart” on page 4-30
- “How GlobalSearch and MultiStart Work” on page 4-34
- “Can You Certify That a Solution Is Global?” on page 4-41
- “Refine Start Points” on page 4-44
- “Change Options” on page 4-51
- “Reproduce Results” on page 4-54
- “Find Global or Multiple Local Minima” on page 4-57
- “Maximizing Monochromatic Polarized Light Interference Patterns Using GlobalSearch and MultiStart” on page 4-64
- “Optimize Using Only Feasible Start Points” on page 4-76
- “MultiStart Using lsqcurvefit or lsqnonlin” on page 4-79
- “Parallel MultiStart” on page 4-83
- “Isolated Global Minimum” on page 4-86

Problems That GlobalSearch and MultiStart Can Solve

The `GlobalSearch` and `MultiStart` solvers apply to problems with smooth objective and constraint functions. The solvers search for a global minimum, or for a set of local minima. For more information on which solver to use, see “Table for Choosing a Solver” on page 1-29.

`GlobalSearch` and `MultiStart` work by starting a local solver, such as `fmincon`, from a variety of start points. Generally the start points are random. However, for `MultiStart` you can provide a set of start points. For more information, see “How GlobalSearch and MultiStart Work” on page 4-34.

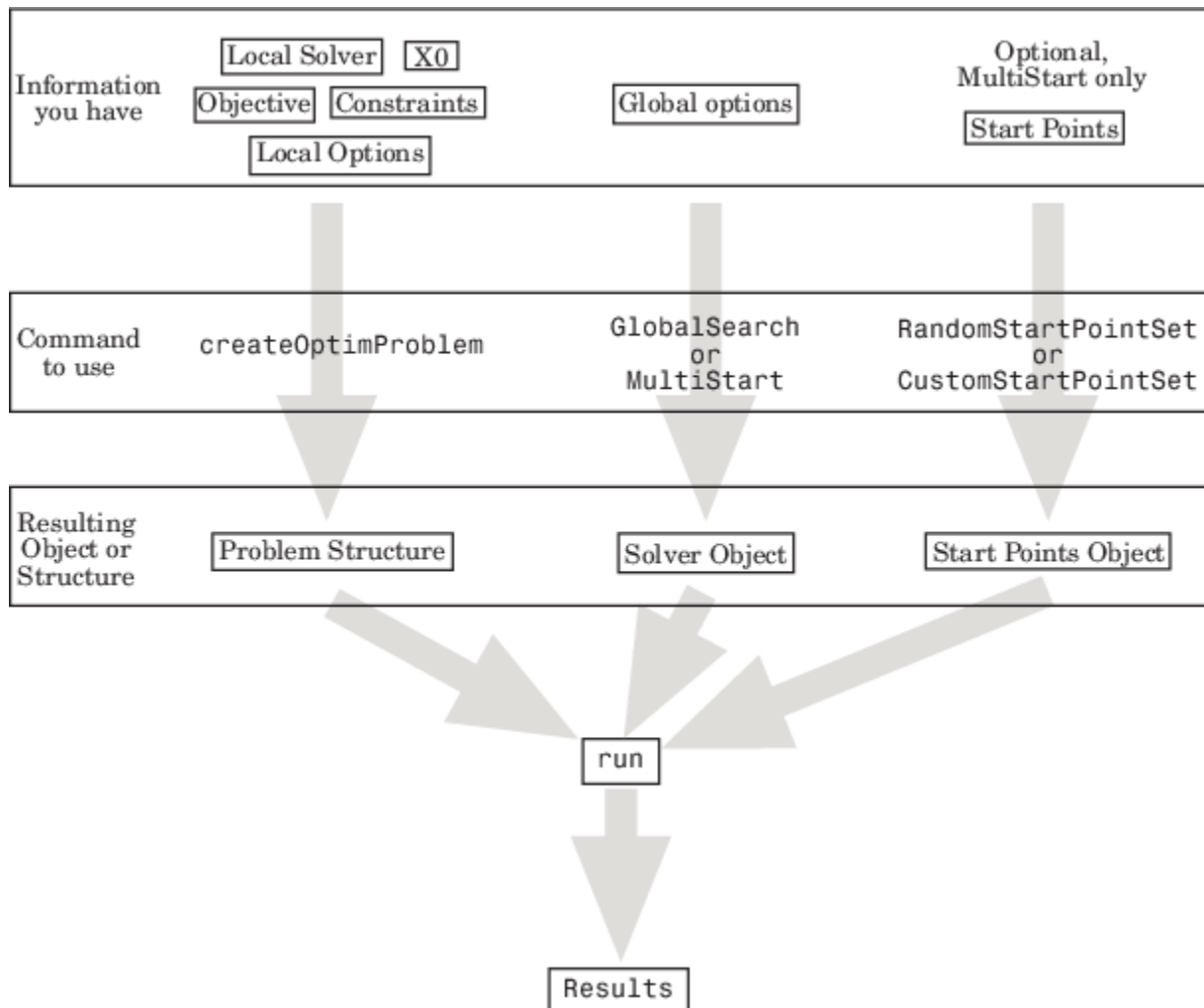
To find out how to use these solvers, see “Workflow for GlobalSearch and MultiStart” on page 4-3.

Workflow for GlobalSearch and MultiStart

To find a global or multiple local solutions for a smooth problem:

- 1 “Create Problem Structure” on page 4-4
- 2 “Create Solver Object” on page 4-7
- 3 (Optional, MultiStart only) “Set Start Points for MultiStart” on page 4-10
- 4 “Run the Solver” on page 4-13

The following figure illustrates these steps.



See Also

Related Examples

- “Global or Multiple Starting Point Search”

Create Problem Structure

In this section...

“About Problem Structures” on page 4-4

“Use the createOptimProblem Function” on page 4-4

“Example: Create a Problem Structure with createOptimProblem” on page 4-5

About Problem Structures

To use the `GlobalSearch` or `MultiStart` solvers, you must first create a problem structure. The recommended way to create a problem structure is using the `createOptimProblem` function on page 4-4. You can create a structure manually, but doing so is error-prone.

Use the createOptimProblem Function

Follow these steps to create a problem structure using the `createOptimProblem` function.

- 1 Define your objective function as a file or anonymous function. For details, see “Compute Objective Functions” on page 2-2. If your solver is `lsqcurvefit` or `lsqnonlin`, ensure the objective function returns a vector, not scalar.
- 2 If relevant, create your constraints, such as bounds and nonlinear constraint functions. For details, see “Write Constraints” on page 2-6.
- 3 Create a start point. For example, to create a three-dimensional random start point `xstart`:

```
xstart = randn(3,1);
```

- 4 (Optional) Create options using `optimoptions`. For example,

```
options = optimoptions(@fmincon,'Algorithm','interior-point');
```

- 5 Enter

```
problem = createOptimProblem(solver,
```

where `solver` is the name of your local solver:

- For `GlobalSearch`: 'fmincon'
- For `MultiStart` the choices are:
 - 'fmincon'
 - 'fminunc'
 - 'lsqcurvefit'
 - 'lsqnonlin'

For help choosing, see “Optimization Decision Table”.

- 6 Set an initial point using the '`x0`' parameter. If your initial point is `xstart`, and your solver is `fmincon`, your entry is now

```
problem = createOptimProblem('fmincon','x0',xstart,
```

- 7 Include the function handle for your objective function in `objective`:

```
problem = createOptimProblem('fmincon','x0',xstart, ...
    'objective',@objfun,
```

- 8 Set bounds and other constraints as applicable.

Constraint	Name
lower bounds	'lb'
upper bounds	'ub'
matrix Aineq for linear inequalities $A_{ineq} x \leq b_{ineq}$	'Aineq'
vector bineq for linear inequalities $A_{ineq} x \leq b_{ineq}$	'bineq'
matrix Aeq for linear equalities $A_{eq} x = b_{eq}$	'Aeq'
vector beq for linear equalities $A_{eq} x = b_{eq}$	'beq'
nonlinear constraint function	'nonlcon'

- 9 If using the `lsqcurvefit` local solver, include vectors of input data and response data, named 'xdata' and 'ydata' respectively.
- 10 *Best practice: validate the problem structure by running your solver on the structure.* For example, if your local solver is `fmincon`:

```
[x,fval,exitflag,output] = fmincon(problem);
```

Example: Create a Problem Structure with createOptimProblem

This example minimizes the function from “Run the Solver” on page 4-13, subject to the constraint $x_1 + 2x_2 \geq 4$. The objective is

$$\text{sixmin} = 4x^2 - 2.1x^4 + x^6/3 + xy - 4y^2 + 4y^4. \quad (4-1)$$

Use the interior-point algorithm of `fmincon`, and set the start point to [2;3].

- 1 Write a function handle for the objective function.

```
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
```

- 2 Write the linear constraint matrices. Change the constraint to “less than” form:

```
A = [-1,-2];
b = -4;
```

- 3 Create the local options to use the interior-point algorithm:

```
opts = optimoptions(@fmincon,'Algorithm','interior-point');
```

- 4 Create the problem structure with `createOptimProblem`:

```
problem = createOptimProblem('fmincon', ...
    'x0',[2;3],'objective',sixmin, ...
    'Aineq',A,'bineq',b,'options',opts)
```

- 5 The resulting structure:

```
problem =
```

```
struct with fields:
```

```
objective: @(x)(4*x(1)^2-2.1*x(1)^4+x(1)^6/3+x(1)*x(2)-4*x(2)^2+4*x(2)^4)
x0: [2x1 double]
```

```
Aineq: [-1 -2]
bineq: -4
Aeq: []
beq: []
lb: []
ub: []
nonlcon: []
solver: 'fmincon'
options: [1x1 optim.options.Fmincon]
```

- 6 *Best practice: validate the problem structure by running your solver on the structure:*

```
[x,fval,exitflag,output] = fmincon(problem);
```

See Also

Related Examples

- “Workflow for GlobalSearch and MultiStart” on page 4-3

Create Solver Object

In this section...
“What Is a Solver Object?” on page 4-7
“Properties (Global Options) of Solver Objects” on page 4-7
“Creating a Nondefault GlobalSearch Object” on page 4-8
“Creating a Nondefault MultiStart Object” on page 4-9

What Is a Solver Object?

A solver object contains your preferences for the global portion of the optimization.

You do not need to set any preferences. Create a `GlobalSearch` object named `gs` with default settings as follows:

```
gs = GlobalSearch;
```

Similarly, create a `MultiStart` object named `ms` with default settings as follows:

```
ms = MultiStart;
```

Properties (Global Options) of Solver Objects

Global options are properties of a `GlobalSearch` or `MultiStart` object.

Properties for both GlobalSearch and MultiStart

Property Name	Meaning
Display	Detail level of iterative display. Set to 'off' for no display, 'final' (default) for a report at the end of the run, or 'iter' for reports as the solver progresses. For more information and examples, see “Iterative Display” on page 4-21.
FunctionTolerance	Solvers consider objective function values within FunctionTolerance of each other to be identical (not distinct). Default: 1e-6. Solvers group solutions when the solutions satisfy both FunctionTolerance and XTolerance tolerances.
XTolerance	Solvers consider solutions within XTolerance distance of each other to be identical (not distinct). Default: 1e-6. Solvers group solutions when the solutions satisfy both FunctionTolerance and XTolerance tolerances.
MaxTime	Solvers halt if the run exceeds MaxTime seconds, as measured by a clock (not processor seconds). Default: Inf
StartPointsToRun	Choose whether to run 'all' (default) start points, only those points that satisfy 'bounds', or only those points that are feasible with respect to bounds and inequality constraints with 'bounds-ineqs'. For an example, see “Optimize Using Only Feasible Start Points” on page 4-76.
OutputFcn	Functions to run after each local solver run. See “Output Functions for GlobalSearch and MultiStart” on page 4-27. Default: []
PlotFcn	Plot functions to run after each local solver run. See “Plot Functions for GlobalSearch and MultiStart” on page 4-30. Default: []

Properties for GlobalSearch

Property Name	Meaning
NumTrialPoints	Number of trial points to examine. Default: 1000
BasinRadiusFactor	See GlobalSearch Properties for detailed descriptions of these properties.
DistanceThresholdFactor	
MaxWaitCycle	
NumStageOnePoints	
PenaltyThresholdFactor	

Properties for MultiStart

Property Name	Meaning
UseParallel	When true, MultiStart attempts to distribute start points to multiple processors for the local solver. Disable by setting to false (default). For details, see “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11. For an example, see “Parallel MultiStart” on page 4-83.

Creating a Nondefault GlobalSearch Object

Suppose you want to solve a problem and:

- Consider local solutions identical if they are within 0.01 of each other and the function values are within the default `FunctionTolerance` tolerance.
- Spend no more than 2000 seconds on the computation.

To solve the problem, create a `GlobalSearch` object `gs` as follows:

```
gs = GlobalSearch('XTolerance',0.01,'MaxTime',2000);
```

Creating a Nondefault MultiStart Object

Suppose you want to solve a problem such that:

- You consider local solutions identical if they are within 0.01 of each other and the function values are within the default `FunctionTolerance` tolerance.
- You spend no more than 2000 seconds on the computation.

To solve the problem, create a `MultiStart` object `ms` as follows:

```
ms = MultiStart('XTolerance',0.01,'MaxTime',2000);
```

See Also

Related Examples

- “Workflow for `GlobalSearch` and `MultiStart`” on page 4-3

Set Start Points for MultiStart

In this section...

“Four Ways to Set Start Points” on page 4-10
 “Positive Integer for Start Points” on page 4-10
 “RandomStartPointSet Object for Start Points” on page 4-10
 “CustomStartPointSet Object for Start Points” on page 4-11
 “Cell Array of Objects for Start Points” on page 4-12

Four Ways to Set Start Points

There are four ways you tell `MultiStart` which start points to use for the local solver:

- Pass a positive integer on page 4-10 `k`. `MultiStart` generates $k - 1$ start points as if using a `RandomStartPointSet` object and the problem structure. `MultiStart` also uses the `x0` start point from the problem structure, for a total of `k` start points.
- Pass a `RandomStartPointSet` object on page 4-10.
- Pass a `CustomStartPointSet` object on page 4-11.
- Pass a cell array on page 4-12 of `RandomStartPointSet` and `CustomStartPointSet` objects. Pass a cell array if you have some specific points you want to run, but also want `MultiStart` to use other random start points.

Note You can control whether `MultiStart` uses all start points, or only those points that satisfy bounds or other inequality constraints. For more information, see “Filter Start Points (Optional)” on page 4-39.

Positive Integer for Start Points

The syntax for running `MultiStart` for `k` start points is

```
[xmin,fmin,flag,outpt,allmins] = run(ms,problem,k);
```

The positive integer `k` specifies the number of start points `MultiStart` uses. `MultiStart` generates random start points using the dimension of the problem and bounds from the problem structure. `MultiStart` generates $k - 1$ random start points, and also uses the `x0` start point from the problem structure.

RandomStartPointSet Object for Start Points

Create a `RandomStartPointSet` object as follows:

```
stpoints = RandomStartPointSet;
```

Run `MultiStart` starting from a `RandomStartPointSet` as follows:

```
[xmin,fmin,flag,outpt,allmins] = run(ms,problem,stpoints);
```

By default a `RandomStartPointSet` object generates 10 start points. Control the number of start points with the `NumStartPoints` property. For example, to generate 40 start points:

```
stpoints = RandomStartPointSet('NumStartPoints',40);
```

You can set an `ArtificialBound` for a `RandomStartPointSet`. This `ArtificialBound` works in conjunction with the bounds from the problem structure:

- If a component has no bounds, `RandomStartPointSet` uses a lower bound of `-ArtificialBound`, and an upper bound of `ArtificialBound`.
- If a component has a lower bound `lb` but no upper bound, `RandomStartPointSet` uses an upper bound of `lb + 2*ArtificialBound`.
- Similarly, if a component has an upper bound `ub` but no lower bound, `RandomStartPointSet` uses a lower bound of `ub - 2*ArtificialBound`.

For example, to generate 100 start points with an `ArtificialBound` of 50:

```
stpoints = RandomStartPointSet('NumStartPoints',100, ...
    'ArtificialBound',50);
```

A `RandomStartPointSet` object generates start points with the same dimension as the `x0` point in the problem structure; see `list`.

CustomStartPointSet Object for Start Points

To use a specific set of starting points, package them in a `CustomStartPointSet` as follows:

- 1 Place the starting points in a matrix. Each row of the matrix represents one starting point. `MultiStart` runs all the rows of the matrix, subject to filtering with the `StartPointsToRun` property. For more information, see “MultiStart Algorithm” on page 4-38.
- 2 Create a `CustomStartPointSet` object from the matrix:

```
tpoints = CustomStartPointSet(ptmatrix);
```

For example, create a set of 40 five-dimensional points, with each component of a point equal to 10 plus an exponentially distributed variable with mean 25:

```
pts = -25*log(rand(40,5)) + 10;
tpoints = CustomStartPointSet(pts);
```

Run `MultiStart` starting from a `CustomStartPointSet` as follows:

```
[xmin,fmin,flag,outpt,allmins] = run(ms,problem,tpoints);
```

To get the original matrix of points from a `CustomStartPointSet` object, use `list`:

```
pts = list(tpoints); % Assumes tpoints is a CustomStartPointSet
```

A `CustomStartPointSet` has two properties: `StartPointsDimension` and `NumStartPoints`. You can use these properties to query a `CustomStartPointSet` object. For example, the `tpoints` object in the example has the following properties:

```
tpoints.StartPointsDimension
ans =
    5
```

```
tpoints.NumStartPoints
ans =
   40
```

Cell Array of Objects for Start Points

To use a specific set of starting points along with some randomly generated points, pass a cell array of `RandomStartPointSet` or `CustomStartPointSet` objects.

For example, to use both the 40 specific five-dimensional points of “CustomStartPointSet Object for Start Points” on page 4-11 and 40 additional five-dimensional points from `RandomStartPointSet`:

```
pts = -25*log(rand(40,5)) + 10;  
tpoints = CustomStartPointSet(pts);  
rpts = RandomStartPointSet('NumStartPoints',40);  
allpts = {tpoints,rpts};
```

Run `MultiStart` starting from the `allpts` cell array:

```
% Assume ms and problem exist  
[xmin,fmin,flag,outpt,allmins] = run(ms,problem,allpts);
```

See Also

Related Examples

- “Workflow for GlobalSearch and MultiStart” on page 4-3

Run the Solver

In this section...

“Optimize by Calling run” on page 4-13

“Example of Run with GlobalSearch” on page 4-13

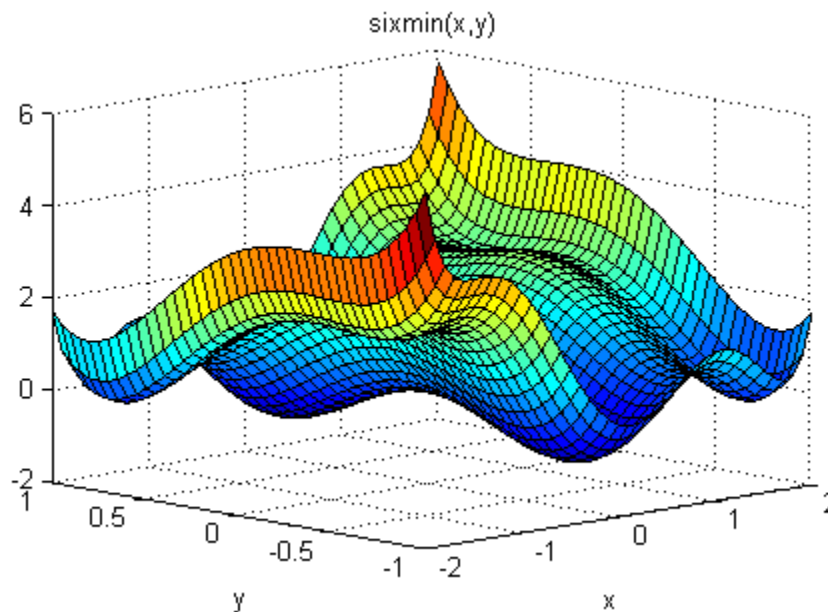
“Example of Run with MultiStart” on page 4-14

Optimize by Calling run

Running a solver is nearly identical for `GlobalSearch` and `MultiStart`. The only difference in syntax is `MultiStart` takes an additional input describing the start points.

For example, suppose you want to find several local minima of the `sixmin` function

$$\text{sixmin} = 4x^2 - 2.1x^4 + x^6/3 + xy - 4y^2 + 4y^4.$$



This function is also called the six-hump camel back function [3]. All the local minima lie in the region $-3 \leq x, y \leq 3$.

Example of Run with GlobalSearch

To find several local minima of the `sixmin` function using `GlobalSearch`, enter:

```
% % Set the random stream to get exactly the same output
% rng(14,'twister')
gs = GlobalSearch;
opts = optimoptions(@fmincon,'Algorithm','interior-point');
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3],...
```

```

    'options',opts);
[xming,fming,flagg,outptg,manyminsg] = run(gs,problem);

```

The output of the run (which varies, based on the random seed):

```

xming,fming,flagg,outptg,manyminsg
xming =
    0.0898    -0.7127

fming =
   -1.0316

flagg =
     1

outptg =

    struct with fields:

                funcCount: 2115
            localSolverTotal: 3
            localSolverSuccess: 3
        localSolverIncomplete: 0
            localSolverNoSolution: 0
                message: 'GlobalSearch stopped because it analyzed all the trial po...'

manyminsg =
    1x2 GlobalOptimSolution array with properties:

    X
    Fval
    Exitflag
    Output
    X0

```

Example of Run with MultiStart

To find several local minima of the `sixmin` function using 50 runs of `fmincon` with `MultiStart`, enter:

```

% % Set the random stream to get exactly the same output
% rng(14,'twister')
ms = MultiStart;
opts = optimoptions(@fmincon,'Algorithm','interior-point');
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3],...
    'options',opts);
[xminm,fminm,flagm,outptm,manyminsm] = run(ms,problem,50);

```

The output of the run (which varies based on the random seed):

```

xminm,fminm,flagm,outptm,manyminsm

xminm =
    0.0898    -0.7127

```



```
fminm =  
    -1.0316  
  
flagm =  
     1  
  
outptm =  
  
    struct with fields:  
  
        funcCount: 2034  
        localSolverTotal: 50  
        localSolverSuccess: 50  
        localSolverIncomplete: 0  
        localSolverNoSolution: 0  
        message: 'MultiStart completed the runs from all start points....'  
  
manyminsm =  
    1x6 GlobalOptimSolution array with properties:  
  
    X  
    Fval  
    Exitflag  
    Output  
    X0
```

In this case, `MultiStart` located all six local minima, while `GlobalSearch` located two. For pictures of the `MultiStart` solutions, see “Visualize the Basins of Attraction” on page 4-24.

See Also

Related Examples

- “Global or Multiple Starting Point Search”

Single Solution

You obtain the single best solution found during the run by calling `run` with the syntax

```
[x,fval,exitflag,output] = run(...);
```

- `x` is the location of the local minimum with smallest objective function value.
- `fval` is the objective function value evaluated at `x`.
- `exitflag` is an exit flag for the global solver. Values:

Global Solver Exit Flags

2	At least one feasible local minimum found. Some runs of the local solver did not converge.
1	At least one feasible local minimum found. All runs of the local solver converged (had positive exit flag).
0	No local minimum found. Local solver called at least once, and at least one local solver exceeded the <code>MaxIterations</code> or <code>MaxFunctionEvaluations</code> tolerances.
-1	One or more local solver runs stopped by the local solver output or plot function.
-2	No feasible local minimum found.
-5	<code>MaxTime</code> limit exceeded.
-8	No solution found. All runs had local solver exit flag -2 or lower, not all equal -2.
-10	Failures encountered in user-provided functions.

- `output` is a structure with details about the multiple runs of the local solver. For more information, see “Global Output Structures” on page 4-23.

The list of outputs is for the case `exitflag > 0`. If `exitflag <= 0`, then `x` is the following:

- If some local solutions are feasible, `x` represents the location of the lowest objective function value. “Feasible” means the constraint violations are smaller than `problem.options.ConstraintTolerance`.
- If no solutions are feasible, `x` is the solution with lowest infeasibility.
- If no solutions exist, `x`, `fval`, and `output` are empty entries (`[]`).

See Also

Related Examples

- “Run the Solver” on page 4-13

Multiple Solutions

In this section...

“About Multiple Solutions” on page 4-17

“Change the Definition of Distinct Solutions” on page 4-19

About Multiple Solutions

You obtain multiple solutions in an object by calling `run` with the syntax

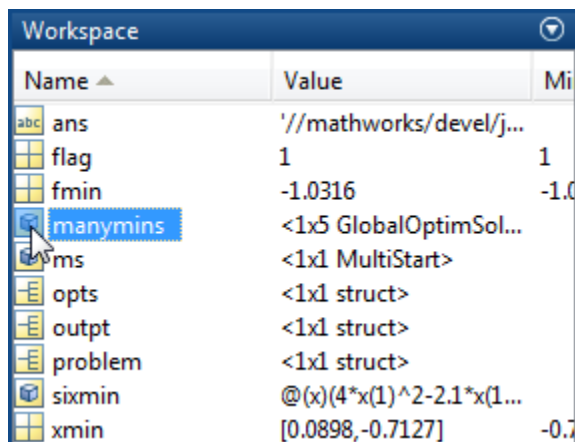
```
[x,fval,exitflag,output,manymins] = run(...);
```

`manymins` is a vector of solution objects; see `GlobalOptimSolution`. The `manymins` vector is in order of objective function value, from lowest (best) to highest (worst). Each solution object contains the following properties (fields):

- `X` — a local minimum
- `Fval` — the value of the objective function at `X`
- `Exitflag` — the exit flag for the local solver (described in the local solver function reference page: `fmincon exitflag`, `fminunc exitflag`, `lsqcurvefit exitflag`, or `lsqnonlin exitflag`)
- `Output` — an output structure for the local solver (described in the local solver function reference page: `fmincon output`, `fminunc output`, `lsqcurvefit output`, or `lsqnonlin output`)
- `X0` — a cell array of start points that led to the solution point `X`

There are several ways to examine the vector of solution objects:

- In the MATLAB Workspace Browser. Double-click the solution object, and then double-click the resulting display in the Variables editor.



Variables - manymins				
manymins ×				
manymins <1x5 GlobalOptimSolution>				
	1	2	3	
1	<1x1 Global...	<1x1 Global...	<1x1 Global...	<

Variables - manymins(1, 1)				
manymins × manymins(1, 1) ×				
manymins(1, 1) <1x1 GlobalOptimSolution>				
Property ^	Value	Min	Max	
X	[0.0898,-0.7127]	-0.7127	0.0898	
Fval	-1.0316	-1.0316	-1.0316	
Exitflag	1	1	1	
Output	<1x1 struct>			
X0	<1x19 cell>			

- Using dot notation. GlobalOptimSolution properties are capitalized. Use proper capitalization to access the properties.

For example, to find the vector of function values, enter:

```
fcnvals = [manymins.Fval]
```

```
fcnvals =
    -1.0316    -0.2155         0
```

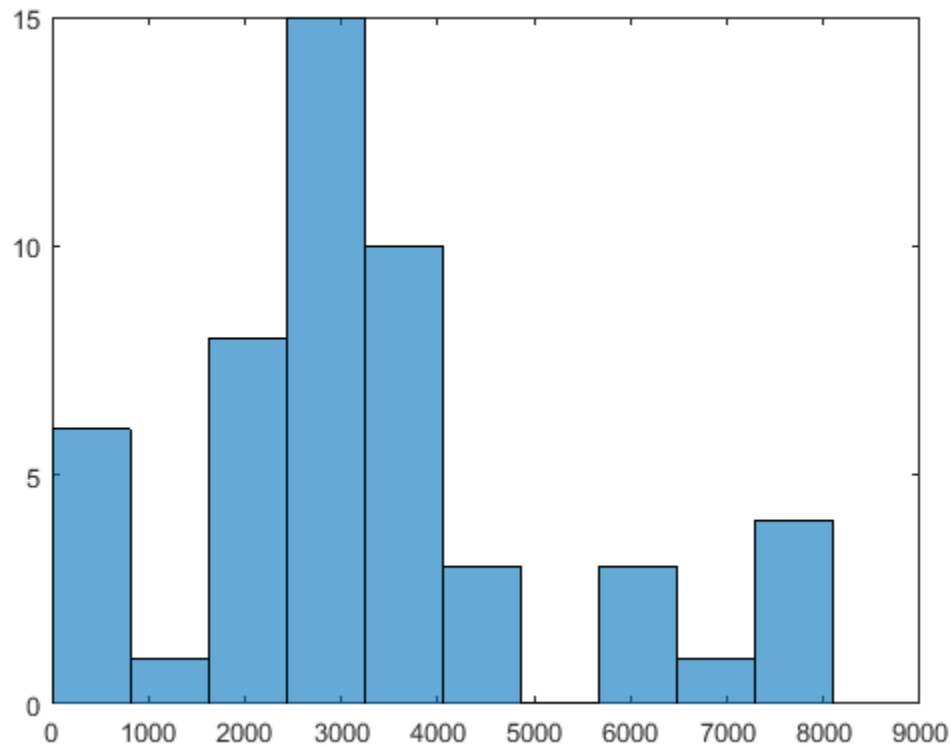
To get a cell array of all the start points that led to the lowest function value (the first element of manymins), enter:

```
smallX0 = manymins(1).X0
```

- Plot some field values. For example, to see the range of resulting Fval, enter:

```
histogram([manymins.Fval],10)
```

This results in a histogram of the computed function values. (The figure shows a histogram from a different example than the previous few figures.)



Change the Definition of Distinct Solutions

You might find out, after obtaining multiple local solutions, that your tolerances were not appropriate. You can have many more local solutions than you want, spaced too closely together. Or you can have fewer solutions than you want, with `GlobalSearch` or `MultiStart` clumping together too many solutions.

To deal with this situation, run the solver again with different tolerances. The `XTolerance` and `FunctionTolerance` tolerances determine how the solvers group their outputs into the `GlobalOptimSolution` vector. These tolerances are properties of the `GlobalSearch` or `MultiStart` object.

For example, suppose you want to use the active-set algorithm in `fmincon` to solve the problem in “Example of Run with `MultiStart`” on page 4-14. Further suppose that you want to have tolerances of 0.01 for both `XTolerance` and `FunctionTolerance`. The run method groups local solutions whose objective function values are within `FunctionTolerance` of each other, and which are also less than `XTolerance` apart from each other. To obtain the solution:

```
% % Set the random stream to get exactly the same output
% rng(14,'twister')
ms = MultiStart('FunctionTolerance',0.01,'XTolerance',0.01);
opts = optimoptions(@fmincon,'Algorithm','active-set');
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
```

```
'objective',sixmin,'lb',[-3,-3],'ub',[3,3],...  
'options',opts);  
[xminm,fminm,flagm,outptm,someminsm] = run(ms,problem,50);
```

MultiStart completed the runs from all start points.

All 50 local solver runs converged with a positive local solver exit flag.

```
someminsm
```

```
someminsm =
```

```
1x5 GlobalOptimSolution
```

```
Properties:
```

```
X  
Fval  
Exitflag  
Output  
X0
```

In this case, MultiStart generated five distinct solutions. Here “distinct” means that the solutions are more than 0.01 apart in either objective function value or location.

See Also

Related Examples

- “Run the Solver” on page 4-13
- “Visualize the Basins of Attraction” on page 4-24

Iterative Display

In this section...

“Types of Iterative Display” on page 4-21

“Examine Types of Iterative Display” on page 4-21

Types of Iterative Display

Iterative display gives you information about the progress of solvers during their runs.

There are two types of iterative display:

- Global solver display
- Local solver display

Both types appear at the command line, depending on global and local options.

Obtain local solver iterative display by setting the `Display` option in the `problem.options` field to `'iter'` or `'iter-detailed'` with `optimoptions`. For more information, see “Iterative Display”.

Obtain global solver iterative display by setting the `Display` property in the `GlobalSearch` or `MultiStart` object to `'iter'`.

Global solvers set the default `Display` option of the local solver to `'off'`, unless the problem structure has a value for this option. Global solvers do not override any setting you make for local options.

Note Setting the local solver `Display` option to anything other than `'off'` can produce a great deal of output. The default `Display` option created by `optimoptions(@solver)` is `'final'`.

Examine Types of Iterative Display

Run the example described in “Run the Solver” on page 4-13 using `GlobalSearch` with `GlobalSearch` iterative display:

```
%% Set the random stream to get exactly the same output
% rng(14,'twister')
gs = GlobalSearch('Display','iter');
opts = optimoptions(@fmincon,'Algorithm','interior-point');
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3],...
    'options',opts);
[xming,fming,flagg,outptg,manyminsg] = run(gs,problem);
```

Num Pts Analyzed	F-count	Best f(x)	Current Penalty	Threshold Penalty	Local f(x)	Local exitflag	Procedure
0	34	-1.032			-1.032	1	Initial Point
200	1275	-1.032			-0.2155	1	Stage 1 Local
300	1377	-1.032	248.7	-0.2137			Stage 2 Search
400	1477	-1.032	278	1.134			Stage 2 Search
446	1561	-1.032	1.6	2.073	-0.2155	1	Stage 2 Local
500	1615	-1.032	9.055	0.3214			Stage 2 Search
600	1715	-1.032	-0.7299	-0.7686			Stage 2 Search
700	1815	-1.032	0.3191	-0.7431			Stage 2 Search
800	1915	-1.032	296.4	0.4577			Stage 2 Search
900	2015	-1.032	10.68	0.5116			Stage 2 Search

```
1000    2115    -1.032    -0.9207    -0.9254
```

Stage 2 Search

GlobalSearch stopped because it analyzed all the trial points.

All 3 local solver runs converged with a positive local solver exit flag.

Run the same example without GlobalSearch iterative display, but with fmincon iterative display:

```
gs.Display = 'final';
problem.options.Display = 'iter';
[xming,fming,flagg,outptg,manyminsg] = run(gs,problem);
```

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of step
0	3	-1.980435e-02	0.000e+00	1.996e+00	
1	9	-6.970985e-02	0.000e+00	3.140e+00	2.533e-01
2	13	-8.662720e-02	0.000e+00	2.775e+00	1.229e-01
3	18	-1.176972e-01	0.000e+00	1.629e+00	1.811e-01
4	21	-2.132377e-01	0.000e+00	2.097e-01	8.636e-02
5	24	-2.153982e-01	0.000e+00	7.701e-02	1.504e-02
6	27	-2.154521e-01	0.000e+00	1.547e-02	1.734e-03
7	30	-2.154637e-01	0.000e+00	1.222e-03	1.039e-03
8	33	-2.154638e-01	0.000e+00	1.543e-04	8.413e-05
9	36	-2.154638e-01	0.000e+00	1.543e-06	6.610e-06
10	39	-2.154638e-01	0.000e+00	1.686e-07	7.751e-08

Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in feasible directions, to within the selected value of the function tolerance, and constraints were satisfied to within the selected value of the constraint tolerance.

<stopping criteria details>

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of step
0	3	-1.980435e-02	0.000e+00	1.996e+00	
... MANY ITERATIONS DELETED ...					
8	33	-1.031628e+00	0.000e+00	8.742e-07	2.287e-07

Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in feasible directions, to within the selected value of the function tolerance, and constraints were satisfied to within the selected value of the constraint tolerance.

<stopping criteria details>

GlobalSearch stopped because it analyzed all the trial points.

All 4 local solver runs converged with a positive local solver exit flag.

Setting GlobalSearch iterative display, as well as fmincon iterative display, yields both displays intermingled.

For an example of iterative display in a parallel environment, see “Parallel MultiStart” on page 4-83.

See Also

Related Examples

- “Global or Multiple Starting Point Search”

Global Output Structures

run can produce two types of output structures:

- A global output structure. This structure contains information about the overall run from multiple starting points. Details follow.
- Local solver output structures. The vector of `GlobalOptimSolution` objects contains one such structure in each element of the vector. For a description of this structure, see “Output Structures”, or the function reference pages for the local solvers: `fmincon` output, `fminunc` output, `lsqcurvefit` output , or `lsqnonlin` output .

Global Output Structure

Field	Meaning
<code>funcCount</code>	Total number of calls to user-supplied functions (objective or nonlinear constraint)
<code>localSolverTotal</code>	Number of local solver runs started
<code>localSolverSuccess</code>	Number of local solver runs that finished with a positive exit flag
<code>localSolverIncomplete</code>	Number of local solver runs that finished with a 0 exit flag
<code>localSolverNoSolution</code>	Number of local solver runs that finished with a negative exit flag
<code>message</code>	<code>GlobalSearch</code> or <code>MultiStart</code> exit message

A positive exit flag from a local solver generally indicates a successful run. A negative exit flag indicates a failure. A 0 exit flag indicates that the solver stopped by exceeding the iteration or function evaluation limit. For more information, see “Exit Flags and Exit Messages” or “Tolerances and Stopping Criteria”.

See Also

Related Examples

- “Global or Multiple Starting Point Search”

Visualize the Basins of Attraction

Which start points lead to which basin? For a steepest descent solver, nearby points generally lead to the same basin; see “Basins of Attraction” on page 1-25. However, for Optimization Toolbox solvers, basins are more complicated.

Plot the MultiStart start points from the example, “Example of Run with MultiStart” on page 4-14, color-coded with the basin where they end.

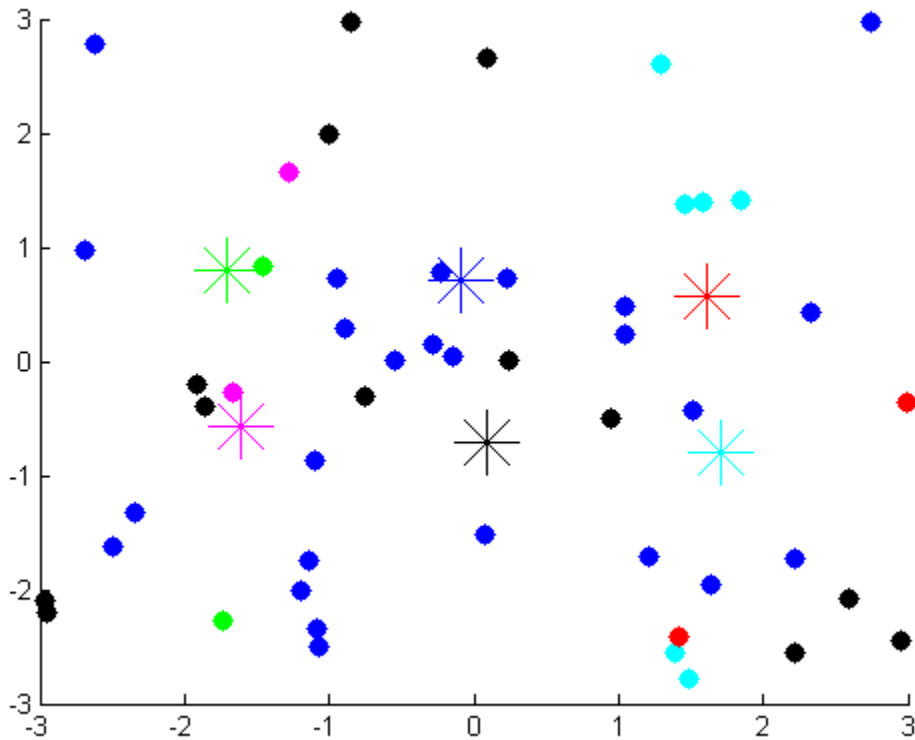
```
% rng(14,'twister')
% Uncomment the previous line to get the same output
ms = MultiStart;
opts = optimoptions(@fmincon,'Algorithm','interior-point');
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
+ x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
'objective',sixmin,'lb',[-3,-3],'ub',[3,3],...
'options',opts);
[xminm,fminm,flagm,outptm,manyminsm] = run(ms,problem,50);

possColors = 'kbgcrm';
hold on
for i = 1:size(manyminsm,2)

    % Color of this line
    cIdx = rem(i-1, length(possColors)) + 1;
    color = possColors(cIdx);

    % Plot start points
    u = manyminsm(i).X0;
    x0ThisMin = reshape([u{:}], 2, length(u));
    plot(x0ThisMin(1, :), x0ThisMin(2, :), '.', ...
        'Color',color,'MarkerSize',25);

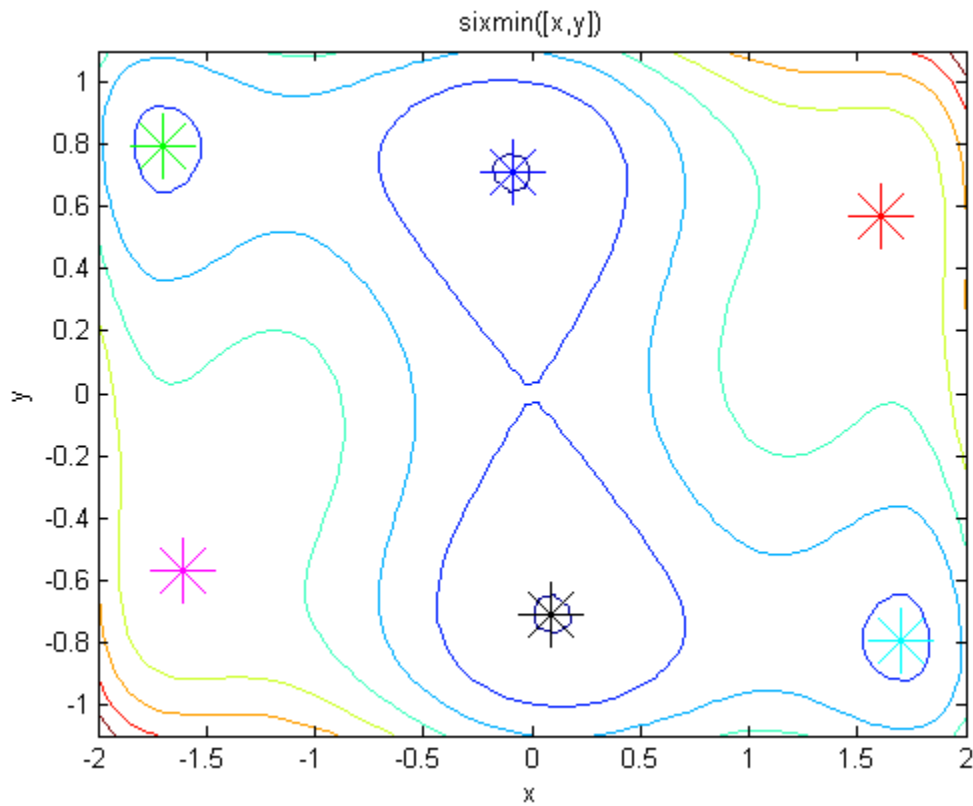
    % Plot the basin with color i
    plot(manyminsm(i).X(1), manyminsm(i).X(2), '*', ...
        'Color', color, 'MarkerSize',25);
end % basin center marked with a *, start points with dots
hold off
```



The figure shows the centers of the basins by colored * symbols. Start points with the same color as the * symbol converge to the center of the * symbol.

Start points do not always converge to the closest basin. For example, the red points are closer to the cyan basin center than to the red basin center. Also, many black and blue start points are closer to the opposite basin centers.

The magenta and red basins are shallow, as you can see in the following contour plot.



See Also

Related Examples

- "Multiple Solutions" on page 4-17

Output Functions for GlobalSearch and MultiStart

In this section...

“What Are Output Functions?” on page 4-27

“GlobalSearch Output Function” on page 4-27

“No Parallel Output Functions” on page 4-28

What Are Output Functions?

Output functions allow you to examine intermediate results in an optimization. Additionally, they allow you to halt a solver programmatically.

There are two types of output functions, like the two types of output structures on page 4-23:

- Global output functions run after each local solver run. They also run when the global solver starts and ends.
- Local output functions run after each iteration of a local solver. See “Output Functions for Optimization Toolbox”.

To use global output functions:

- Write output functions using the syntax described in “OutputFcn” on page 17-3.
- Set the `OutputFcn` property of your `GlobalSearch` or `MultiStart` solver to the function handle of your output function. You can use multiple output functions by setting the `OutputFcn` property to a cell array of function handles.

GlobalSearch Output Function

This output function stops `GlobalSearch` after it finds five distinct local minima with positive exit flags, or after it finds a local minimum value less than 0.5. The output function uses a persistent local variable, `foundLocal`, to store the local results. `foundLocal` enables the output function to determine whether a local solution is distinct from others, to within a tolerance of $1e-4$.

To store local results using nested functions instead of persistent variables, see “Example of a Nested Output Function”.

- 1 Write the output function using the syntax described in “OutputFcn” on page 17-3.

```
function stop = StopAfterFive(optimValues, state)
persistent foundLocal
stop = false;
switch state
    case 'init'
        foundLocal = []; % initialized as empty
    case 'iter'
        newf = optimValues.localsolution.Fval;
        exitflag = optimValues.localsolution.Exitflag;
        % Now check if the exit flag is positive and
        % the new value differs from all others by at least 1e-4
        % If so, add the new value to the newf list
        if exitflag > 0 && all(abs(newf - foundLocal) > 1e-4)
            foundLocal = [foundLocal;newf];
```

```

        % Now check if the latest value added to foundLocal
        % is less than 1/2
        % Also check if there are 5 local minima in foundLocal
        % If so, then stop
        if foundLocal(end) < 0.5 || length(foundLocal) >= 5
            stop = true;
        end
    end
end
end

```

2 Save `StopAfterFive.m` as a file in a folder on your MATLAB path.

3 Write the objective function and create an optimization problem structure as in “Find Global or Multiple Local Minima” on page 4-57.

```

function f = sawtoothxy(x,y)
[t r] = cart2pol(x,y); % change to polar coordinates
h = cos(2*t - 1/2)/2 + cos(t) + 2;
g = (sin(r) - sin(2*r)/2 + sin(3*r)/3 - sin(4*r)/4 + 4) ...
    .*r.^2./(r+1);
f = g.*h;
end

```

4 Save `sawtoothxy.m` as a file in a folder on your MATLAB path.

5 At the command line, create the problem structure:

```

problem = createOptimProblem('fmincon',...
    'objective',@(x)sawtoothxy(x(1),x(2)),...
    'x0',[100,-50],'options',...
    optimoptions(@fmincon,'Algorithm','sqp'));

```

6 Create a `GlobalSearch` object with `@StopAfterFive` as the output function, and set the iterative display property to `'iter'`.

```

gs = GlobalSearch('OutputFcn',@StopAfterFive,'Display','iter');

```

7 (Optional) To get the same answer as this example, set the default random number stream.

```

rng default

```

8 Run the problem.

```

[x,fval] = run(gs,problem)

```

Num Pts Analyzed	F-count	Best f(x)	Current Penalty	Threshold Penalty	Local f(x)	Local exitflag	Procedure
0	200	555.5			555.5	0	Initial Point
200	1463	1.547e-15			1.547e-15	1	Stage 1 Local

```

GlobalSearch stopped by the output or plot function.

```

```

1 out of 2 local solver runs converged with a positive local solver exit flag.

```

```

x =

```

```

    1.0e-07 *
    0.0414    0.1298

```

```

fval =

```

```

    1.5467e-15

```

The run stopped early because `GlobalSearch` found a point with a function value less than 0.5.

No Parallel Output Functions

While `MultiStart` can run in parallel, it does not support global output functions and plot functions in parallel. Furthermore, while local output functions and plot functions run on workers when

`MultiStart` runs in parallel, the effect differs from running serially. Local output and plot functions do not create a display when running on workers. You do not see any other effects of output and plot functions until the worker passes its results to the client (the originator of the `MultiStart` parallel jobs).

For information on running `MultiStart` in parallel, see “Parallel Computing”.

See Also

Related Examples

- “Global or Multiple Starting Point Search”
- “Plot Functions for GlobalSearch and MultiStart” on page 4-30

Plot Functions for GlobalSearch and MultiStart

In this section...

“What Are Plot Functions?” on page 4-30

“MultiStart Plot Function” on page 4-30

“No Parallel Plot Functions” on page 4-33

What Are Plot Functions?

The `PlotFcn` field of options specifies one or more functions that an optimization function calls at each iteration. Plot functions plot various measures of progress while the algorithm executes. Pass a function handle or cell array of function handles. The structure of a plot function is the same as the structure of an output function. For more information on this structure, see “OutputFcn” on page 17-3.

Plot functions are specialized output functions (see “Output Functions for GlobalSearch and MultiStart” on page 4-27). There are two predefined plot functions:

- `@gsplotbestf` plots the best objective function value.
- `@gsplotfuncount` plots the number of function evaluations.

Plot function windows have **Pause** and **Stop** buttons. By default, all plots appear in one window.

To use global plot functions:

- Write plot functions using the syntax described in “OutputFcn” on page 17-3.
- Set the `PlotFcn` property of your `GlobalSearch` or `MultiStart` object to the function handle of your plot function. You can use multiple plot functions by setting the `PlotFcn` property to a cell array of function handles.

Details of Built-In Plot Functions

The built-in plot functions have characteristics that can surprise you.

- `@gsplotbestf` can have plots that are not strictly decreasing. This is because early values can result from local solver runs with negative exit flags (such as infeasible solutions). A subsequent local solution with positive exit flag is better even if its function value is higher. Once a local solver returns a value with a positive exit flag, the plot is monotone decreasing.
- `@gsplotfuncount` might not plot the total number of function evaluations. This is because `GlobalSearch` can continue to perform function evaluations after it calls the plot function for the last time. For more information, see “GlobalSearch Algorithm” on page 4-35.

MultiStart Plot Function

This example plots the number of local solver runs it takes to obtain a better local minimum for `MultiStart`. The example also uses a built-in plot function to show the current best function value.

The example problem is the same as in “Find Global or Multiple Local Minima” on page 4-57, with additional bounds.

The example uses persistent variables to store previous best values. The plot function examines the best function value after each local solver run, available in the `bestfval` field of the `optimValues` structure. If the value is not lower than the previous best, the plot function adds 1 to the number of consecutive calls with no improvement and draws a bar chart. If the value is lower than the previous best, the plot function starts a new bar in the chart with value 1. Before plotting, the plot function takes a logarithm of the number of consecutive calls. The logarithm helps keep the plot legible, since some values can be much larger than others.

To store local results using nested functions instead of persistent variables, see “Example of a Nested Output Function”.

Plot Function Example

This example minimizes the `sawtoothxy` helper function, which is listed at the end of this example on page 4-32. In general, save your objective function in a file on your MATLAB® path.

The `NumberToNextBest` custom plot function is attached to this example. In general, save your plot function in a file on your MATLAB path. Here is a listing.

type `NumberToNextBest`

```
function stop = NumberToNextBest(optimValues, state)

persistent bestfv bestcounter

stop = false;
switch state
    case 'init'
        % Initialize variable to record best function value.
        bestfv = [];

        % Initialize counter to record number of
        % local solver runs to find next best minimum.
        bestcounter = 1;

        % Create the histogram.
        bar(log(bestcounter), 'tag', 'NumberToNextBest');
        xlabel('Number of New Best Fval Found');
        ylabel('Log Number of Local Solver Runs');
        title('Number of Local Solver Runs to Find Lower Minimum')
    case 'iter'
        % Find the axes containing the histogram.
        NumToNext = ...
            findobj(get(gca, 'Children'), 'Tag', 'NumberToNextBest');

        % Update the counter that records number of local
        % solver runs to find next best minimum.
        if ~isequal(optimValues.bestfval, bestfv)
            bestfv = optimValues.bestfval;
            bestcounter = [bestcounter 1];
        else
            bestcounter(end) = bestcounter(end) + 1;
        end

        % Update the histogram.
        set(NumToNext, 'Ydata', log(bestcounter))
end
```

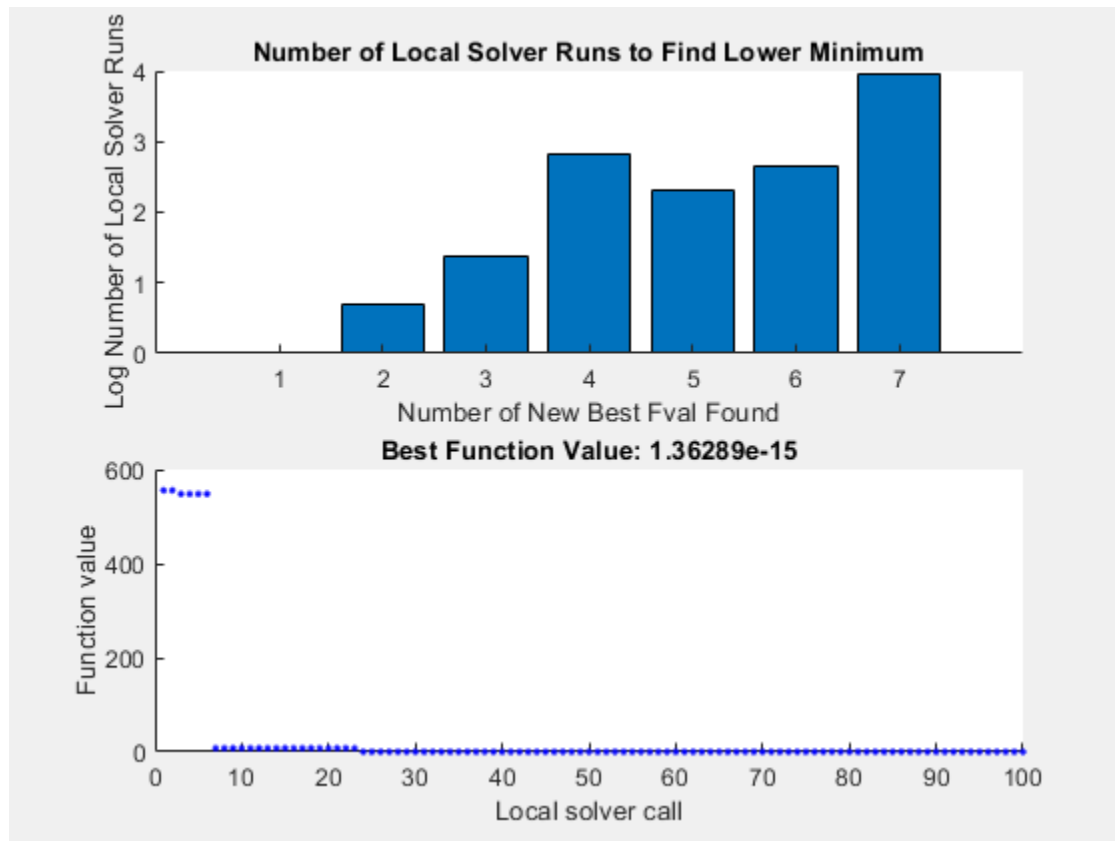
Create the problem structure and global solver object. Set lower bounds of $[-3e3, -4e3]$, upper bounds of $[4e3, 3e3]$ and set the global solver to use the `NumberToNextBest` custom plot function and the `gsplotbestf` built-in plot function.

```
problem = createOptimProblem('fmincon',...
    'objective',@(x)sawtoothxy(x(1),x(2)),...
    'x0',[100,-50],'lb',[-3e3 -4e3],...
    'ub',[4e3,3e3],'options',...
    optimoptions(@fmincon,'Algorithm','sqp'));

ms = MultiStart('PlotFcn',{@NumberToNextBest,@gsplotbestf});
```

Run the global solver for 100 local solver runs.

```
rng(2); % For reproducibility
[x,fv] = run(ms,problem,100);
```



MultiStart completed some of the runs from the start points.

33 out of 100 local solver runs converged with a positive local solver exit flag.

Helper Functions

This code creates the `sawtoothxy` helper function.

```
function f = sawtoothxy(x,y)
[t,r] = cart2pol(x,y); % change to polar coordinates
```

```
h = cos(2*t - 1/2)/2 + cos(t) + 2;  
g = (sin(r) - sin(2*r)/2 + sin(3*r)/3 - sin(4*r)/4 + 4) ...  
    .*r.^2./(r+1);  
f = g.*h;  
end
```

No Parallel Plot Functions

While `MultiStart` can run in parallel, it does not support global output functions and plot functions in parallel. Furthermore, while local output functions and plot functions run on workers when `MultiStart` runs in parallel, the effect differs from running serially. Local output and plot functions do not create a display when running on workers. You do not see any other effects of output and plot functions until the worker passes its results to the client (the originator of the `MultiStart` parallel jobs).

For information on running `MultiStart` in parallel, see “Parallel Computing”.

See Also

Related Examples

- “Global or Multiple Starting Point Search”
- “Output Functions for GlobalSearch and MultiStart” on page 4-27

How GlobalSearch and MultiStart Work

In this section...

“Multiple Runs of a Local Solver” on page 4-34

“Differences Between the Solver Objects” on page 4-34

“GlobalSearch Algorithm” on page 4-35

“MultiStart Algorithm” on page 4-38

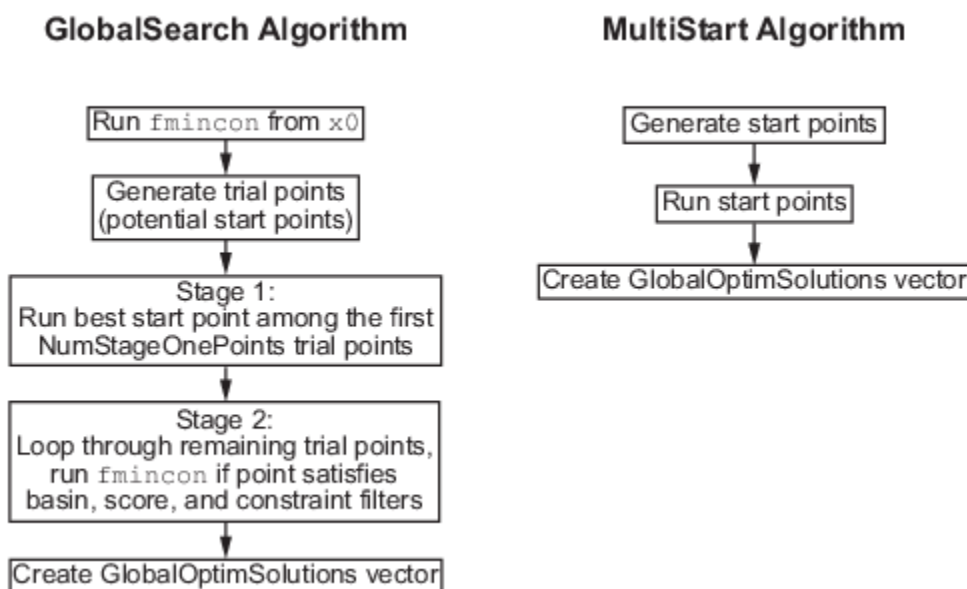
“Bibliography” on page 4-40

Multiple Runs of a Local Solver

GlobalSearch and MultiStart have similar approaches to finding global or multiple minima. Both algorithms start a local solver (such as `fmincon`) from multiple start points. The algorithms use multiple start points to sample multiple basins of attraction. For more information, see “Basins of Attraction” on page 1-25.

Differences Between the Solver Objects

“GlobalSearch and MultiStart Algorithm Overview” on page 4-34 contains a sketch of the GlobalSearch and MultiStart algorithms.



GlobalSearch and MultiStart Algorithm Overview

The main differences between GlobalSearch and MultiStart are:

- GlobalSearch uses a scatter-search mechanism for generating start points. MultiStart uses uniformly distributed start points within bounds, or user-supplied start points.
- GlobalSearch analyzes start points and rejects those points that are unlikely to improve the best local minimum found so far. MultiStart runs all start points (or, optionally, all start points that are feasible with respect to bounds or inequality constraints).

- `MultiStart` gives a choice of local solver: `fmincon`, `fminunc`, `lsqcurvefit`, or `lsqnonlin`. The `GlobalSearch` algorithm uses `fmincon`.
- `MultiStart` can run in parallel, distributing start points to multiple processors for local solution. To run `MultiStart` in parallel, see “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11.

Deciding Which Solver to Use

The differences between these solver objects boil down to the following decision on which to use:

- Use `GlobalSearch` to find a single global minimum most efficiently on a single processor.
- Use `MultiStart` to:
 - Find multiple local minima.
 - Run in parallel.
 - Use a solver other than `fmincon`.
 - Search thoroughly for a global minimum.
 - Explore your own start points.

GlobalSearch Algorithm

For a description of the algorithm, see Ugray et al. [1].

When you run a `GlobalSearch` object, the algorithm performs the following steps:

1. “Run `fmincon` from `x0`” on page 4-35
2. “Generate Trial Points” on page 4-35
3. “Obtain Stage 1 Start Point, Run” on page 4-36
4. “Initialize Basins, Counters, Threshold” on page 4-36
5. “Begin Main Loop” on page 4-36
6. “Examine Stage 2 Trial Point to See if `fmincon` Runs” on page 4-36
7. “When `fmincon` Runs” on page 4-37
8. “When `fmincon` Does Not Run” on page 4-37
9. “Create `GlobalOptimSolution`” on page 4-38

Run `fmincon` from `x0`

`GlobalSearch` runs `fmincon` from the start point you give in the problem structure. If this run converges, `GlobalSearch` records the start point and end point for an initial estimate on the radius of a basin of attraction. Furthermore, `GlobalSearch` records the final objective function value for use in the score function (see “Obtain Stage 1 Start Point, Run” on page 4-36).

The score function is the sum of the objective function value at a point and a multiple of the sum of the constraint violations. So a feasible point has score equal to its objective function value. The multiple for constraint violations is initially 1000. `GlobalSearch` updates the multiple during the run.

Generate Trial Points

`GlobalSearch` uses the scatter search algorithm to generate a set of `NumTrialPoints` trial points. Trial points are potential start points. For a description of the scatter search algorithm, see Glover

[2]. GlobalSearch generates trial points within any finite bounds you set (lb and ub). Unbounded components have artificial bounds imposed: $lb = -1e4 + 1$, $ub = 1e4 + 1$. This range is not symmetric about the origin so that the origin is not in the scatter search. Components with one-sided bounds have artificial bounds imposed on the unbounded side, shifted by the finite bounds to keep $lb < ub$.

Obtain Stage 1 Start Point, Run

GlobalSearch evaluates the score function of a set of NumStageOnePoints trial points. It then takes the point with the best score and runs fmincon from that point. GlobalSearch removes the set of NumStageOnePoints trial points from its list of points to examine.

Initialize Basins, Counters, Threshold

The localSolverThreshold is initially the smaller of the two objective function values at the solution points. The solution points are the fmincon solutions starting from x0 and from the Stage 1 start point. If both of these solution points do not exist or are infeasible, localSolverThreshold is initially the penalty function value of the Stage 1 start point.

The GlobalSearch heuristic assumption is that basins of attraction are spherical. The initial estimate of basins of attraction for the solution point from x0 and the solution point from Stage 1 are spheres centered at the solution points. The radius of each sphere is the distance from the initial point to the solution point. These estimated basins can overlap.

There are two sets of counters associated with the algorithm. Each counter is the number of consecutive trial points that:

- Lie within a basin of attraction. There is one counter for each basin.
- Have score function greater than localSolverThreshold. For a definition of the score, see "Run fmincon from x0" on page 4-35.

All counters are initially 0.

Begin Main Loop

GlobalSearch repeatedly examines a remaining trial point from the list, and performs the following steps. It continually monitors the time, and stops the search if elapsed time exceeds MaxTime seconds.

Examine Stage 2 Trial Point to See if fmincon Runs

Call the trial point p. Run fmincon from p if the following conditions hold:

- p is not in any existing basin. The criterion for every basin i is:
 $|p - \text{center}(i)| > \text{DistanceThresholdFactor} * \text{radius}(i)$.

DistanceThresholdFactor is an option (default value 0.75).

radius is an estimated radius that updates in Update Basin Radius and Threshold on page 4-37 and React to Large Counter Values on page 4-38.

- $\text{score}(p) < \text{localSolverThreshold}$.
- (optional) p satisfies bound and/or inequality constraints. This test occurs if you set the StartPointsToRun property of the GlobalSearch object to 'bounds' or 'bounds-ineqs'.

When fmincon Runs

1 Reset Counters

Set the counters for basins and threshold to 0.

2 Update Solution Set

If `fmincon` runs starting from `p`, it can yield a positive exit flag, which indicates convergence. In that case, `GlobalSearch` updates the vector of `GlobalOptimSolution` objects. Call the solution point `xp` and the objective function value `fp`. There are two cases:

- For every other solution point `xq` with objective function value `fq`,

$$|xq - xp| > XTolerance * \max(1, |xp|)$$

or

$$|fq - fp| > FunctionTolerance * \max(1, |fp|).$$

In this case, `GlobalSearch` creates a new element in the vector of `GlobalOptimSolution` objects. For details of the information contained in each object, see `GlobalOptimSolution`.

- For some other solution point `xq` with objective function value `fq`,

$$|xq - xp| \leq XTolerance * \max(1, |xp|)$$

and

$$|fq - fp| \leq FunctionTolerance * \max(1, |fp|).$$

In this case, `GlobalSearch` regards `xp` as equivalent to `xq`. The `GlobalSearch` algorithm modifies the `GlobalOptimSolution` of `xq` by adding `p` to the cell array of `X0` points.

There is one minor tweak that can happen to this update. If the exit flag for `xq` is greater than 1, and the exit flag for `xp` is 1, then `xp` replaces `xq`. This replacement can lead to some points in the same basin being more than a distance of `XTolerance` from `xp`.

3 Update Basin Radius and Threshold

If the exit flag of the current `fmincon` run is positive:

- Set threshold to the score value at start point `p`.
- Set basin radius for `xp` equal to the maximum of the existing radius (if any) and the distance between `p` and `xp`.

4 Report to Iterative Display

When the `GlobalSearch Display` property is `'iter'`, every point that `fmincon` runs creates one line in the `GlobalSearch` iterative display.

When fmincon Does Not Run

1 Update Counters

Increment the counter for every basin containing `p`. Reset the counter of every other basin to 0.

Increment the threshold counter if `score(p) >= localSolverThreshold`. Otherwise, reset the counter to 0.

2 React to Large Counter Values

For each basin with counter equal to `MaxWaitCycle`, multiply the basin radius by `1 - BasinRadiusFactor`. Reset the counter to 0. (Both `MaxWaitCycle` and `BasinRadiusFactor` are settable properties of the `GlobalSearch` object.)

If the threshold counter equals `MaxWaitCycle`, increase the threshold:

`new threshold = threshold + PenaltyThresholdFactor*(1 + abs(threshold)).`

Reset the counter to 0.

3 Report to Iterative Display

Every 200th trial point creates one line in the `GlobalSearch` iterative display.

Create GlobalOptimSolution

After reaching `MaxTime` seconds or running out of trial points, `GlobalSearch` creates a vector of `GlobalOptimSolution` objects. `GlobalSearch` orders the vector by objective function value, from lowest (best) to highest (worst). This concludes the algorithm.

MultiStart Algorithm

When you run a `MultiStart` object, the algorithm performs the following steps:

- “Validate Inputs” on page 4-38
- “Generate Start Points” on page 4-38
- “Filter Start Points (Optional)” on page 4-39
- “Run Local Solver” on page 4-39
- “Check Stopping Conditions” on page 4-39
- “Create GlobalOptimSolution Object” on page 4-39

Validate Inputs

`MultiStart` checks input arguments for validity. Checks include running the local solver once on problem inputs. Even when run in parallel, `MultiStart` performs these checks serially.

Generate Start Points

If you call `MultiStart` with the syntax

```
[x,fval] = run(ms,problem,k)
```

for an integer `k`, `MultiStart` generates `k - 1` start points exactly as if you used a `RandomStartPointSet` object. The algorithm also uses the `x0` start point from the `problem` structure, for a total of `k` start points.

A `RandomStartPointSet` object does not have any points stored inside the object. Instead, `MultiStart` calls `list`, which generates random points within the bounds given by the `problem` structure. If an unbounded component exists, `list` uses an artificial bound given by the `ArtificialBound` property of the `RandomStartPointSet` object.

If you provide a `CustomStartPointSet` object, `MultiStart` does not generate start points, but uses the points in the object.

Filter Start Points (Optional)

If you set the `StartPointsToRun` property of the `MultiStart` object to `'bounds'` or `'bounds-ineqs'`, `MultiStart` does not run the local solver from infeasible start points. In this context, “infeasible” means start points that do not satisfy bounds, or start points that do not satisfy both bounds and inequality constraints.

The default setting of `StartPointsToRun` is `'all'`. In this case, `MultiStart` does not discard infeasible start points.

Run Local Solver

`MultiStart` runs the local solver specified in `problem.solver`, starting at the points that pass the `StartPointsToRun` filter. If `MultiStart` is running in parallel, it sends start points to worker processors one at a time, and the worker processors run the local solver.

At each of its iterations, the local solver checks whether `MaxTime` seconds have elapsed since `MultiStart` began calculating. If so, `MultiStart` exits that iteration without reporting a solution.

When the local solver stops, `MultiStart` stores the results and continues to the next step.

Report to Iterative Display

When the `MultiStart.Display` property is `'iter'`, every point that the local solver runs creates one line in the `MultiStart` iterative display.

Check Stopping Conditions

`MultiStart` stops when it runs out of start points. It also stops when it exceeds a total run time of `MaxTime` seconds.

Create GlobalOptimSolution Object

After `MultiStart` reaches a stopping condition, the algorithm creates a vector of `GlobalOptimSolution` objects as follows:

- 1 Sort the local solutions by objective function value (`Fval`) from lowest to highest. For the `lsqnonlin` and `lsqcurvefit` local solvers, the objective function is the norm of the residual.
- 2 Loop over the local solutions `j` beginning with the lowest (best) `Fval`.
- 3 Find all the solutions `k` satisfying both:

$$|Fval(k) - Fval(j)| \leq \text{FunctionTolerance} * \max(1, |Fval(j)|)$$

$$|x(k) - x(j)| \leq \text{XTolerance} * \max(1, |x(j)|)$$
- 4 Record `j`, `Fval(j)`, the local solver output structure for `j`, and a cell array of the start points for `j` and all the `k`. Remove those points `k` from the list of local solutions. This point is one entry in the vector of `GlobalOptimSolution` objects.

The resulting vector of `GlobalOptimSolution` objects is in order by `Fval`, from lowest (best) to highest (worst).

Report to Iterative Display

After examining all the local solutions, `MultiStart` gives a summary to the iterative display. This summary includes the number of local solver runs that converged, the number that failed to converge, and the number that had errors.

Bibliography

- [1] Ugray, Zsolt, Leon Lasdon, John C. Plummer, Fred Glover, James Kelly, and Rafael Martí. *Scatter Search and Local NLP Solvers: A Multistart Framework for Global Optimization*. INFORMS Journal on Computing, Vol. 19, No. 3, 2007, pp. 328-340.
- [2] Glover, F. "A template for scatter search and path relinking." *Artificial Evolution* (J.-K. Hao, E.Lutton, E.Ronald, M.Schoenauer, D.Snyers, eds.). Lecture Notes in Computer Science, 1363, Springer, Berlin/Heidelberg, 1998, pp. 13-54.
- [3] Dixon, L. and G. P. Szegö. "The Global Optimization Problem: an Introduction." *Towards Global Optimisation 2* (Dixon, L. C. W. and G. P. Szegö, eds.). Amsterdam, The Netherlands: North Holland, 1978.

See Also

Related Examples

- "Global or Multiple Starting Point Search"

Can You Certify That a Solution Is Global?

In this section...

“No Guarantees” on page 4-41

“Check if a Solution Is a Local Solution with patternsearch” on page 4-41

“Identify a Bounded Region That Contains a Global Solution” on page 4-42

“Use MultiStart with More Start Points” on page 4-42

No Guarantees

How can you tell if you have located the global minimum of your objective function? The short answer is that you cannot; you have no guarantee that the result of a Global Optimization Toolbox solver is a global optimum. While all Global Optimization Toolbox solvers repeatedly attempt to locate a global solution, no solver employs an algorithm that can certify a solution as global.

However, you can use the strategies in this section for investigating solutions.

Check if a Solution Is a Local Solution with patternsearch

Before you can determine if a purported solution is a global minimum, first check that it is a local minimum. To do so, run `patternsearch` on the problem.

To convert the problem to use `patternsearch` instead of `fmincon` or `fminunc`, enter

```
problem.solver = 'patternsearch';
```

Also, change the start point to the solution you just found, and clear the options:

```
problem.x0 = x;
problem.options = [];
```

For example, Check Nearby Points shows the following:

```
options = optimoptions(@fmincon,'Algorithm','active-set');
ffun = @(x)(x(1)-(x(1)-x(2))^2);
problem = createOptimProblem('fmincon', ...
    'objective',ffun,'x0',[1/2 1/3], ...
    'lb',[0 -1],'ub',[1 1],'options',options);
[x,fval,exitflag] = fmincon(problem)
```

```
x =
    1.0e-007 *
         0    0.1614
```

```
fval =
    -2.6059e-016
```

```
exitflag =
         1
```

However, checking this purported solution with `patternsearch` shows that there is a better solution. Start `patternsearch` from the reported solution `x`:

```
% set the candidate solution x as the start point
problem.x0 = x;
problem.solver = 'patternsearch';
problem.options = [];
[xp,fvalp,exitflagp] = patternsearch(problem)

Optimization terminated: mesh size less than options.MeshTolerance.

xp =

    1.0000   -1.0000

fvalp =

   -3.0000

exitflagp =

     1
```

Identify a Bounded Region That Contains a Global Solution

Suppose you have a smooth objective function in a bounded region. Given enough time and start points, `MultiStart` eventually locates a global solution.

Therefore, if you can bound the region where a global solution can exist, you can obtain some degree of assurance that `MultiStart` locates the global solution.

For example, consider the function

$$f = x^6 + y^6 + \sin(x + y)(x^2 + y^2) - \cos\left(\frac{x^2}{1 + y^2}\right)(2 + x^4 + x^2y^2 + y^4).$$

The initial summands $x^6 + y^6$ force the function to become large and positive for large values of $|x|$ or $|y|$. The components of the global minimum of the function must be within the bounds

$$-10 \leq x, y \leq 10,$$

since 10^6 is much larger than all the multiples of 10^4 that occur in the other summands of the function.

You can identify smaller bounds for this problem; for example, the global minimum is between -2 and 2 . It is more important to identify reasonable bounds than it is to identify the best bounds.

Use MultiStart with More Start Points

To check whether there is a better solution to your problem, run `MultiStart` with additional start points. Use `MultiStart` instead of `GlobalSearch` for this task because `GlobalSearch` does not run the local solver from all start points.

For example, see “Example: Searching for a Better Solution” on page 4-46.

See Also

Related Examples

- “Refine Start Points” on page 4-44
- “What Is Global Optimization?” on page 1-24

Refine Start Points

In this section...

“About Refining Start Points” on page 4-44

“Methods of Generating Start Points” on page 4-44

“Example: Searching for a Better Solution” on page 4-46

About Refining Start Points

If some components of your problem are unconstrained, `GlobalSearch` and `MultiStart` use artificial bounds to generate random start points uniformly in each component. However, if your problem has far-flung minima, you need widely dispersed start points to find these minima.

Use these methods to obtain widely dispersed start points:

- Give widely separated bounds in your problem structure.
- Use a `RandomStartPointSet` object with the `MultiStart` algorithm. Set a large value of the `ArtificialBound` property in the `RandomStartPointSet` object.
- Use a `CustomStartPointSet` object with the `MultiStart` algorithm. Use widely dispersed start points.

There are advantages and disadvantages of each method.

Method	Advantages	Disadvantages
Give bounds in problem	Automatic point generation	Makes a more complex Hessian
	Can use with <code>GlobalSearch</code>	Unclear how large to set the bounds
	Easy to do	Changes problem
	Bounds can be asymmetric	Only uniform points
Large <code>ArtificialBound</code> in <code>RandomStartPointSet</code>	Automatic point generation	<code>MultiStart</code> only
	Does not change problem	Only symmetric, uniform points
	Easy to do	Unclear how large to set <code>ArtificialBound</code>
<code>CustomStartPointSet</code>	Customizable	<code>MultiStart</code> only
	Does not change problem	Requires programming for generating points

Methods of Generating Start Points

- “Uniform Grid” on page 4-44
- “Perturbed Grid” on page 4-45
- “Widely Dispersed Points for Unconstrained Components” on page 4-45

Uniform Grid

To generate a uniform grid of start points:

- 1 Generate multidimensional arrays with `ndgrid`. Give the lower bound, spacing, and upper bound for each component.

For example, to generate a set of three-dimensional arrays with

- First component from -2 through 0, spacing 0.5
- Second component from 0 through 2, spacing 0.25
- Third component from -10 through 5, spacing 1

```
[X,Y,Z] = ndgrid(-2:.5:0,0:.25:2,-10:5);
```

- 2 Place the arrays into a single matrix, with each row representing one start point. For example:

```
W = [X(:),Y(:),Z(:)];
```

In this example, `W` is a 720-by-3 matrix.

- 3 Put the matrix into a `CustomStartPointSet` object. For example:

```
custpts = CustomStartPointSet(W);
```

Call `run` with the `CustomStartPointSet` object as the third input. For example,

```
% Assume problem structure and ms MultiStart object exist
[x,fval,flag,output,manymins] = run(ms,problem,custpts);
```

Perturbed Grid

Integer start points can yield less robust solutions than slightly perturbed start points.

To obtain a perturbed set of start points:

- 1 Generate a matrix of start points as in steps 1-2 of “Uniform Grid” on page 4-44.
- 2 Perturb the start points by adding a random normal matrix with 0 mean and relatively small variance.

For the example in “Uniform Grid” on page 4-44, after making the `W` matrix, add a perturbation:

```
[X,Y,Z] = ndgrid(-2:.5:0,0:.25:2,-10:5);
W = [X(:),Y(:),Z(:)];
W = W + 0.01*randn(size(W));
```

- 3 Put the matrix into a `CustomStartPointSet` object. For example:

```
custpts = CustomStartPointSet(W);
```

Call `run` with the `CustomStartPointSet` object as the third input. For example,

```
% Assume problem structure and ms MultiStart object exist
[x,fval,flag,output,manymins] = run(ms,problem,custpts);
```

Widely Dispersed Points for Unconstrained Components

Some components of your problem can lack upper or lower bounds. For example:

- Although no explicit bounds exist, there are levels that the components cannot attain. For example, if one component represents the weight of a single diamond, there is an implicit upper bound of 1 kg (the Hope Diamond is under 10 g). In such a case, give the implicit bound as an upper bound.

- There truly is no upper bound. For example, the size of a computer file in bytes has no effective upper bound. The largest size can be in gigabytes or terabytes today, but in 10 years, who knows?

For truly unbounded components, you can use the following methods of sampling. To generate approximately $1/n$ points in each region $(\exp(n), \exp(n+1))$, use the following formula. If u is random and uniformly distributed from 0 through 1, then $r = 2u - 1$ is uniformly distributed between -1 and 1. Take

$$y = \text{sgn}(r)(\exp(1/|r|) - e).$$

y is symmetric and random. For a variable bounded below by lb , take

$$y = \text{lb} + (\exp(1/u) - e).$$

Similarly, for a variable bounded above by ub , take

$$y = \text{ub} - (\exp(1/u) - e).$$

For example, suppose you have a three-dimensional problem with

- $x(1) > 0$
- $x(2) < 100$
- $x(3)$ unconstrained

To make 150 start points satisfying these constraints:

```
u = rand(150,3);
r1 = 1./u(:,1);
r1 = exp(r1) - exp(1);
r2 = 1./u(:,2);
r2 = -exp(r2) + exp(1) + 100;
r3 = 1./(2*u(:,3)-1);
r3 = sign(r3).*(exp(abs(r3)) - exp(1));
custpts = CustomStartPointSet([r1,r2,r3]);
```

The following is a variant of this algorithm. Generate a number between 0 and infinity by the method for lower bounds. Use this number as the radius of a point. Generate the other components of the point by taking random numbers for each component and multiply by the radius. You can normalize the random numbers, before multiplying by the radius, so their norm is 1. For a worked example of this method, see “MultiStart Without Bounds, Widely Dispersed Start Points” on page 4-89.

Example: Searching for a Better Solution

MultiStart fails to find the global minimum in “Find Global or Multiple Local Minima” on page 4-57. There are two simple ways to search for a better solution:

- Use more start points
- Give tighter bounds on the search space

Set up the problem structure and MultiStart object:

```
problem = createOptimProblem('fminunc',...
    'objective',@(x)sawtoothxy(x(1),x(2)),...
    'x0',[100,-50],'options',...)
```



```

    optimoptions(@fminunc,'Algorithm','quasi-newton');
ms = MultiStart;

```

Use More Start Points

Run MultiStart on the problem for 200 start points instead of 50:

```

rng(14,'twister') % for reproducibility
[x,fval,exitflag,output,manymins] = run(ms,problem,200)

```

MultiStart completed some of the runs from the start points.

53 out of 200 local solver runs converged with a positive local solver exit flag.

x =

```

    1.0e-06 *
    -0.2284  -0.5567

```

fval =

```

    2.1382e-12

```

exitflag =

```

    2

```

output =

struct with fields:

```

    funcCount: 32670
    localSolverTotal: 200
    localSolverSuccess: 53
    localSolverIncomplete: 147
    localSolverNoSolution: 0

```

message: 'MultiStart completed some of the runs from the start points. ←←53 out

manymins =

```

    1x53 GlobalOptimSolution

```

Properties:

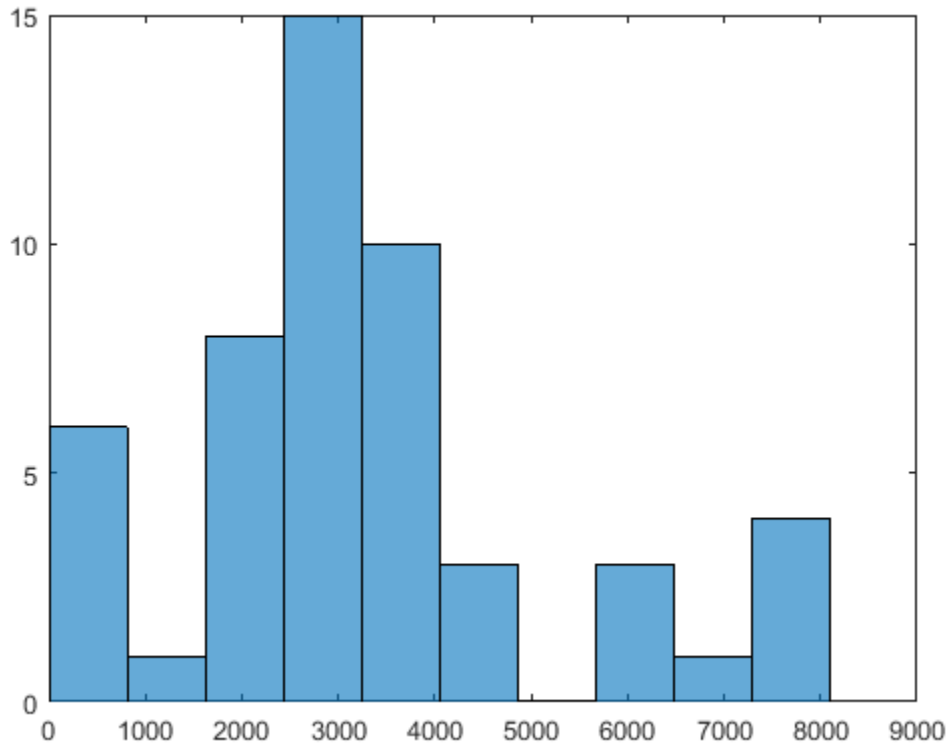
```

    X
    Fval
    Exitflag
    Output
    X0

```

This time MultiStart found the global minimum, and found 51 local minima.

To see the range of local solutions, enter `histogram([manymins.Fval],10)`.



Tighter Bound on the Start Points

Suppose you believe that the interesting local solutions have absolute values of all components less than 100. The default value of the bound on start points is 1000. To use a different value of the bound, generate a `RandomStartPointSet` with the `ArtificialBound` property set to 100:

```
startpts = RandomStartPointSet('ArtificialBound',100,...
    'NumStartPoints',50);
[x,fval,exitflag,output,manymins] = run(ms,problem,startpts)
```

MultiStart completed some of the runs from the start points.

29 out of 50 local solver runs converged with a positive local solver exit flag.

x =

```
1.0e-08 *
    0.9725   -0.6198
```

fval =

```
1.4955e-15
```

exitflag =

2

output =

struct with fields:

```
      funcCount: 7431
      localSolverTotal: 50
      localSolverSuccess: 29
      localSolverIncomplete: 21
      localSolverNoSolution: 0
      message: 'MultiStart completed some of the runs from the start points. ←29 out
```

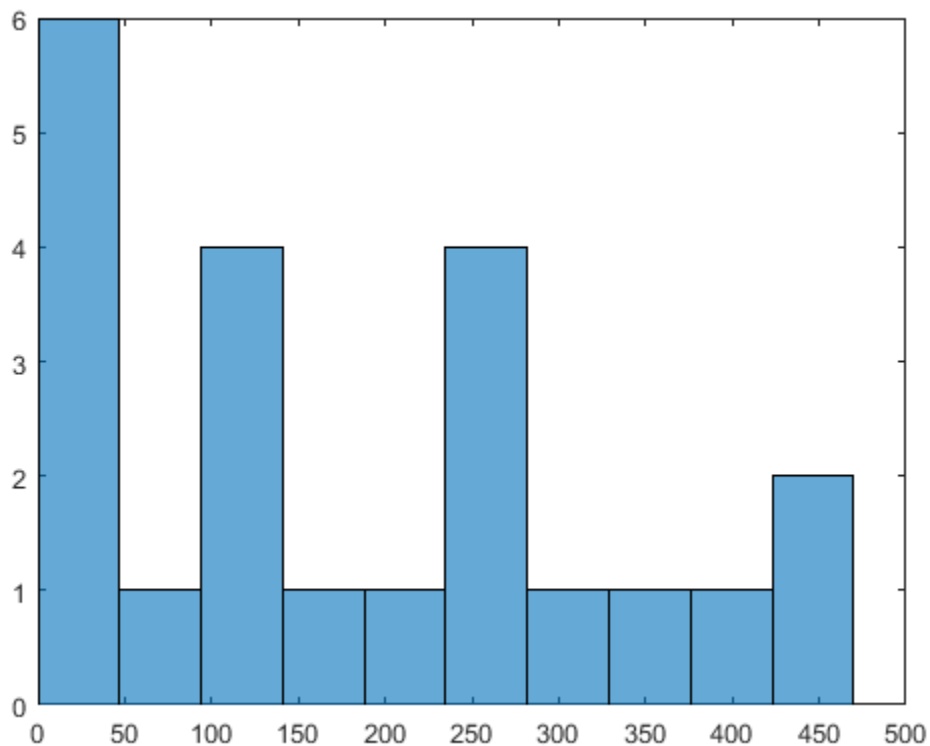
manymins =

1x25 GlobalOptimSolution

Properties:

```
X
Fval
Exitflag
Output
X0
```

MultiStart found the global minimum, and found 22 distinct local solutions. To see the range of local solutions, enter `histogram([manymins.Fval],10)`.



Compared to the minima found in “Use More Start Points” on page 4-47, this run found better (smaller) minima, and had a higher percentage of successful runs.

See Also

Related Examples

- “Global or Multiple Starting Point Search”
- “Isolated Global Minimum” on page 4-86

Change Options

In this section...

“How to Determine Which Options to Change” on page 4-51

“Changing Local Solver Options” on page 4-51

“Changing Global Options” on page 4-52

How to Determine Which Options to Change

After you run a global solver, you might want to change some global or local options. To determine which options to change, the guiding principle is:

- To affect the local solver, set local solver options.
- To affect the start points or solution set, change the `problem` structure, or set the global solver object properties.

For example, to obtain:

- More local minima — Set global solver object properties.
- Faster local solver iterations — Set local solver options.
- Different tolerances for considering local solutions identical (to obtain more or fewer local solutions) — Set global solver object properties.
- Different information displayed at the command line — Decide if you want iterative display from the local solver (set local solver options) or global information (set global solver object properties).
- Different bounds, to examine different regions — Set the bounds in the `problem` structure.

Examples of Choosing Problem Options

- To start your local solver at points only satisfying inequality constraints, set the `StartPointsToRun` property in the global solver object to `'bounds - ineqs'`. This setting can speed your solution, since local solvers do not have to attempt to find points satisfying these constraints. However, the setting can result in many fewer local solver runs, since the global solver can reject many start points. For an example, see “Optimize Using Only Feasible Start Points” on page 4-76.
- To use the `fmincon` interior-point algorithm, set the local solver `Algorithm` option to `'interior-point'`. For an example showing how to do this, see “Examples of Updating Problem Options” on page 4-52.
- For your local solver to have different bounds, set the bounds in the `problem` structure. Examine different regions by setting bounds.
- To see every solution that has positive local exit flag, set the `XTolerance` property in the global solver object to 0. For an example showing how to do this, see “Changing Global Options” on page 4-52.

Changing Local Solver Options

There are several ways to change values in local options:

- Update the values using dot notation and `optimoptions`. The syntax is

```
problem.options = optimoptions(problem.options,'Parameter',value,...);
```

You can also replace the local options entirely:

```
problem.options = optimoptions(@solvername,'Parameter',value,...);
```

- Use dot notation on one local option. The syntax is

```
problem.options.Parameter = newvalue;
```

- Recreate the entire problem structure. For details, see “Create Problem Structure” on page 4-4.

Examples of Updating Problem Options

- 1 Create a problem structure:

```
problem = createOptimProblem('fmincon','x0',[-1 2], ...
    'objective',@rosenboth);
```

- 2 Set the problem to use the sqp algorithm in fmincon:

```
problem.options.Algorithm = 'sqp';
```

- 3 Update the problem to use the gradient in the objective function, have a FunctionTolerance value of 1e-8, and a XTolerance value of 1e-7:

```
problem.options = optimoptions(problem.options,'GradObj','on', ...
    'FunctionTolerance',1e-8,'XTolerance',1e-7);
```

Changing Global Options

There are several ways to change characteristics of a GlobalSearch or MultiStart object:

- Use dot notation. For example, suppose you have a default MultiStart object:

```
ms = MultiStart
ms =
```

MultiStart with properties:

```
    UseParallel: 0
      Display: 'final'
FunctionTolerance: 1.0000e-06
      MaxTime: Inf
      OutputFcn: []
      PlotFcn: []
StartPointsToRun: 'all'
      XTolerance: 1.0000e-06
```

To change ms to have its XTolerance value equal to 1e-3, update the XTolerance field:

```
ms.XTolerance = 1e-3
ms =
```

MultiStart with properties:

```
    UseParallel: 0
      Display: 'final'
FunctionTolerance: 1.0000e-06
      MaxTime: Inf
      OutputFcn: []
```

```

        PlotFcn: []
    StartPointsToRun: 'all'
    XTolerance: 1.0000e-03

```

- Reconstruct the object starting from the current settings. For example, to set the `FunctionTolerance` field in `ms` to `1e-3`, retaining the nondefault value for `XTolerance`:

```

ms = MultiStart(ms,'FunctionTolerance',1e-3)
ms =

```

MultiStart with properties:

```

        UseParallel: 0
        Display: 'final'
    FunctionTolerance: 1.0000e-03
        MaxTime: Inf
        OutputFcn: []
        PlotFcn: []
    StartPointsToRun: 'all'
    XTolerance: 1.0000e-03

```

- Convert a `GlobalSearch` object to a `MultiStart` object, or vice-versa. For example, with the `ms` object from the previous example, create a `GlobalSearch` object with the same values of `XTolerance` and `FunctionTolerance`:

```

gs = GlobalSearch(ms)
gs =

```

GlobalSearch with properties:

```

        NumTrialPoints: 1000
        BasinRadiusFactor: 0.2000
    DistanceThresholdFactor: 0.7500
        MaxWaitCycle: 20
        NumStageOnePoints: 200
    PenaltyThresholdFactor: 0.2000
        Display: 'final'
    FunctionTolerance: 1.0000e-03
        MaxTime: Inf
        OutputFcn: []
        PlotFcn: []
    StartPointsToRun: 'all'
    XTolerance: 1.0000e-03

```

See Also

Related Examples

- “Global or Multiple Starting Point Search”

Reproduce Results

In this section...

“Identical Answers with Pseudorandom Numbers” on page 4-54

“Steps to Take in Reproducing Results” on page 4-54

“Example: Reproducing a GlobalSearch or MultiStart Result” on page 4-54

“Parallel Processing and Random Number Streams” on page 4-55

Identical Answers with Pseudorandom Numbers

GlobalSearch and MultiStart use pseudorandom numbers in choosing start points. Use the same pseudorandom number stream again to:

- Compare various algorithm settings.
- Have an example run repeatably.
- Extend a run, with known initial segment of a previous run.

Both GlobalSearch and MultiStart use the default random number stream.

Steps to Take in Reproducing Results

- 1 Before running your problem, store the current state of the default random number stream:

```
stream = rng;
```

- 2 Run your GlobalSearch or MultiStart problem.

- 3 Restore the state of the random number stream:

```
rng(stream)
```

- 4 If you run your problem again, you get the same result.

Example: Reproducing a GlobalSearch or MultiStart Result

This example shows how to obtain reproducible results for “Find Global or Multiple Local Minima” on page 4-57. The example follows the procedure in “Steps to Take in Reproducing Results” on page 4-54.

- 1 Store the current state of the default random number stream:

```
stream = rng;
```

- 2 Create the sawtoothxy function file:

```
function f = sawtoothxy(x,y)
[t r] = cart2pol(x,y); % change to polar coordinates
h = cos(2*t - 1/2)/2 + cos(t) + 2;
g = (sin(r) - sin(2*r)/2 + sin(3*r)/3 - sin(4*r)/4 + 4) ...
    .*r.^2./(r+1);
f = g.*h;
end
```

- 3 Create the problem structure and GlobalSearch object:

```
problem = createOptimProblem('fmincon',...
    'objective',@(x)sawtoothxy(x(1),x(2)),...
```



```
'x0',[100,-50],'options',...
optimoptions(@fmincon,'Algorithm','sqp'));
gs = GlobalSearch('Display','iter');
```

4 Run the problem:

```
[x,fval] = run(gs,problem)
```

Num Pts Analyzed	F-count	Best f(x)	Current Penalty	Threshold Penalty	Local f(x)	Local exitflag	Procedure
0	465	422.9			422.9	2	Initial Point
200	1730	1.547e-015			1.547e-015	1	Stage 1 Local
300	1830	1.547e-015	6.01e+004	1.074			Stage 2 Search
400	1930	1.547e-015	1.47e+005	4.16			Stage 2 Search
500	2030	1.547e-015	2.63e+004	11.84			Stage 2 Search
600	2130	1.547e-015	1.341e+004	30.95			Stage 2 Search
700	2230	1.547e-015	2.562e+004	65.25			Stage 2 Search
800	2330	1.547e-015	5.217e+004	163.8			Stage 2 Search
900	2430	1.547e-015	7.704e+004	409.2			Stage 2 Search
981	2587	1.547e-015	42.24	516.6	7.573	1	Stage 2 Local
1000	2606	1.547e-015	3.299e+004	42.24			Stage 2 Search

GlobalSearch stopped because it analyzed all the trial points.

All 3 local solver runs converged with a positive local solver exit flag.

```
x =
 1.0e-007 *
 0.0414    0.1298
```

```
fval =
 1.5467e-015
```

You might obtain a different result when running this problem, since the random stream was in an unknown state at the beginning of the run.

5 Restore the state of the random number stream:

```
rng(stream)
```

6 Run the problem again. You get the same output.

```
[x,fval] = run(gs,problem)
```

Num Pts Analyzed	F-count	Best f(x)	Current Penalty	Threshold Penalty	Local f(x)	Local exitflag	Procedure
0	465	422.9			422.9	2	Initial Point
200	1730	1.547e-015			1.547e-015	1	Stage 1 Local

... Output deleted to save space ...

```
x =
 1.0e-007 *
 0.0414    0.1298
```

```
fval =
 1.5467e-015
```

Parallel Processing and Random Number Streams

You obtain reproducible results from `MultiStart` when you run the algorithm in parallel the same way as you do for serial computation. Runs are reproducible because `MultiStart` generates pseudorandom start points locally, and then distributes the start points to parallel processors. Therefore, the parallel processors do not use random numbers.

To reproduce a parallel `MultiStart` run, use the procedure described in “Steps to Take in Reproducing Results” on page 4-54. For a description of how to run `MultiStart` in parallel, see “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11.

See Also

Related Examples

- “Global or Multiple Starting Point Search”

Find Global or Multiple Local Minima

This example illustrates how `GlobalSearch` finds a global minimum efficiently, and how `MultiStart` finds many more local minima.

The objective function for this example has many local minima and a unique global minimum. In polar coordinates, the function is

$$f(r, t) = g(r)h(t)$$

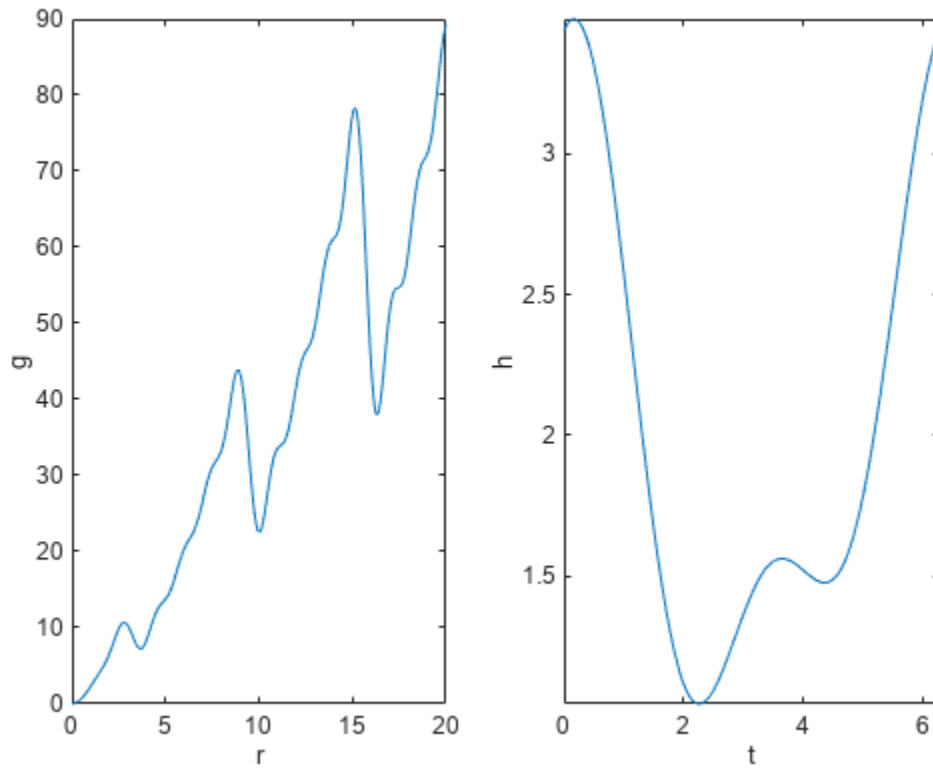
where

$$g(r) = \left(\sin(r) - \frac{\sin(2r)}{2} + \frac{\sin(3r)}{3} - \frac{\sin(4r)}{4} + 4 \right) \frac{r^2}{r+1}$$

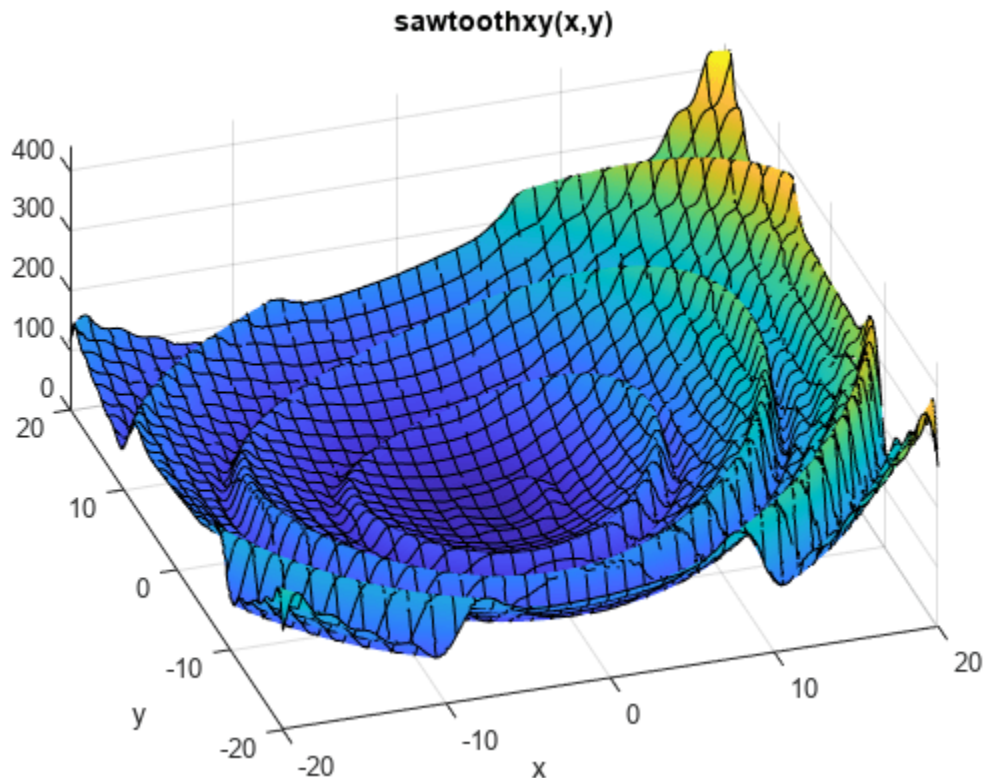
$$h(t) = 2 + \cos(t) + \frac{\cos\left(2t - \frac{1}{2}\right)}{2}.$$

Plot the functions g and h , and create a surface plot of the function f .

```
figure
subplot(1,2,1);
fplot(@(r)(sin(r) - sin(2*r)/2 + sin(3*r)/3 - sin(4*r)/4 + 4) .* r.^2./(r+1), [0 20])
title(''); ylabel('g'); xlabel('r');
subplot(1,2,2);
fplot(@(t)2 + cos(t) + cos(2*t-1/2)/2, [0 2*pi])
title(''); ylabel('h'); xlabel('t');
```



```
figure
fsurf(@(x,y) sawtoothxy(x,y), [-20 20])
% sawtoothxy is defined in the first step below
xlabel('x'); ylabel('y'); title('sawtoothxy(x,y)');
view(-18,52)
```



The global minimum is at $r = 0$, with objective function 0. The function $g(r)$ grows approximately linearly in r , with a repeating sawtooth shape. The function $h(t)$ has two local minima, one of which is global.

The `sawtoothxy.m` file converts from Cartesian to polar coordinates, then computes the value in polar coordinates.

type `sawtoothxy`

```
function f = sawtoothxy(x,y)
[t r] = cart2pol(x,y); % change to polar coordinates
h = cos(2*t - 1/2)/2 + cos(t) + 2;
g = (sin(r) - sin(2*r)/2 + sin(3*r)/3 - sin(4*r)/4 + 4) ...
    .*r.^2./(r+1);
f = g.*h;
end
```

Single Global Minimum Via GlobalSearch

To search for the global minimum using `GlobalSearch`, first create a problem structure. Use the 'sqp' algorithm for `fmincon`,

```
problem = createOptimProblem('fmincon',...
    'objective',@(x)sawtoothxy(x(1),x(2)),...
    'x0',[100,-50],'options',...
    optimoptions(@fmincon,'Algorithm','sqp','Display','off'));
```

The start point is [100, -50] instead of [0,0] so GlobalSearch does not start at the global solution.

Validate the problem structure by running `fmincon`.

```
[x,fval] = fmincon(problem)
```

```
x = 1x2
```

```
45.7038 -107.6610
```

```
fval = 555.6645
```

Create the `GlobalSearch` object, and set iterative display.

```
gs = GlobalSearch('Display','iter');
```

For reproducibility, set the random number generator seed.

```
rng(14,'twister')
```

Run the solver.

```
[x,fval] = run(gs,problem)
```

Num Pts Analyzed	F-count	Best f(x)	Current Penalty	Threshold Penalty	Local f(x)	Local exitflag	Procedure
0	200	555.7			555.7	0	Initial Point
200	1463	1.547e-15			1.547e-15	1	Stage 1 Local
300	1564	1.547e-15	5.858e+04	1.074			Stage 2 Search
400	1664	1.547e-15	1.84e+05	4.16			Stage 2 Search
500	1764	1.547e-15	2.683e+04	11.84			Stage 2 Search
600	1864	1.547e-15	1.122e+04	30.95			Stage 2 Search
700	1964	1.547e-15	1.353e+04	65.25			Stage 2 Search
800	2064	1.547e-15	6.249e+04	163.8			Stage 2 Search
900	2164	1.547e-15	4.119e+04	409.2			Stage 2 Search
950	2359	1.547e-15	477	589.7	387	2	Stage 2 Local
952	2423	1.547e-15	368.4	477	250.7	2	Stage 2 Local
1000	2471	1.547e-15	4.031e+04	530.9			Stage 2 Search

`GlobalSearch` stopped because it analyzed all the trial points.

3 out of 4 local solver runs converged with a positive local solver exit flag.

```
x = 1x2
```

```
10-7 x
```

```
0.0414 0.1298
```

```
fval = 1.5467e-15
```

The solver finds three local minima, including the global minimum near [0,0].

Multiple Local Minima Via MultiStart

To search for multiple minima using `MultiStart`, first create a problem structure. Because the problem is unconstrained, use the `fminunc` solver. Set options not to show any display at the command line.

```

problem = createOptimProblem('fminunc',...
    'objective',@(x)sawtoothxy(x(1),x(2)),...
    'x0',[100,-50],'options',...
    optimoptions(@fminunc,'Display','off'));

```

Validate the problem structure by running it.

```
[x,fval] = fminunc(problem)
```

```
x = 1×2
```

```
    8.4420 -110.2602
```

```
fval = 435.2573
```

Create a default MultiStart object.

```
ms = MultiStart;
```

Run the solver for 50 iterations, recording the local minima.

```
rng(1) % For reproducibility
[x,fval,eflag,output,manymins] = run(ms,problem,50)
```

MultiStart completed some of the runs from the start points.

10 out of 50 local solver runs converged with a positive local solver exit flag.

```
x = 1×2
```

```
   -73.8348 -197.7810
```

```
fval = 766.8260
```

```
eflag = 2
```

```
output = struct with fields:
    funcCount: 8583
    localSolverTotal: 50
    localSolverSuccess: 10
    localSolverIncomplete: 40
    localSolverNoSolution: 0
    message: 'MultiStart completed some of the runs from the start points....'
```

```
manymins=1×10 object
1×10 GlobalOptimSolution array with properties:
```

```

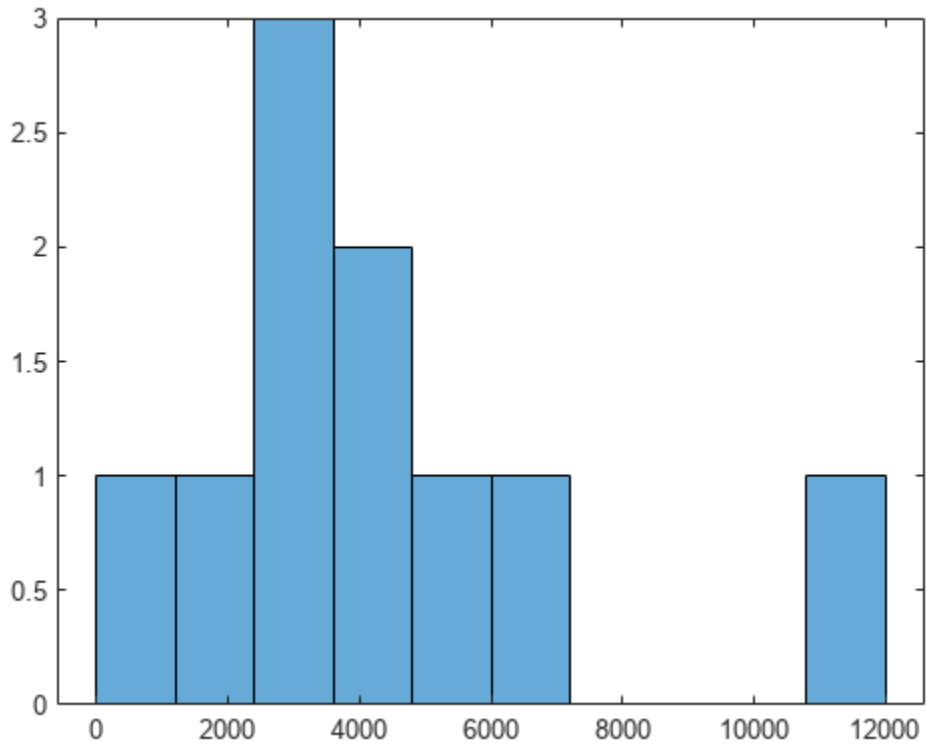
X
Fval
Exitflag
Output
X0

```

The solver does not find the global minimum near $[0,0]$. The solver finds 10 distinct local minima.

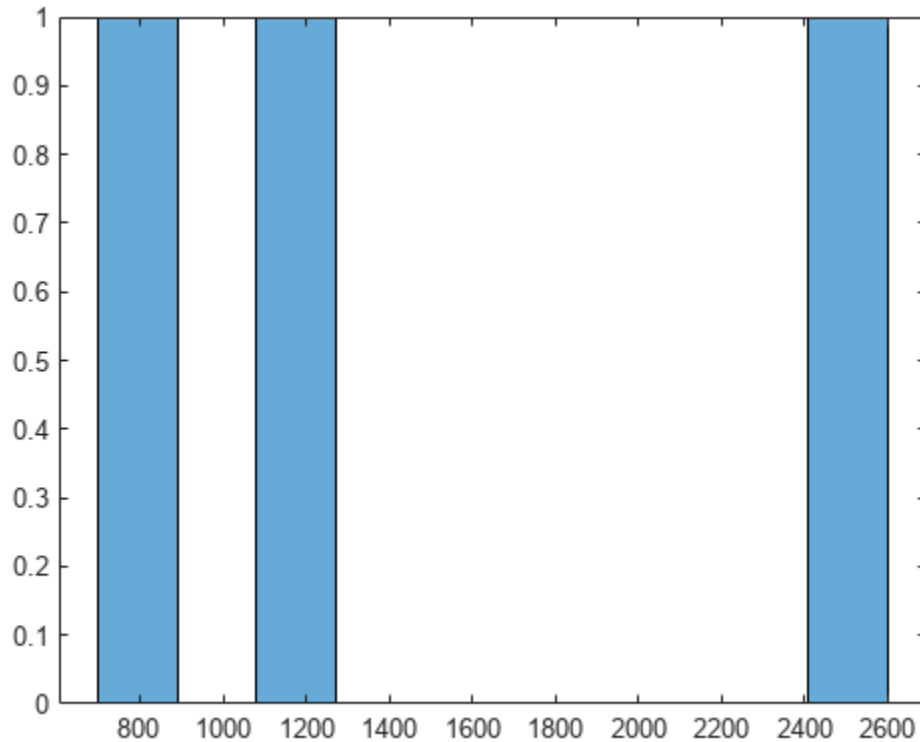
Plot the function values at the local minima:

```
histogram([manymins.Fval],10)
```



Plot the function values at the three best points:

```
bestf = [manymins.Fval];  
histogram(bestf(1:3),10)
```

`MultiStart` starts `fminunc` from start points with components uniformly distributed between -1000 and 1000. `fminunc` often gets stuck in one of the many local minima. `fminunc` exceeds its iteration limit or function evaluation limit 40 times.

See Also

`GlobalSearch` | `MultiStart` | `createOptimProblem`

More About

- “Workflow for `GlobalSearch` and `MultiStart`” on page 4-3
- “Visualize the Basins of Attraction” on page 4-24

Maximizing Monochromatic Polarized Light Interference Patterns Using GlobalSearch and MultiStart

This example shows how to use the functions `GlobalSearch` and `MultiStart`.

Introduction

This example shows how Global Optimization Toolbox functions, particularly `GlobalSearch` and `MultiStart`, can help locate the maximum of an electromagnetic interference pattern. For simplicity of modeling, the pattern arises from monochromatic polarized light spreading out from point sources.

The electric field due to source i measured in the direction of polarization at point x and time t is

$$E_i = \frac{A_i}{d_i(x)} \sin(\phi_i + \omega(t - d_i(x)/c)),$$

where ϕ_i is the phase at time zero for source i , c is the speed of light, ω is the frequency of the light, A_i is the amplitude of source i , and $d_i(x)$ is the distance from source i to x .

For a fixed point x the intensity of the light is the time average of the square of the net electric field. The net electric field is sum of the electric fields due to all sources. The time average depends only on the sizes and relative phases of the electric fields at x . To calculate the net electric field, add up the individual contributions using the phasor method. For phasors, each source contributes a vector. The length of the vector is the amplitude divided by distance from the source, and the angle of the vector, $\phi_i - \omega d_i(x)/c$ is the phase at the point.

For this example, we define three point sources with the same frequency (ω) and amplitude (A), but varied initial phase (ϕ_i). We arrange these sources on a fixed plane.

```
% Frequency is proportional to the number of peaks
relFreqConst = 2*pi*2.5;
amp = 2.2;
phase = -[0; 0.54; 2.07];

numSources = 3;
height = 3;

% All point sources are aligned at [x_i,y_i,z]
xcoords = [2.4112
           0.2064
           1.6787];
ycoords = [0.3957
           0.3927
           0.9877];
zcoords = height*ones(numSources,1);

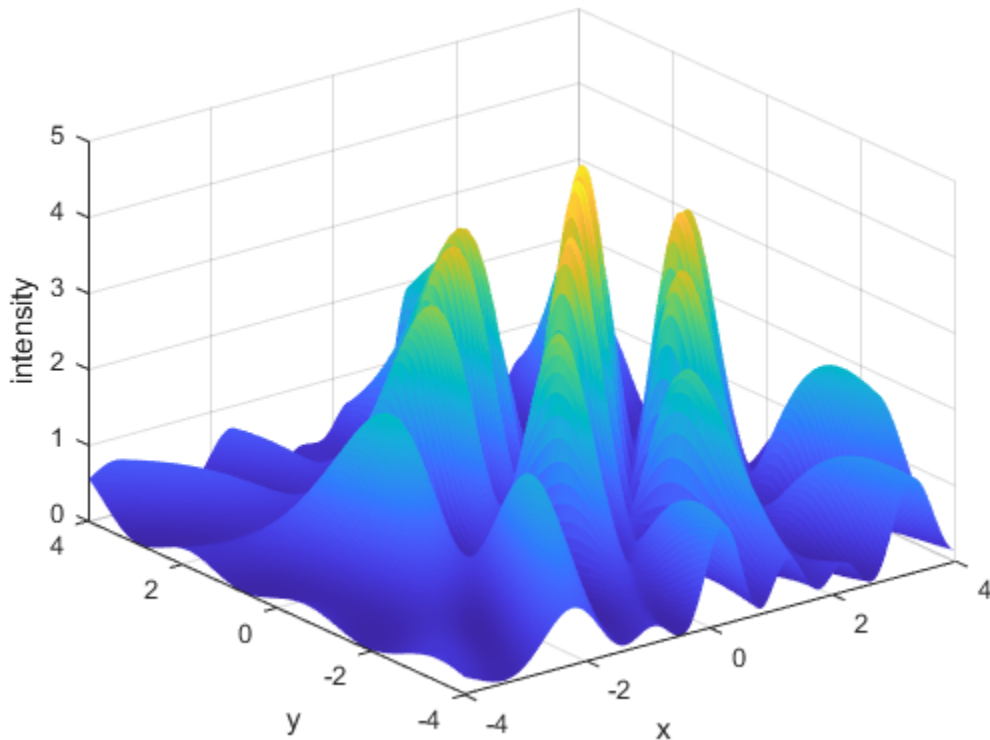
origins = [xcoords ycoords zcoords];
```

Visualize the Interference Pattern

Now let's visualize a slice of the interference pattern on the plane $z = 0$.

As you can see from the plot below, there are many peaks and valleys indicating constructive and destructive interference.

```
% Pass additional parameters via an anonymous function:
waveIntensity_x = @(x) waveIntensity(x,amp,phase, ...
    relFreqConst,numSources,origins);
% Generate the grid
[X,Y] = meshgrid(-4:0.035:4,-4:0.035:4);
% Compute the intensity over the grid
Z = arrayfun(@(x,y) waveIntensity_x([x y]),X,Y);
% Plot the surface and the contours
figure
surf(X,Y,Z,'EdgeColor','none')
xlabel('x')
ylabel('y')
zlabel('intensity')
```



Posing the Optimization Problem

We are interested in the location where this wave intensity reaches its highest peak.

The wave intensity (I) falls off as we move away from the source proportional to $1/d_i(x)$. Therefore, let's restrict the space of viable solutions by adding constraints to the problem.

If we limit the exposure of the sources with an aperture, then we can expect the maximum to lie in the intersection of the projection of the apertures onto our observation plane. We model the effect of an aperture by restricting the search to a circular region centered at each source.

We also restrict the solution space by adding bounds to the problem. Although these bounds may be redundant (given the nonlinear constraints), they are useful since they restrict the range in which start points are generated (see “How GlobalSearch and MultiStart Work” on page 4-34 for more information).

Now our problem has become:

$$\max_{x,y} I(x, y)$$

subject to

$$(x - x_{c1})^2 + (y - y_{c1})^2 \leq r_1^2$$

$$(x - x_{c2})^2 + (y - y_{c2})^2 \leq r_2^2$$

$$(x - x_{c3})^2 + (y - y_{c3})^2 \leq r_3^2$$

$$-0.5 \leq x \leq 3.5$$

$$-2 \leq y \leq 3$$

where (x_{cn}, y_{cn}) and r_n are the coordinates and aperture radius of the n^{th} point source, respectively. Each source is given an aperture with radius 3. The given bounds encompass the feasible region.

The objective ($I(x, y)$) and nonlinear constraint functions are defined in separate MATLAB® files, `waveIntensity.m` and `apertureConstraint.m`, respectively, which are listed at the end of this example.

Visualization with Constraints

Now let's visualize the contours of our interference pattern with the nonlinear constraint boundaries superimposed. The feasible region is the interior of the intersection of the three circles (yellow, green, and blue). The bounds on the variables are indicated by the dashed-line box.

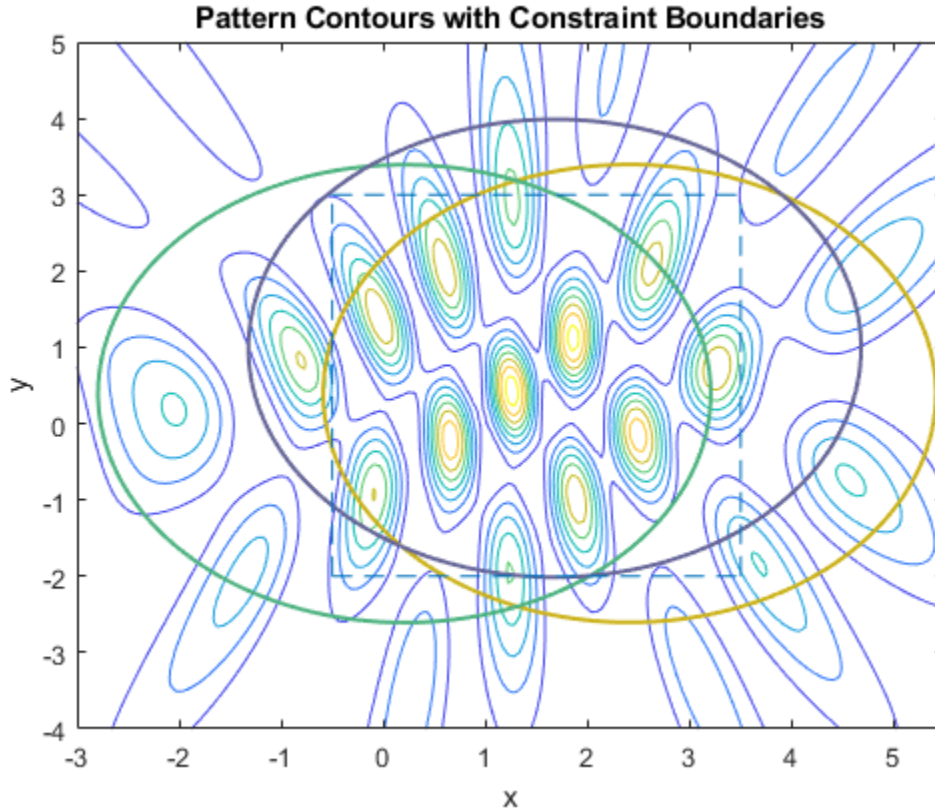
```
% Visualize the contours of our interference surface
domain = [-3 5.5 -4 5];
figure;
ezcontour(@X,Y) arrayfun(@(x,y) waveIntensity_x([x y]),X,Y),domain,150);
hold on

% Plot constraints
g1 = @(x,y) (x-xcoords(1)).^2 + (y-ycoords(1)).^2 - 9;
g2 = @(x,y) (x-xcoords(2)).^2 + (y-ycoords(2)).^2 - 9;
g3 = @(x,y) (x-xcoords(3)).^2 + (y-ycoords(3)).^2 - 9;
h1 = ezplot(g1,domain);
h1.Color = [0.8 0.7 0.1]; % yellow
h1.LineWidth = 1.5;
h2 = ezplot(g2,domain);
h2.Color = [0.3 0.7 0.5]; % green
h2.LineWidth = 1.5;
h3 = ezplot(g3,domain);
h3.Color = [0.4 0.4 0.6]; % blue
h3.LineWidth = 1.5;
```

```

% Plot bounds
lb = [-0.5 -2];
ub = [3.5 3];
line([lb(1) lb(1)],[lb(2) ub(2)],'LineStyle','--')
line([ub(1) ub(1)],[lb(2) ub(2)],'LineStyle','--')
line([lb(1) ub(1)],[lb(2) lb(2)],'LineStyle','--')
line([lb(1) ub(1)],[ub(2) ub(2)],'LineStyle','--')
title('Pattern Contours with Constraint Boundaries')

```



Setting Up and Solving the Problem with a Local Solver

Given the nonlinear constraints, we need a constrained nonlinear solver, namely, `fmincon`.

Let's set up a problem structure describing our optimization problem. We want to maximize the intensity function, so we negate the values returned from `waveIntensity`. Let's choose an arbitrary start point that happens to be near the feasible region.

For this small problem, we'll use `fmincon`'s SQP algorithm.

```

% Pass additional parameters via an anonymous function:
apertureConstraint_x = @(x) apertureConstraint(x,xcoords,ycoords);

% Set up fmincon's options
x0 = [3 -1];
opts = optimoptions('fmincon','Algorithm','sqp');
problem = createOptimProblem('fmincon','objective', ...
    @(x) -waveIntensity_x(x),'x0',x0,'lb',lb,'ub',ub, ...

```

```
    'nonlcon',apertureConstraint_x,'options',opts);  
  
% Call fmincon  
[xlocal,fvallocal] = fmincon(problem)
```

Local minimum found that satisfies the constraints.

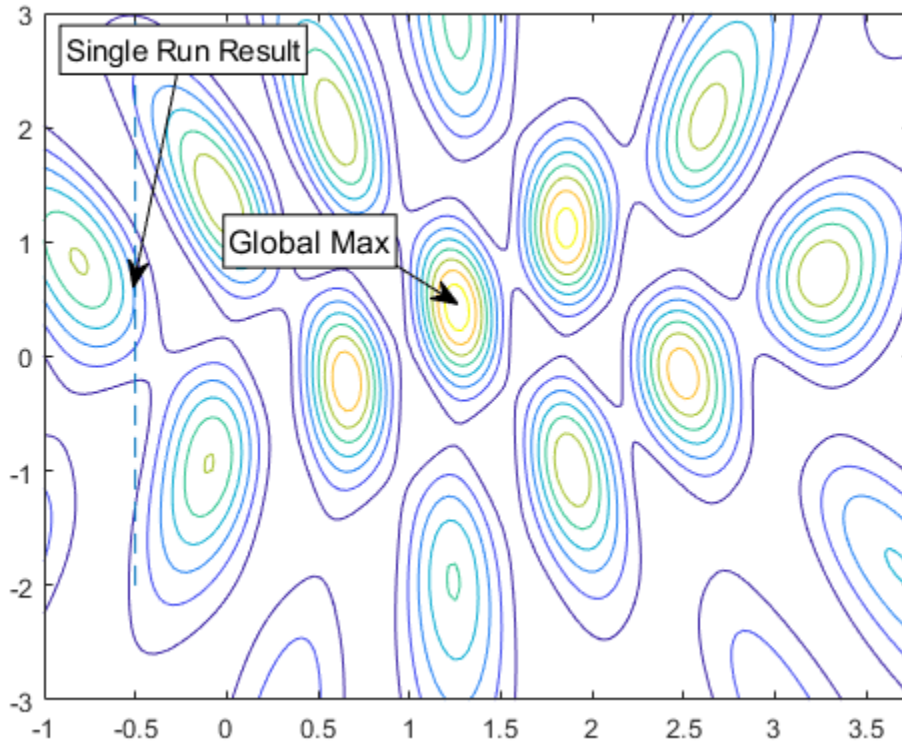
Optimization completed because the objective function is non-decreasing in feasible directions, to within the value of the optimality tolerance, and constraints are satisfied to within the value of the constraint tolerance.

```
xlocal =  
    -0.5000    0.4945
```

```
fvallocal =  
    -1.4438
```

Now, let's see how we did by showing the result of `fmincon` in our contour plot. Notice that `fmincon` did not reach the global maximum, which is also annotated on the plot. Note that we'll only plot the bound that was active at the solution.

```
[~,maxIdx] = max(Z(:));  
xmax = [X(maxIdx),Y(maxIdx)]  
figure  
contour(X,Y,Z)  
hold on  
  
% Show bounds  
line([lb(1) lb(1)],[lb(2) ub(2)],'LineStyle','--')  
  
% Create textarrow showing the location of xlocal  
annotation('textarrow',[0.25 0.21],[0.86 0.60],'TextEdgeColor',[0 0 0],...  
    'TextBackgroundColor',[1 1 1],'FontSize',11,'String',{'Single Run Result'});  
  
% Create textarrow showing the location of xglobal  
annotation('textarrow',[0.44 0.50],[0.63 0.58],'TextEdgeColor',[0 0 0],...  
    'TextBackgroundColor',[1 1 1],'FontSize',12,'String',{'Global Max'});  
  
axis([-1 3.75 -3 3])  
  
xmax =  
    1.2500    0.4450
```



Using GlobalSearch and MultiStart

Given an arbitrary initial guess, `fmincon` gets stuck at a nearby local maximum. Global Optimization Toolbox solvers, particularly `GlobalSearch` and `MultiStart`, give us a better chance at finding the global maximum since they will try `fmincon` from multiple generated initial points (or our own custom points, if we choose).

Our problem has already been set up in the problem structure, so now we construct our solver objects and run them. The first output from `run` is the location of the best result found.

```
% Construct a GlobalSearch object
gs = GlobalSearch;
% Construct a MultiStart object based on our GlobalSearch attributes
ms = MultiStart;

rng(4, 'twister') % for reproducibility

% Run GlobalSearch
tic;
[xgs,~,~,~,solsgs] = run(gs,problem);
toc
xgs

% Run MultiStart with 15 randomly generated points
tic;
[xms,~,~,~,solms] = run(ms,problem,15);
```

```
toc
xms
```

GlobalSearch stopped because it analyzed all the trial points.

All 14 local solver runs converged with a positive local solver exit flag.
Elapsed time is 0.229525 seconds.

```
xgs =
    1.2592    0.4284
```

MultiStart completed the runs from all start points.

All 15 local solver runs converged with a positive local solver exit flag.
Elapsed time is 0.109984 seconds.

```
xms =
    1.2592    0.4284
```

Examining Results

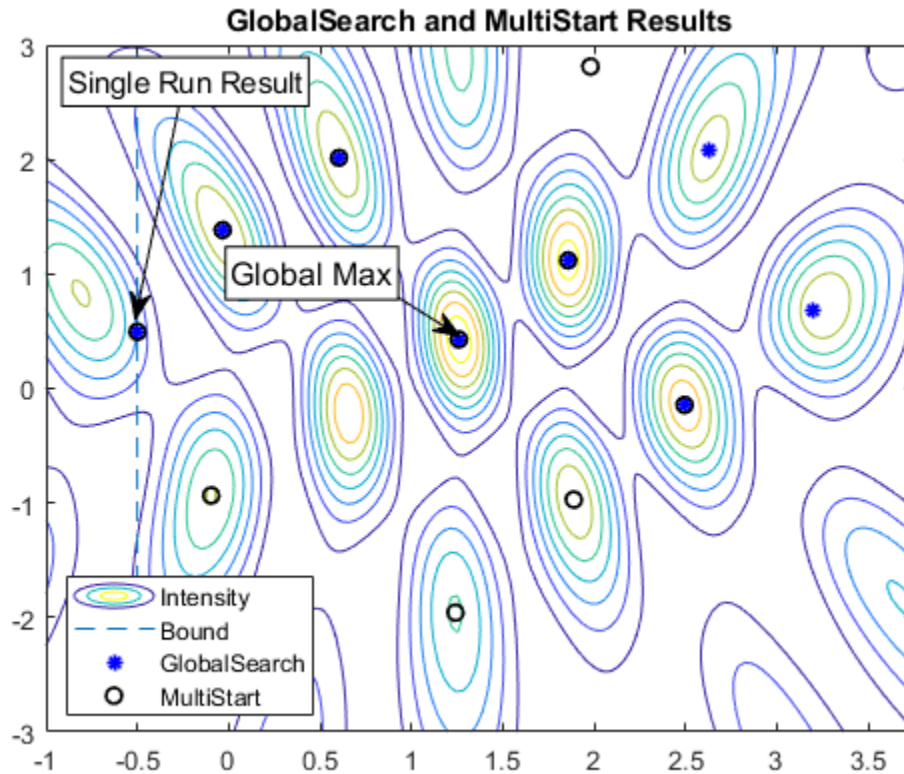
Let's examine the results that both solvers have returned. An important thing to note is that the results will vary based on the random start points created for each solver. Another run through this example may give different results. The coordinates of the best results `xgs` and `xms` printed to the command line. We'll show unique results returned by `GlobalSearch` and `MultiStart` and highlight the best results from each solver, in terms of proximity to the global solution.

The fifth output of each solver is a vector containing distinct minima (or maxima, in this case) found. We'll plot the (x,y) pairs of the results, `solsgs` and `solsms`, against our contour plot we used before.

```
% Plot GlobalSearch results using the '*' marker
xGS = cell2mat({solsgs(:).X}');
scatter(xGS(:,1),xGS(:,2),'*','MarkerEdgeColor',[0 0 1],'LineWidth',1.25)

% Plot MultiStart results using a circle marker
xMS = cell2mat({solsms(:).X}');
scatter(xMS(:,1),xMS(:,2),'o','MarkerEdgeColor',[0 0 0],'LineWidth',1.25)
legend('Intensity','Bound','GlobalSearch','MultiStart','Location','best')

title('GlobalSearch and MultiStart Results')
```

Relaxing the Bounds

With the tight bounds on the problem, both `GlobalSearch` and `MultiStart` were able to locate the global maximum in this run.

Finding tight bounds can be difficult to do in practice, when not much is known about the objective function or constraints. In general though, we may be able to guess a reasonable region in which we would like to restrict the set of start points. For illustration purposes, let's relax our bounds to define a larger area in which to generate start points and re-try the solvers.

```
% Relax the bounds to spread out the start points
```

```
problem.lb = -5*ones(2,1);
```

```
problem.ub = 5*ones(2,1);
```

```
% Run GlobalSearch
```

```
tic;
```

```
[xgs,~,~,~,solsgs] = run(gs,problem);
```

```
toc
```

```
xgs
```

```
% Run MultiStart with 15 randomly generated points
```

```
tic;
```

```
[xms,~,~,~,solms] = run(ms,problem,15);
```

```
toc
```

```
xms
```

`GlobalSearch` stopped because it analyzed all the trial points.

All 4 local solver runs converged with a positive local solver exit flag.
Elapsed time is 0.173760 seconds.

```
xgs =  
    0.6571    -0.2096
```

MultiStart completed the runs from all start points.

All 15 local solver runs converged with a positive local solver exit flag.
Elapsed time is 0.134150 seconds.

```
xms =  
    2.4947    -0.1439
```

```
% Show the contours
```

```
figure  
contour(X,Y,Z)  
hold on
```

```
% Create textarrow showing the location of xglobal
```

```
annotation('textarrow',[0.44 0.50],[0.63 0.58],'TextEdgeColor',[0 0 0],...  
          'TextBackgroundColor',[1 1 1],'FontSize',12,'String',{'Global Max'});  
axis([-1 3.75 -3 3])
```

```
% Plot GlobalSearch results using the '*' marker
```

```
xGS = cell2mat({solsgs(:).X});  
scatter(xGS(:,1),xGS(:,2),'*','MarkerEdgeColor',[0 0 1],'LineWidth',1.25)
```

```
% Plot MultiStart results using a circle marker
```

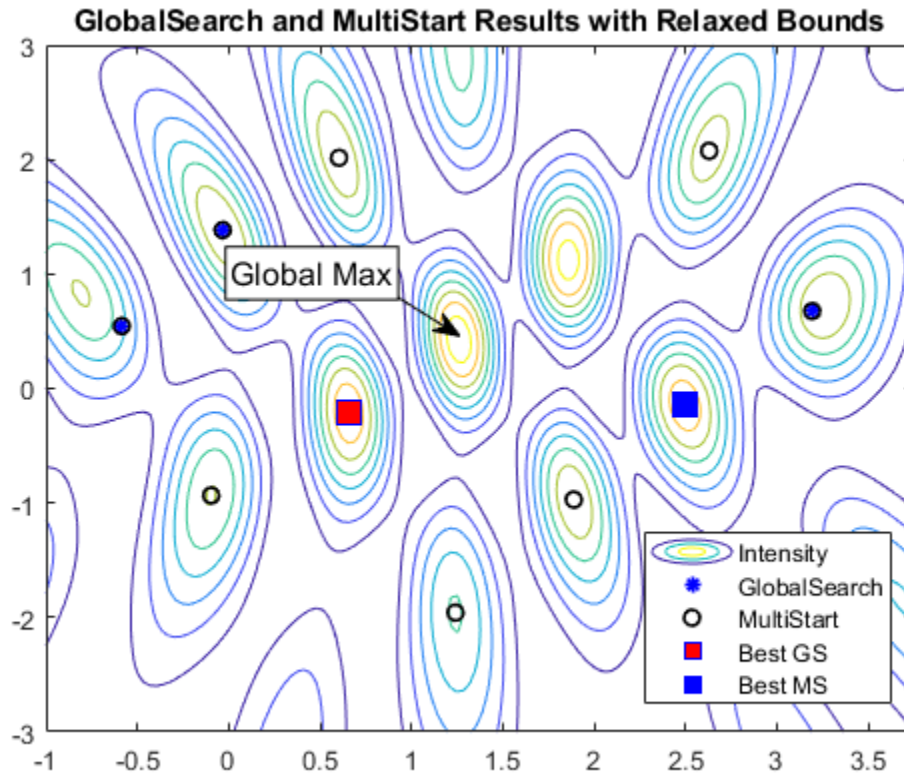
```
xMS = cell2mat({solms(:).X});  
scatter(xMS(:,1),xMS(:,2),'o','MarkerEdgeColor',[0 0 0],'LineWidth',1.25)
```

```
% Highlight the best results from each:
```

```
% GlobalSearch result in red, MultiStart result in blue
```

```
plot(xgs(1),xgs(2),'sb','MarkerSize',12,'MarkerFaceColor',[1 0 0])  
plot(xms(1),xms(2),'sb','MarkerSize',12,'MarkerFaceColor',[0 0 1])  
legend('Intensity','GlobalSearch','MultiStart','Best GS','Best MS','Location','best')
```

```
title('GlobalSearch and MultiStart Results with Relaxed Bounds')
```



The best result from GlobalSearch is shown by the red square and the best result from MultiStart is shown by the blue square.

Tuning GlobalSearch Parameters

Notice that in this run, given the larger area defined by the bounds, neither solver was able to identify the point of maximum intensity. We could try to overcome this in a couple of ways. First, we examine GlobalSearch.

Notice that GlobalSearch only ran `fmincon` a few times. To increase the chance of finding the global maximum, we would like to run more points. To restrict the start point set to the candidates most likely to find the global maximum, we'll instruct each solver to ignore start points that do not satisfy constraints by setting the `StartPointsToRun` property to `bounds-ineqs`. Additionally, we will set the `MaxWaitCycle` and `BasinRadiusFactor` properties so that GlobalSearch will be able to identify the narrow peaks quickly. Reducing `MaxWaitCycle` causes GlobalSearch to decrease the basin of attraction radius by the `BasinRadiusFactor` more often than with the default setting.

```
% Increase the total candidate points, but filter out the infeasible ones
gs = GlobalSearch(gs, 'StartPointsToRun', 'bounds-ineqs', ...
    'MaxWaitCycle', 3, 'BasinRadiusFactor', 0.3);
% Run GlobalSearch
tic;
xgs = run(gs, problem);
toc
xgs
```

GlobalSearch stopped because it analyzed all the trial points.

All 10 local solver runs converged with a positive local solver exit flag.
Elapsed time is 0.242955 seconds.

```
xgs =  
    1.2592    0.4284
```

Utilizing MultiStart's Parallel Capabilities

A brute force way to improve our chances of finding the global maximum is to simply try more start points. Again, this may not be practical in all situations. In our case, we've only tried a small set so far and the run time was not terribly long. So, it's reasonable to try more start points. To speed the computation we'll run MultiStart in parallel if Parallel Computing Toolbox™ is available.

```
% Set the UseParallel property of MultiStart  
ms = MultiStart(ms,'UseParallel',true);  
  
try  
    demoOpenedPool = false;  
    % Create a parallel pool if one does not already exist  
    % (requires Parallel Computing Toolbox)  
    if max(size(gcf)) == 0 % if no pool  
        parpool  
        demoOpenedPool = true;  
    end  
catch ME  
    warning(message('globaloptim:globaloptimdemos:opticalInterferenceDemo:noPCT'));  
end  
  
% Run the solver  
tic;  
xms = run(ms,problem,100);  
toc  
xms  
  
if demoOpenedPool  
    % Make sure to delete the pool if one was created in this example  
    delete(gcf) % delete the pool  
end
```

MultiStart completed the runs from all start points.

All 100 local solver runs converged with a positive local solver exit flag.
Elapsed time is 0.956671 seconds.

```
xms =  
    1.2592    0.4284
```

Objective and Nonlinear Constraints

Here we list the functions that define the optimization problem:

```
function p = waveIntensity(x,amp,phase,relFreqConst,numSources,origins)
% WaveIntensity Intensity function for opticalInterferenceDemo.

% Copyright 2009 The MathWorks, Inc.

d = distanceFromSource(x,numSources,origins);
ampVec = [sum(amp./d .* cos(phase - d*relFreqConst));
          sum(amp./d .* sin(phase - d*relFreqConst))];

% Intensity is ||AmpVec||^2
p = ampVec'*ampVec;

function [c,ceq] = apertureConstraint(x,xcoords,ycoords)
% apertureConstraint Aperture constraint function for opticalInterferenceDemo.

% Copyright 2009 The MathWorks, Inc.

ceq = [];
c = (x(1) - xcoords).^2 + (x(2) - ycoords).^2 - 9;

function d = distanceFromSource(v,numSources,origins)
% distanceFromSource Distance function for opticalInterferenceDemo.

% Copyright 2009 The MathWorks, Inc.

d = zeros(numSources,1);
for k = 1:numSources
    d(k) = norm(origins(k,:) - [v 0]);
end
```

See Also

GlobalSearch | MultiStart

More About

- “Example: Searching for a Better Solution” on page 4-46
- “Isolated Global Minimum” on page 4-86

Optimize Using Only Feasible Start Points

You can set the `StartPointsToRun` option so that `MultiStart` and `GlobalSearch` use only start points that satisfy inequality constraints. This option can speed your optimization, since the local solver does not have to search for a feasible region. However, the option can cause the solvers to miss some basins of attraction.

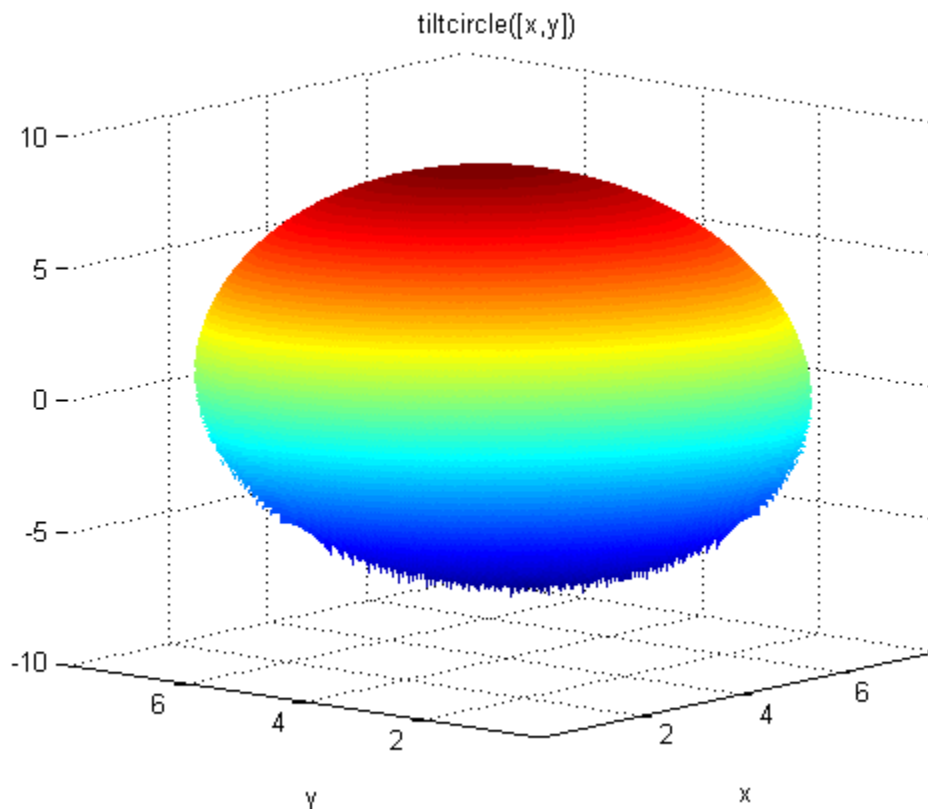
There are three settings for the `StartPointsToRun` option:

- `all` — Accepts all start points
- `bounds` — Rejects start points that do not satisfy bounds
- `bounds-ineqs` — Rejects start points that do not satisfy bounds or inequality constraints

For example, suppose your objective function is

```
function y = tiltcircle(x)
vx = x(:)-[4;4]; % ensure vx is in column form
y = vx'*[1;1] + sqrt(16 - vx'*vx); % complex if norm(x-[4;4])>4
```

`tiltcircle` returns complex values for $\text{norm}(x - [4 \ 4]) > 4$.



Write a constraint function that is positive on the set where $\text{norm}(x - [4 \ 4]) > 4$

```
function [c ceq] = myconstraint(x)
ceq = [];
```

```
cx = x(:) - [4;4]; % ensure x is a column vector
c = cx'*cx - 16; % negative where tiltcircle(x) is real
```

Set GlobalSearch to use only start points satisfying inequality constraints:

```
gs = GlobalSearch('StartPointsToRun','bounds-ineqs');
```

To complete the example, create a problem structure and run the solver:

```
opts = optimoptions(@fmincon,'Algorithm','interior-point');
problem = createOptimProblem('fmincon',...
    'x0',[4 4],'objective',@tiltcircle,...
    'nonlcon',@myconstraint,'lb',[-10 -10],...
    'ub',[10 10],'options',opts);
rng(7,'twister'); % for reproducibility
[x,fval,exitflag,output,solutionset] = run(gs,problem)
```

GlobalSearch stopped because it analyzed all the trial points.

All 5 local solver runs converged with a positive local solver exit flag.

```
x =
```

```
    1.1716    1.1716
```

```
fval =
```

```
   -5.6530
```

```
exitflag =
```

```
    1
```

```
output =
```

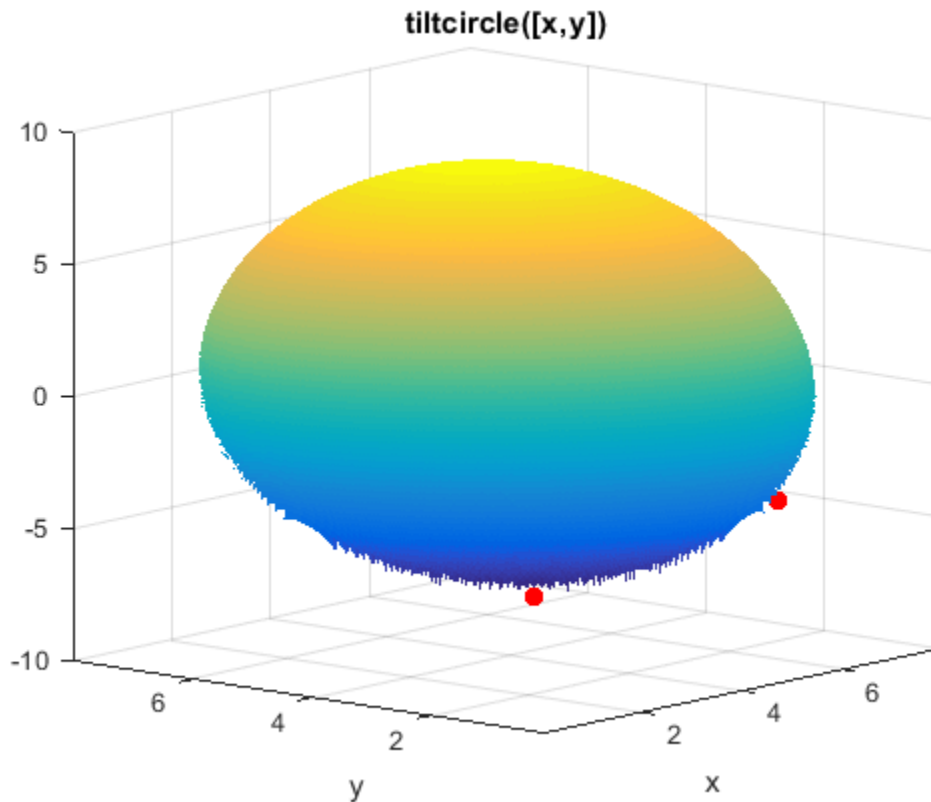
```
struct with fields:
```

```
    funcCount: 3242
    localSolverTotal: 5
    localSolverSuccess: 5
    localSolverIncomplete: 0
    localSolverNoSolution: 0
    message: 'GlobalSearch stopped because it analyzed all the trial po...'
```

```
solutionset =
```

```
1x4 GlobalOptimSolution array with properties:
```

```
    X
    Fval
    Exitflag
    Output
    X0
```



tiltcircle With Local Minima

The `tiltcircle` function has just one local minimum. Yet `GlobalSearch` (`fmincon`) stops at several points. Does this mean `fmincon` makes an error?

The reason that `fmincon` stops at several boundary points is subtle. The `tiltcircle` function has an infinite gradient on the boundary, as you can see from a one-dimensional calculation:

$$\frac{d}{dx}\sqrt{16-x^2} = \frac{-x}{\sqrt{16-x^2}} = \pm \infty \text{ at } |x| = 4.$$

So there is a huge gradient normal to the boundary. This gradient overwhelms the small additional tilt from the linear term. As far as `fmincon` can tell, boundary points are stationary points for the constrained problem.

This behavior can arise whenever you have a function that has a square root.

See Also

More About

- “Find Global or Multiple Local Minima” on page 4-57
- “Isolated Global Minimum” on page 4-86

MultiStart Using lsqcurvefit or lsqnonlin

This example shows how to fit a function to data using `lsqcurvefit` together with `MultiStart`. The end of the example shows the same solution using `lsqnonlin`.

Many fitting problems have multiple local solutions. `MultiStart` can help find the global solution, meaning the best fit. This example first uses `lsqcurvefit` because of its convenient syntax.

The model is

$$y = a + bx_1 \sin(cx_2 + d),$$

where the input data is $x = (x_1, x_2)$, and the parameters $a, b, c,$ and d are the unknown model coefficients.

Step 1. Create the objective function.

Write an anonymous function that takes a data matrix `xdata` with N rows and two columns, and returns a response vector with N rows. The function also takes a coefficient matrix `p`, corresponding to the coefficient vector (a, b, c, d) .

```
fitfcn = @(p,xdata)p(1) + p(2)*xdata(:,1).*sin(p(3)*xdata(:,2)+p(4));
```

Step 2. Create the training data.

Create 200 data points and responses. Use the values $a = -3, b = 1/4, c = 1/2, d = 1$. Include random noise in the response.

```
rng default % For reproducibility
N = 200; % Number of data points
preal = [-3,1/4,1/2,1]; % Real coefficients

xdata = 5*rand(N,2); % Data points
ydata = fitfcn(preal,xdata) + 0.1*randn(N,1); % Response data with noise
```

Step 3. Set bounds and initial point.

Set bounds for `lsqcurvefit`. There is no reason for d to exceed π in absolute value, because the sine function takes values in its full range over any interval of width 2π . Assume that the coefficient c must be smaller than 20 in absolute value, because allowing a high frequency can cause unstable responses or inaccurate convergence.

```
lb = [-Inf,-Inf,-20,-pi];
ub = [Inf,Inf,20,pi];
```

Set the initial point arbitrarily to $(5,5,5,0)$.

```
p0 = 5*ones(1,4); % Arbitrary initial point
p0(4) = 0; % Ensure the initial point satisfies the bounds
```

Step 4. Find the best local fit.

Fit the parameters to the data, starting at `p0`.

```
[xfitted,errorfitted] = lsqcurvefit(fitfcn,p0,xdata,ydata,lb,ub)
```

Local minimum possible.

lsqcurvefit stopped because the final change in the sum of squares relative to its initial value is less than the value of the function tolerance.

xfitted = 1×4

```
-2.6149   -0.0238    6.0191   -1.6998
```

errorfitted = 28.2524

lsqcurvefit finds a local solution that is not particularly close to the model parameter values (-3,1/4,1/2,1).

Step 5. Set up the problem for MultiStart.

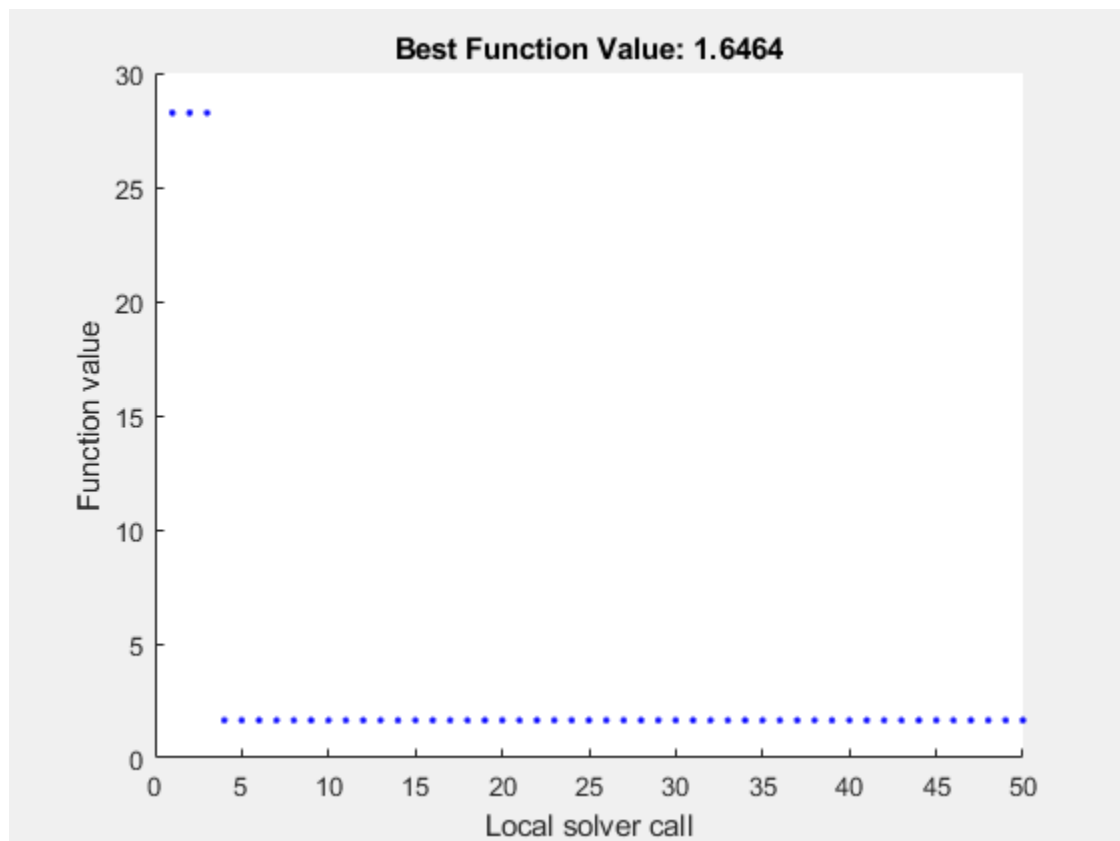
Create a problem structure so MultiStart can solve the same problem.

```
problem = createOptimProblem('lsqcurvefit','x0',p0,'objective',fitfcn,...
    'lb',lb,'ub',ub,'xdata',xdata,'ydata',ydata);
```

Step 6. Find a global solution.

Solve the fitting problem using MultiStart with 50 iterations. Plot the smallest error as the number of MultiStart iterations.

```
ms = MultiStart('PlotFcns',@gsplotbestf);
[xmulti,errormulti] = run(ms,problem,50)
```



MultiStart completed the runs from all start points.

All 50 local solver runs converged with a positive local solver exit flag.

```
xmulti = 1×4
```

```
    -2.9852    -0.2472    -0.4968    -1.0438
```

```
errormulti = 1.6464
```

MultiStart finds a global solution near the parameter values (-3,-1/4,-1/2,-1). (This is equivalent to a solution near `preal = (-3,1/4,1/2,1)`, because changing the sign of all the coefficients except the first gives the same numerical values of `fitfcn`.) The norm of the residual error decreases from about 28 to about 1.6, a decrease of more than a factor of 10.

Formulate Problem for lsqnonlin

For an alternative approach, use `lsqnonlin` as the fitting function. In this case, use the difference between predicted values and actual data values as the objective function.

```
fitfcn2 = @(p)fitfcn(p,xdata)-ydata;
[xlsqnonlin,errorlsqnonlin] = lsqnonlin(fitfcn2,p0,lb,ub)
```

Local minimum possible.

`lsqnonlin` stopped because the final change in the sum of squares relative to its initial value is less than the value of the function tolerance.

```
xlsqnonlin = 1×4
```

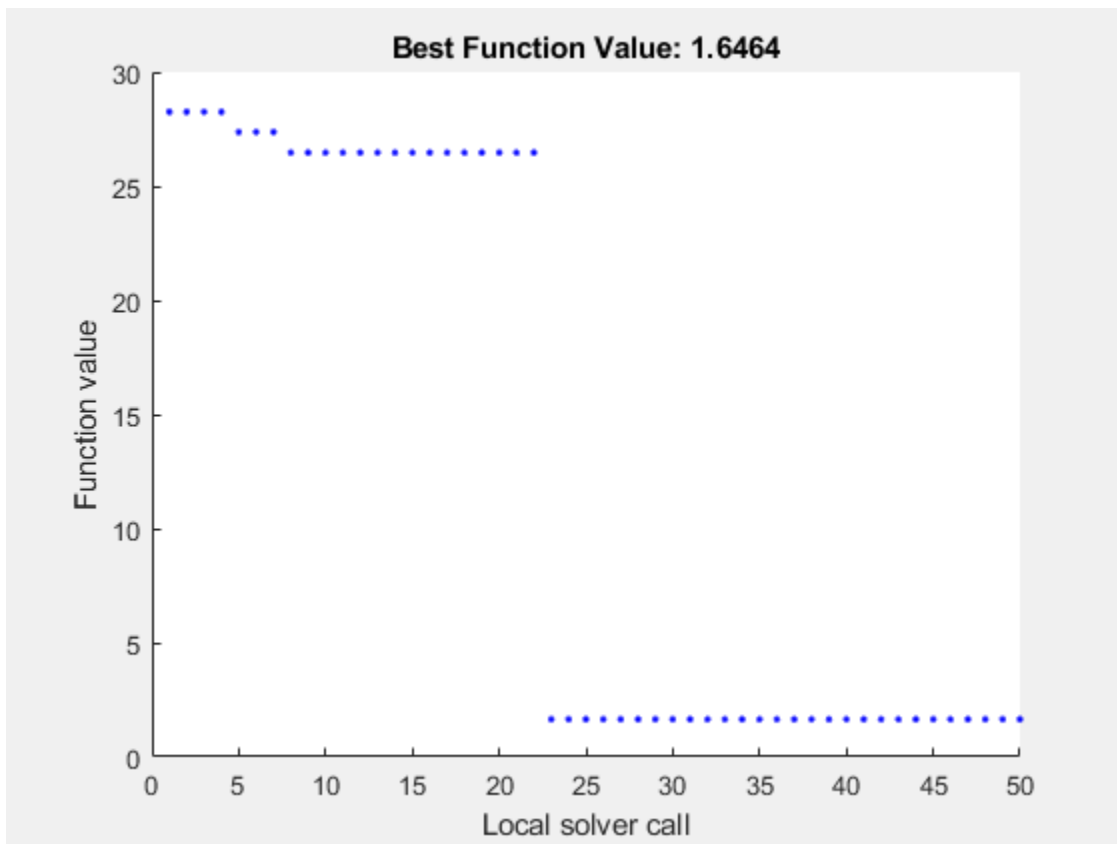
```
    -2.6149    -0.0238     6.0191    -1.6998
```

```
errorlsqnonlin = 28.2524
```

Starting from the same initial point `p0`, `lsqnonlin` finds the same relatively poor solution as `lsqcurvefit`.

Run MultiStart using `lsqnonlin` as the local solver.

```
problem2 = createOptimProblem('lsqnonlin','x0',p0,'objective',fitfcn2,...
    'lb',lb,'ub',ub);
[xmultinonlin,errormultinonlin] = run(ms,problem2,50)
```



MultiStart completed the runs from all start points.

All 50 local solver runs converged with a positive local solver exit flag.

```
xmultinonlin = 1x4
```

```
-2.9852 -0.2472 -0.4968 -1.0438
```

```
errormultinonlin = 1.6464
```

Again, MultiStart finds a much better solution than the local solver alone.

See Also

More About

- “Visualize the Basins of Attraction” on page 4-24
- “Find Global or Multiple Local Minima” on page 4-57

Parallel MultiStart

In this section...

“Steps for Parallel MultiStart” on page 4-83

“Speedup with Parallel Computing” on page 4-84

Steps for Parallel MultiStart

If you have a multicore processor or access to a processor network, you can use Parallel Computing Toolbox™ functions with `MultiStart`. This example shows how to find multiple minima in parallel for a problem, using a processor with two cores. The problem is the same as in “Find Global or Multiple Local Minima” on page 4-57.

- 1 Write a function file to compute the objective:

```
function f = sawtoothxy(x,y)
[t r] = cart2pol(x,y); % change to polar coordinates
h = cos(2*t - 1/2)/2 + cos(t) + 2;
g = (sin(r) - sin(2*r)/2 + sin(3*r)/3 - sin(4*r)/4 + 4) ...
    .*r.^2./(r+1);
f = g.*h;
end
```

- 2 Create the problem structure:

```
problem = createOptimProblem('fminunc',...
    'objective',@(x)sawtoothxy(x(1),x(2)),...
    'x0',[100,-50],'options',...
    optimoptions(@fminunc,'Algorithm','quasi-newton'));
```

- 3 Validate the problem structure by running it:

```
[x,fval] = fminunc(problem)
```

```
x =
    8.4420 -110.2602
```

```
fval =
    435.2573
```

- 4 Create a `MultiStart` object, and set the object to use parallel processing and iterative display:

```
ms = MultiStart('UseParallel',true,'Display','iter');
```

- 5 Set up parallel processing:

```
parpool
```

```
Starting parpool using the 'local' profile ... connected to 4 workers.
```

```
ans =
```

```
Pool with properties:
```

```
    Connected: true
    NumWorkers: 4
    Cluster: local
    AttachedFiles: {}
```

```
IdleTimeout: 30 minute(s) (30 minutes remaining)
SpmEnabled: true
```

6 Run the problem on 50 start points:

```
[x,fval,exitflag,output,manymins] = run(ms,problem,50);
Running the local solvers in parallel.
```

Run Index	Local exitflag	Local f(x)	Local # iter	Local F-count	First-order optimality
17	2	3953	4	21	0.1626
16	0	1331	45	201	65.02
34	0	7271	54	201	520.9
33	2	8249	4	18	2.968
... Many iterations omitted ...					
47	2	2740	5	21	0.0422
35	0	8501	48	201	424.8
50	0	1225	40	201	21.89

MultiStart completed some of the runs from the start points.

17 out of 50 local solver runs converged with a positive local solver exit flag.

Notice that the run indexes look random. Parallel MultiStart runs its start points in an unpredictable order.

Notice that MultiStart confirms parallel processing in the first line of output, which states: "Running the local solvers in parallel."

7 When finished, shut down the parallel environment:

```
delete(gcp)
Parallel pool using the 'local' profile is shutting down.
```

For an example of how to obtain better solutions to this problem, see "Example: Searching for a Better Solution" on page 4-46. You can use parallel processing along with the techniques described in that example.

Speedup with Parallel Computing

The results of MultiStart runs are stochastic. The timing of runs is stochastic, too. Nevertheless, some clear trends are apparent in the following table. The data for the table came from one run at each number of start points, on a machine with two cores.

Start Points	Parallel Seconds	Serial Seconds
50	3.6	3.4
100	4.9	5.7
200	8.3	10
500	16	23
1000	31	46

Parallel computing can be slower than serial when you use only a few start points. As the number of start points increases, parallel computing becomes increasingly more efficient than serial.

There are many factors that affect speedup (or slowdown) with parallel processing. For more information, see “Improving Performance with Parallel Computing”.

See Also

More About

- “Find Global or Multiple Local Minima” on page 4-57
- “Isolated Global Minimum” on page 4-86

Isolated Global Minimum

In this section...

“Difficult-To-Locate Global Minimum” on page 4-86

“Default Settings Cannot Find the Global Minimum — Add Bounds” on page 4-87

“GlobalSearch with Bounds and More Start Points” on page 4-88

“MultiStart with Bounds and Many Start Points” on page 4-88

“MultiStart Without Bounds, Widely Dispersed Start Points” on page 4-89

“MultiStart with a Regular Grid of Start Points” on page 4-89

“MultiStart with Regular Grid and Promising Start Points” on page 4-90

Difficult-To-Locate Global Minimum

Finding a start point in the basin of attraction of the global minimum can be difficult when the basin is small or when you are unsure of the location of the minimum. To solve this type of problem you can:

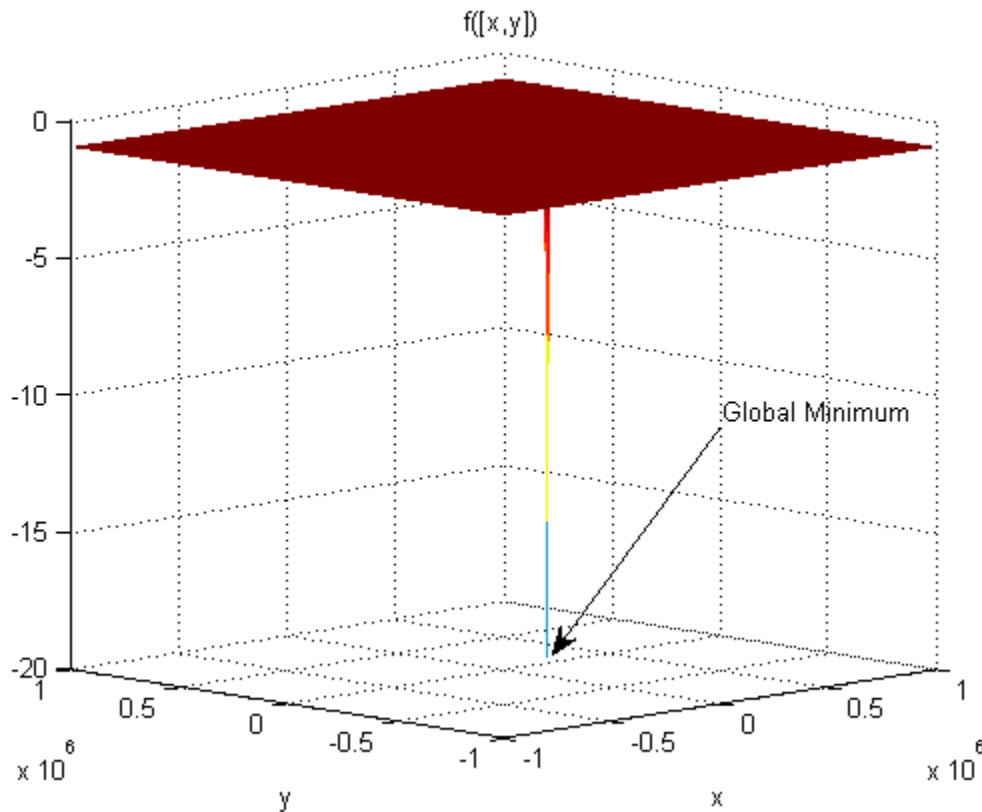
- Add sensible bounds
- Take a huge number of random start points
- Make a methodical grid of start points
- For an unconstrained problem, take widely dispersed random start points

This example shows these methods and some variants.

The function $-sech(x)$ is nearly 0 for all $|x| > 5$, and $-sech(0) = -1$. The example is a two-dimensional version of the sech function, with one minimum at $[1, 1]$, the other at $[1e5, -1e5]$:

$$f(x,y) = -10sech(|x - (1,1)|) - 20sech(.0003(|x - (1e5,-1e5)|) - 1.$$

f has a global minimum of -21 at $(1e5,-1e5)$, and a local minimum of -11 at $(1,1)$.



The minimum at $(1e5, -1e5)$ shows as a narrow spike. The minimum at $(1,1)$ does not show since it is too narrow.

The following sections show various methods of searching for the global minimum. Some of the methods are not successful on this problem. Nevertheless, you might find each method useful for different problems.

Default Settings Cannot Find the Global Minimum — Add Bounds

`GlobalSearch` and `MultiStart` cannot find the global minimum using default global options, since the default start point components are in the range $(-9999, 10001)$ for `GlobalSearch` and $(-1000, 1000)$ for `MultiStart`.

With additional bounds of $-1e6$ and $1e6$ in problem, `GlobalSearch` usually does not find the global minimum:

```
x1 = [1;1];x2 = [1e5;-1e5];
f = @(x)-10*sech(norm(x(:)-x1)) -20*sech((norm(x(:)-x2))*3e-4) -1;
opts = optimoptions(@fmincon,'Algorithm','active-set');
problem = createOptimProblem('fmincon','x0',[0,0],'objective',f,...
    'lb',[-1e6;-1e6],'ub',[1e6;1e6],'options',opts);
gs = GlobalSearch;
rng(14,'twister') % for reproducibility
[xfinal,fval] = run(gs,problem)
```

`GlobalSearch` stopped because it analyzed all the trial points.

All 32 local solver runs converged with a positive local solver exit flag.

```
xfinal =  
    1.0000    1.0000  
  
fval =  
   -11.0000
```

GlobalSearch with Bounds and More Start Points

To find the global minimum, you can search more points. This example uses 1e5 start points, and a MaxTime of 300 s:

```
gs.NumTrialPoints = 1e5;  
gs.MaxTime = 300;  
[xg,fvalg] = run(gs,problem)
```

GlobalSearch stopped because maximum time is exceeded.

GlobalSearch called the local solver 2186 times before exceeding the clock time limit (MaxTime = 300 seconds). 1943 local solver runs converged with a positive local solver exit flag.

```
xg =  
    1.0e+04 *  
    10.0000   -10.0000  
  
fvalg =  
   -21.0000
```

In this case, GlobalSearch found the global minimum.

MultiStart with Bounds and Many Start Points

Alternatively, you can search using MultiStart with many start points. This example uses 1e5 start points, and a MaxTime of 300 s:

```
ms = MultiStart(gs);  
[xm,fvalm] = run(ms,problem,1e5)
```

MultiStart stopped because maximum time was exceeded.

MultiStart called the local solver 17266 times before exceeding the clock time limit (MaxTime = 300 seconds). 17266 local solver runs converged with a positive local solver exit flag.

```
xm =  
    1.0000    1.0000  
  
fvalm =  
   -11.0000
```

In this case, MultiStart failed to find the global minimum.

MultiStart Without Bounds, Widely Dispersed Start Points

You can also use `MultiStart` to search an unbounded region to find the global minimum. Again, you need many start points to have a good chance of finding the global minimum.

The first five lines of code generate 10,000 widely dispersed random start points using the method described in “Widely Dispersed Points for Unconstrained Components” on page 4-45. `newprob` is a problem structure using the `fminunc` local solver and no bounds:

```
rng(0,'twister') % for reproducibility
u = rand(1e4,1);
u = 1./u;
u = exp(u) - exp(1);
s = rand(1e4,1)*2*pi;
stpts = [u.*cos(s),u.*sin(s)];
startpts = CustomStartPointSet(stpts);

opts = optimoptions(@fminunc,'Algorithm','quasi-newton');
newprob = createOptimProblem('fminunc','x0',[0;0],'objective',f,...
    'options',opts);
[xcust,fcust] = run(ms,newprob,startpts)
```

`MultiStart` completed the runs from all start points.

All 10000 local solver runs converged with a positive local solver exit flag.

```
xcust =
    1.0e+05 *

    1.0000
   -1.0000

fcust =
   -21.0000
```

In this case, `MultiStart` found the global minimum.

MultiStart with a Regular Grid of Start Points

You can also use a grid of start points instead of random start points. To learn how to construct a regular grid for more dimensions, or one that has small perturbations, see “Uniform Grid” on page 4-44 or “Perturbed Grid” on page 4-45.

```
xx = -1e6:1e4:1e6;
[xxx,yyy] = meshgrid(xx,xx);
z = [xxx(:),yyy(:)];
bigstart = CustomStartPointSet(z);
[xgrid,fgrid] = run(ms,newprob,bigstart)
```

`MultiStart` completed the runs from all start points.

All 10000 local solver runs converged with a positive local solver exit flag.

```
xgrid =
```

```

1.0e+004 *
    10.0000
   -10.0000

fgrid =
   -21.0000

```

In this case, MultiStart found the global minimum.

MultiStart with Regular Grid and Promising Start Points

Making a regular grid of start points, especially in high dimensions, can use an inordinate amount of memory or time. You can filter the start points to run only those with small objective function value.

To perform this filtering most efficiently, write your objective function in a vectorized fashion. For information, see “Write a Vectorized Function” on page 2-3 or “Vectorize the Objective and Constraint Functions” on page 6-83. The following function handle computes a vector of objectives based on an input matrix whose rows represent start points:

```

x1 = [1;1];x2 = [1e5;-1e5];
g = @(x) -10*sech(sqrt((x(:,1)-x1(1)).^2 + (x(:,2)-x1(2)).^2)) ...
        -20*sech(sqrt((x(:,1)-x2(1)).^2 + (x(:,2)-x2(2)).^2))-1;

```

Suppose you want to run the local solver only for points where the value is less than -2. Start with a denser grid than in “MultiStart with a Regular Grid of Start Points” on page 4-89, then filter out all the points with high function value:

```

xx = -1e6:1e3:1e6;
[xxx,yyy] = meshgrid(xx,xx);
z = [xxx(:),yyy(:)];
idx = g(z) < -2; % index of promising start points
zz = z(idx,:);
smallstartset = CustomStartPointSet(zz);
opts = optimoptions(@fminunc,'Algorithm','quasi-newton','Display','off');
newprobg = createOptimProblem('fminunc','x0',[0,0],...
    'objective',g,'options',opts);
    % row vector x0 since g expects rows
[xfew,ffew] = run(ms,newprobg,smallstartset)

```

MultiStart completed the runs from all start points.

All 2 local solver runs converged with a positive local solver exit flag.

```

xfew =
    100000    -100000

ffew =
    -21

```

In this case, MultiStart found the global minimum. There are only two start points in `smallstartset`, one of which is the global minimum.

See Also

More About

- “Parallel MultiStart” on page 4-83
- “Visualize the Basins of Attraction” on page 4-24
- “Refine Start Points” on page 4-44

Problem-Based Multiple Start

- “Minimize Nonlinear Function Using Multiple-Start Solver, Problem-Based” on page 5-2
- “Specify Start Points for MultiStart, Problem-Based” on page 5-3
- “MultiStart with lsqnonlin, Problem-Based” on page 5-6
- “Find Multiple Local Solutions Using MultiStart or GlobalSearch, Problem-Based” on page 5-9

Minimize Nonlinear Function Using Multiple-Start Solver, Problem-Based

Find a local minimum of the peaks function on the range $-5 \leq x, y \leq 5$ starting from the point $[-1, 2]$.

```
x = optimvar("x",LowerBound=-5,UpperBound=5);
y = optimvar("y",LowerBound=-5,UpperBound=5);
x0.x = -1;
x0.y = 2;
prob = optimproblem(Objective=peaks(x,y));
opts = optimoptions("fmincon",Display="none");
[sol,fval] = solve(prob,x0,Options=opts)
```

```
sol = struct with fields:
    x: -3.3867
    y: 3.6341
```

```
fval = 1.1224e-07
```

Try to find a better solution by using the GlobalSearch solver. This solver runs fmincon multiple times, which potentially yields a better solution.

```
ms = GlobalSearch;
[sol2,fval2] = solve(prob,x0,ms)
```

Solving problem using GlobalSearch.

GlobalSearch stopped because it analyzed all the trial points.

All 15 local solver runs converged with a positive local solver exit flag.

```
sol2 = struct with fields:
    x: 0.2283
    y: -1.6255
```

```
fval2 = -6.5511
```

GlobalSearch finds a solution with a better (lower) objective function value. The exit message shows that fmincon, the local solver, runs 15 times. The returned solution has an objective function value of about -6.5511, which is lower than the value at the first solution, 1.1224e-07.

See Also

GlobalSearch | MultiStart | run | solve

Related Examples

- “Global or Multiple Starting Point Search”

Specify Start Points for MultiStart, Problem-Based

When solving a problem using `MultiStart` in the problem-based approach, you can specify the start points using one or both of these techniques:

- Create a vector of `OptimizationValues` objects using the `optimvalues` function. Pass this vector as the `x0` argument to `solve`.
- Set the `MinNumStartPoints` name-value argument when you call `solve`. If `MinNumStartPoints` exceeds the number of points in `x0`, then `solve` creates extra points at random within the problem bounds.

Vector of Initial Points

For this example, create the initial points vector as a grid for the variable `x` consisting of integer values from `-10` through `10`, and for the variable `y` consisting of half-integer values from `-5/2` through `5/2`. The `optimvalues` function requires a problem, so create an optimization problem with the objective function `peaks(x,y)`.

You must specify the points for `optimvalues` so that the dimension (index) of the number of points is last. For example, to give multiple values of a scalar `t` with `n` points, specify

`[t(1) t(2) . . . t(n)]` (This is 1-by-`n`, and `n` is the last index.)

To give multiple values of a vector variable `w` of length 2, specify

`[w(1,1) w(1,2) w(1,3) ... w(1,n)]`
`[w(2,1) w(2,2) w(2,3) ... w(2,n)]` (This is 2-by-`n`, and `n` is the last index.)

This rule holds even for row vectors. In other words, you specify multiple row vectors as if each were a column vector.

To give multiple values of a matrix `A` that is 2-by-3, specify

`[A(1,1,1) A(1,2,1) A(1,3,1)]`
`[A(1,1,2) A(1,2,2) A(1,3,2)]` ... `[A(1,1,n) A(1,2,n) A(1,3,n)]`
`[A(2,1,1) A(2,2,1) A(2,3,1)]`
`[A(2,1,2) A(2,2,2) A(2,3,2)]` ... `[A(2,1,n) A(2,2,n) A(2,3,n)]` (This is 2-by-3-by-`n`, and `n` is the last index.)

In the present example, ensure that you specify the multiple values of the scalar variables `x` and `y` as row vectors, as in the scalar `t` example.

```
x = optimvar("x",LowerBound=-10,UpperBound=10);
y = optimvar("y",LowerBound=-5/2,UpperBound=5/2);
prob = optimproblem(Objective=peaks(x,y));
xval = -10:10;
yval = (-5:5)/2;
[x0x,x0y] = meshgrid(xval,yval);
% Convert x0x and x0y to row vectors for optimvalues
x0 = optimvalues(prob,x=x0x(:)',y=x0y(:)');
```

Solve the minimization problem starting from all the points in `x0`.

```
ms = MultiStart;
[sol,fval,eflag,output] = solve(prob,x0,ms)
```

Solving problem using MultiStart.

MultiStart completed the runs from all start points.

All 231 local solver runs converged with a positive local solver exit flag.

```
sol = struct with fields:
  x: 0.2283
  y: -1.6255
```

```
fval = -6.5511
```

```
eflag =
  LocalMinimumFoundAllConverged
```

```
output = struct with fields:
  funcCount: 2230
  localSolverTotal: 231
  localSolverSuccess: 231
  localSolverIncomplete: 0
  localSolverNoSolution: 0
  message: 'MultiStart completed the runs from all start points....'
  local: [1x1 struct]
  objectiveDerivative: "reverse-AD"
  constraintDerivative: "auto"
  globalSolver: 'MultiStart'
  solver: 'fmincon'
```

Random Start Points

Solve the problem again, this time using MultiStart from 25 random initial points. Set the initial point for solve to [-1,2].

```
init.x = -1;
init.y = 2;
rng default % For reproducibility
[sol2,fval2,eflag2,output2] = solve(prob,init,ms,MinNumStartPoints=25)
```

Solving problem using MultiStart.

MultiStart completed the runs from all start points.

All 25 local solver runs converged with a positive local solver exit flag.

```
sol2 = struct with fields:
  x: 0.2283
  y: -1.6255
```

```
fval2 = -6.5511
```

```
eflag2 =
  LocalMinimumFoundAllConverged
```

```
output2 = struct with fields:
  funcCount: 161
```

```
    localSolverTotal: 25
    localSolverSuccess: 25
    localSolverIncomplete: 0
    localSolverNoSolution: 0
        message: 'MultiStart completed the runs from all start points....'
            local: [1x1 struct]
    objectiveDerivative: "reverse-AD"
    constraintDerivative: "auto"
        globalSolver: 'MultiStart'
            solver: 'fmincon'
```

This time, `MultiStart` runs from 25 pseudorandom initial points. The solution is the same as before.

See Also

`MultiStart` | `solve` | `optimvalues`

Related Examples

- “Global or Multiple Starting Point Search”

MultiStart with lsqnonlin, Problem-Based

This example shows how to fit a function to data using `lsqnonlin` together with `MultiStart` in the problem-based approach.

Many fitting problems have multiple local solutions. `MultiStart` can help find the global solution, meaning the best fit.

The model is

$$y = a + bx_1 \sin(cx_2 + d),$$

where the input data is $x = (x_1, x_2)$, and the parameters a , b , c , and d are the unknown model coefficients.

Create Problem Data

Most problems involve data from measurements. For this problem, create artificial data including noise. Create 200 data points and responses. Specify the values $a = -3$, $b = 1/4$, $c = 1/2$, and $d = 1$.

```
rng default % For reproducibility
fitfcn = @(p,xdata)p(1) + p(2)*xdata(:,1).*sin(p(3)*xdata(:,2)+p(4));
N = 200; % Number of data points
preal = [-3,1/4,1/2,1]; % Real coefficients
xdata = 5*rand(N,2); % Data points
ydata = fitfcn(preal,xdata) + 0.1*randn(N,1); % Response data with noise
```

Create Optimization Variables and Problem

The optimization variables are the model coefficients. Set bounds for the variables as you create them. The variable d does not need to exceed π in absolute value, because the sine function takes values in its full range over any interval of width 2π . Assume that the coefficient c must be smaller than 20 in absolute value, because allowing a high frequency can cause unstable responses or inaccurate convergence.

```
a = optimvar("a");
b = optimvar("b");
c = optimvar("c",LowerBound=-20,UpperBound=20);
d = optimvar("d",LowerBound=-pi,UpperBound=pi);
prob = optimproblem;
```

Create Objective Function

Create the objective function as the sum of squared differences between the model response and the data.

```
resp = a + b*xdata(:,1).*sin(c*xdata(:,2) + d);
prob.Objective = sum((resp - ydata).^2);
```

Create Initial Point and Solve Problem

The initial point is a structure with values for the coefficients. Arbitrarily set the initial point to `[5,5,5,0]`.

```
x0.a = 5;
x0.b = 5;
```

```
x0.c = 5;  
x0.d = 0;  
[sol,fval] = solve(prob,x0)
```

Solving problem using lsqnonlin.

Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to its initial value is less than the value of the function tolerance.

```
sol = struct with fields:  
  a: -2.6149  
  b: -0.0238  
  c: 6.0191  
  d: -1.6998
```

```
fval = 28.2524
```

lsqnonlin finds a local solution that is not particularly close to the model parameter values (-3,1/4,1/2,1).

Search for Improved Solution Using MultiStart

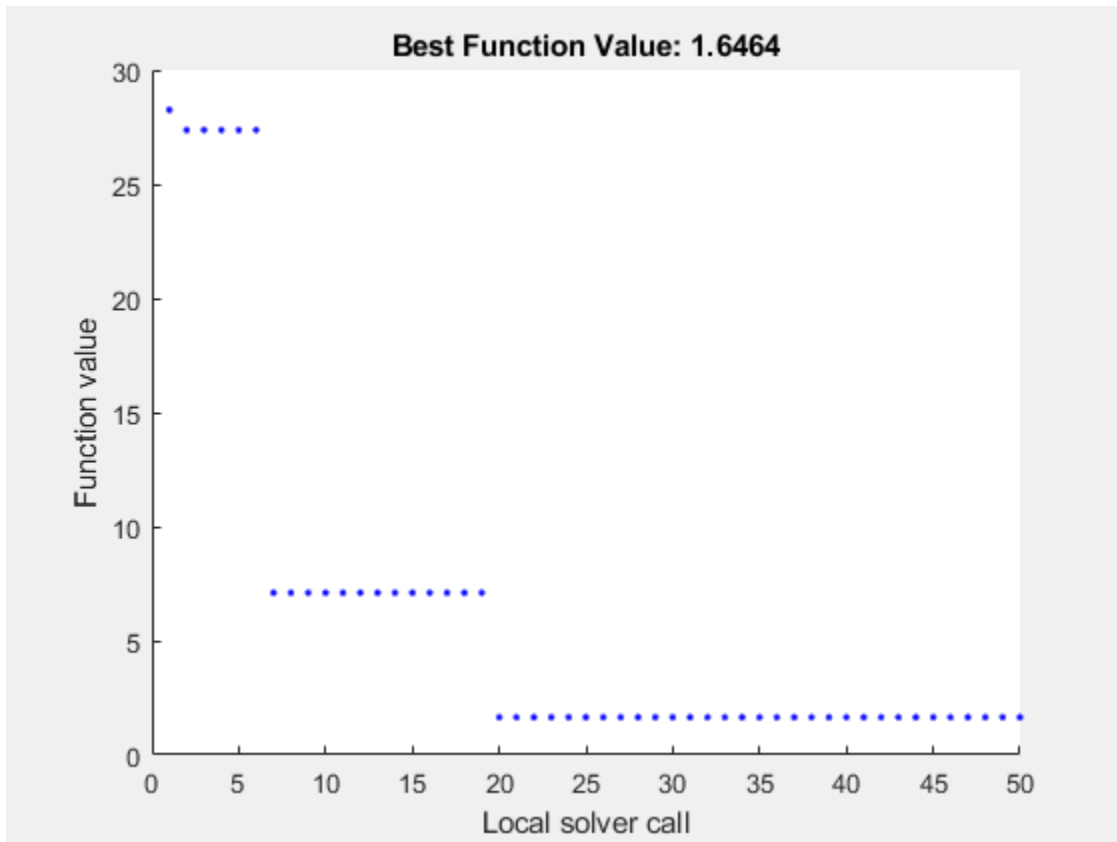
To search for a better solution, create a MultiStart object. Plot the best function value as the solver iterates.

```
ms = MultiStart(PlotFcns=@gsplotbestf);
```

Call solve again, this time using ms. Specify 50 start points for MultiStart.

```
rng default % For reproducibility  
[sol2,fval2] = solve(prob,x0,ms,MinNumStartPoints=50)
```

Solving problem using MultiStart.



MultiStart completed the runs from all start points.

All 50 local solver runs converged with a positive local solver exit flag.

```
sol2 = struct with fields:
  a: -2.9852
  b: -0.2472
  c: -0.4968
  d: -1.0438
```

```
fval2 = 1.6464
```

MultiStart finds a global solution near the parameter values $(-3, -1/4, -1/2, -1)$. This solution is equivalent to a solution near the true parameter values $(-3, 1/4, 1/2, 1)$, because changing the sign of all coefficients except the first gives the same numerical values of the error. The norm of the residual error decreases from about 28 to about 1.6, a decrease of more than a factor of 10.

See Also

MultiStart | solve | lsqnonlin

Related Examples

- “Write Objective Function for Problem-Based Least Squares”
- “Global or Multiple Starting Point Search”

Find Multiple Local Solutions Using MultiStart or GlobalSearch, Problem-Based

In solutions to problems that use the `MultiStart` or `GlobalSearch` function with the problem-based approach, the output structure contains a field named `local` that has information about the local solutions. For example, find the local solutions to the `rastriginsfcn` function of two variables, which is available when you run this example, by using `MultiStart`.

```
x = optimvar("x",LowerBound=-50,UpperBound=50);
y = optimvar("y",LowerBound=-50,UpperBound=50);
fun = rastriginsfcn([x,y]);
prob = optimproblem(Objective=fun);
ms = MultiStart;
x0.x = -30;
x0.y = 20;
rng default % For reproducibility
[sol,fval,exitflag,output] = solve(prob,x0,ms,MinNumStartPoints=50);
```

Solving problem using `MultiStart`.

`MultiStart` completed the runs from all start points.

All 50 local solver runs converged with a positive local solver exit flag.

```
disp(sol)
```

```
    x: 6.8842e-10
    y: 1.0077e-09
```

```
disp(fval)
```

```
    0
```

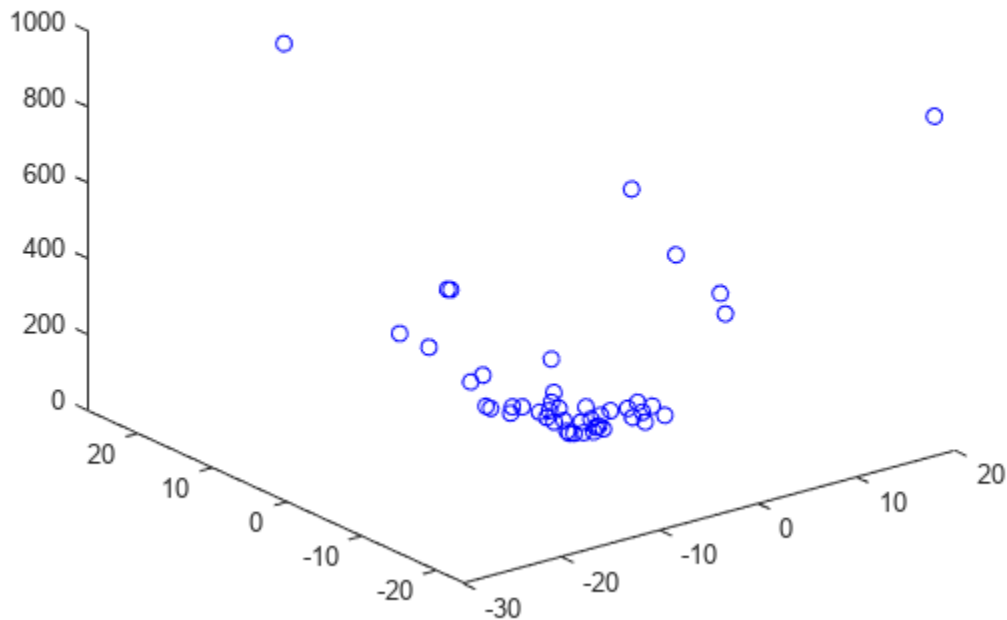
How many local solutions does `MultiStart` find?

```
multisols = output.local.sol;
N = numel(multisols)
```

```
N = 46
```

Plot the values.

```
plot3(multisols.x,multisols.y,multisols.Objective,'bo')
```



MultiStart starts from 50 initial points, but finds only 46 solutions. MultiStart reports that all runs converged. Therefore, some solutions have multiple initial points leading to those solutions. Find the `x0` values that list multiple initial points.

```
myx0 = output.local.x0;
sx = zeros(size(myx0));
for i = 1:length(sx)
    sx(i) = numel(myx0{i});
end
mults = find(sx >= 2)
```

```
mults = 1x4
```

```
    14    17    25    36
```

Determine whether `fmincon`, starting from two initial points in `mults(1)`, ends at the same solution.

```
pts = myx0(mults(1));
r = pts{1}.x;
t01.x = r(1);
s = pts{1}.y;
t01.y = s(1);
disp(t01)
```

```
    x: -30
    y: 20
```



```
opts = optimoptions("fmincon",Display="none");
sol1 = solve(prob,t01,options=opts)

sol1 = struct with fields:
    x: -1.9899
    y: -2.9849

t02.x = r(2);
t02.y = s(2);
disp(t02)

    x: 34.9129
    y: -24.5718

sol2 = solve(prob,t02,options=opts)

sol2 = struct with fields:
    x: -1.9899
    y: -2.9849
```

Even though the starting points `t01` and `t02` are far apart, the solutions `sol1` and `sol2` are identical to display precision.

See Also

[MultiStart](#) | [solve](#) | [GlobalSearch](#)

Related Examples

- “Global or Multiple Starting Point Search”

Using Direct Search

- “What Is Direct Search?” on page 6-2
- “Optimize Using the GPS Algorithm” on page 6-3
- “Coding and Minimizing an Objective Function Using Pattern Search” on page 6-10
- “Constrained Minimization Using Pattern Search, Solver-Based” on page 6-14
- “Effects of Pattern Search Options” on page 6-18
- “Pattern Search Terminology” on page 6-24
- “How Pattern Search Polling Works” on page 6-27
- “Nonuniform Pattern Search (NUPS) Algorithm” on page 6-35
- “Searching and Polling” on page 6-37
- “Setting Solver Tolerances” on page 6-41
- “Search and Poll” on page 6-42
- “Nonlinear Constraint Solver Algorithm for Pattern Search” on page 6-46
- “Custom Plot Function” on page 6-48
- “Pattern Search Climbs Mount Washington” on page 6-52
- “Set Options” on page 6-57
- “Polling Types” on page 6-59
- “Set Mesh Options” on page 6-66
- “Constrained Minimization Using patternsearch and Optimize Live Editor Task” on page 6-71
- “Use Cache” on page 6-80
- “Vectorize the Objective and Constraint Functions” on page 6-83
- “Optimize ODEs in Parallel” on page 6-87
- “Optimization of Stochastic Objective Function” on page 6-95
- “Explore patternsearch Algorithms” on page 6-103
- “Explore patternsearch Algorithms in Optimize Live Editor Task” on page 6-107

What Is Direct Search?

Direct search is a method for solving optimization problems that does not require any information about the gradient of the objective function. Unlike more traditional optimization methods that use information about the gradient or higher derivatives to search for an optimal point, a direct search algorithm searches a set of points around the current point, looking for one where the value of the objective function is lower than the value at the current point. You can use direct search to solve problems for which the objective function is not differentiable, or is not even continuous.

Global Optimization Toolbox functions include three direct search algorithms called the generalized pattern search (GPS) algorithm, the generating set search (GSS) algorithm, and the mesh adaptive search (MADS) algorithm. All are *pattern search* algorithms that compute a sequence of points that approach an optimal point. At each step, the algorithm searches a set of points, called a *mesh*, around the *current point*—the point computed at the previous step of the algorithm. The mesh is formed by adding the current point to a scalar multiple of a set of vectors called a *pattern*. If the pattern search algorithm finds a point in the mesh that improves the objective function at the current point, the new point becomes the current point at the next step of the algorithm.

The GPS algorithm uses fixed direction vectors. The GSS algorithm is identical to the GPS algorithm, except when there are linear constraints, and when the current point is near a linear constraint boundary. The MADS algorithm uses a random selection of vectors to define the mesh. For details, see “Patterns” on page 6-24.

See Also

More About

- “Optimize Using the GPS Algorithm” on page 6-3
- “Pattern Search Terminology” on page 6-24
- “How Pattern Search Polling Works” on page 6-27

Optimize Using the GPS Algorithm

This example shows how to solve an optimization problem using the GPS algorithm, which is the default for the `patternsearch` solver. The example uses the Optimize Live Editor task to complete the optimization using a visual approach.

Objective Function

This example uses the objective function `ps_example`, which is included when you run this example. View the code for the function.

```
type ps_example

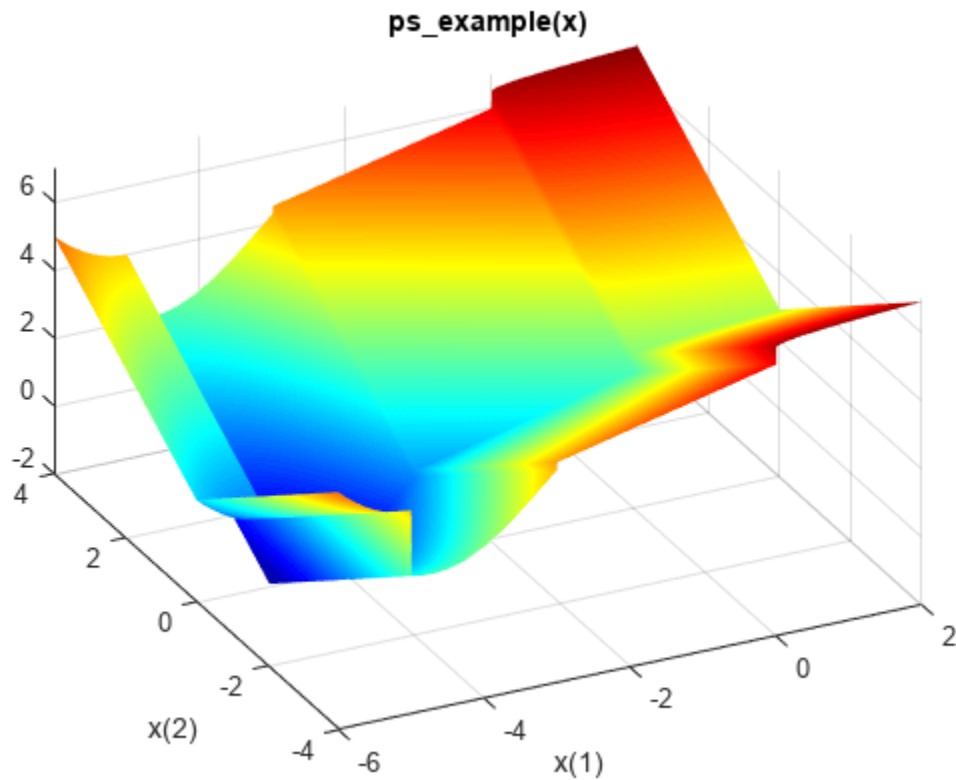
function f = ps_example(x)
%PS_EXAMPLE objective function for patternsearch.

% Copyright 2003-2021 The MathWorks, Inc.

f = zeros(1,size(x,1));
for i = 1:size(x,1)
    if x(i,1) < -5
        f(i) = (x(i,1)+5)^2 + abs(x(i,2));
    elseif x(i,1) < -3
        f(i) = -2*sin(x(i,1)) + abs(x(i,2));
    elseif x(i,1) < 0
        f(i) = 0.5*x(i,1) + 2 + abs(x(i,2));
    elseif x(i,1) >= 0
        f(i) = .3*sqrt(x(i,1)) + 5/2 +abs(x(i,2));
    end
end
```

Plot the function.

```
fsurf(@(x,y)reshape(ps_example([x(:),y(:)]),size(x)),...
    [-6 2 -4 4],'LineStyle','none','MeshDensity',300)
colormap 'jet'
view(-26,43)
xlabel('x(1)')
ylabel('x(2)')
title('ps\_example(x)')
```



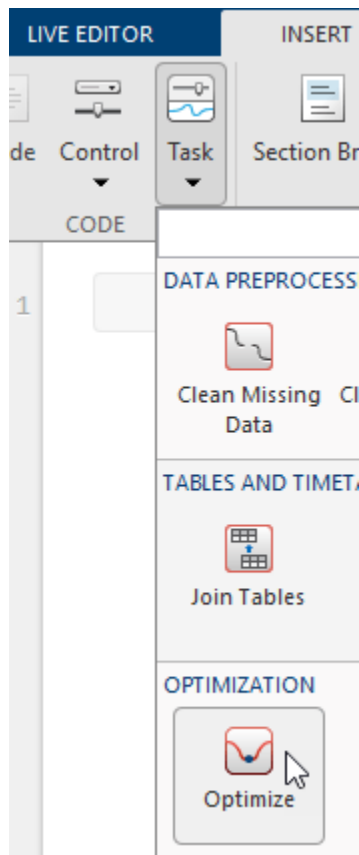
Find the Minimum of the Function

To find the minimum of `ps_example` using the Optimize Live Editor task, complete the following steps.

- Create a new live script by clicking the **New Live Script** button in the **File** section on the **Home** tab.





- Insert an Optimize Live Editor task. Click the **Insert** tab and then, in the **Code** section, select **Task > Optimize**.



Optimize

Solve an optimization problem or system of equations

Select approach

 <p>Problem-based (recommended)</p> <ul style="list-style-type: none">• Easier to define problem• Represents problem inputs symbolically• Built-in automatic differentiation	 <p>Solver-based</p> <ul style="list-style-type: none">• Start with a solver• Represents inputs as matrices/functions• Allows specialized solution methods
--	--


- Click the **Solver-based** task.


Optimize


Minimize a function with or without constraints


▼ Specify problem type


Objective


Linear


Quadratic


Least squares


Nonlinear


Nonsmooth

Select an objective type to see example functions

Unconstrained

Lower bounds

Upper bounds

Linear inequality

Constraints

Linear equality

Second-order cone

Nonlinear

Integer

Select constraint types to see example formulas

Solver fmincon - Constrained nonlinear minimization (recommended) ?

▼ Select problem data

Objective function From file ▼ Browse... New... ?

Initial point (x0) select ▼

► Specify solver options

▼ Display progress

Text display Final output ▼

Plot

Current point

Evaluation count

Objective value and feasibility

Objective value

Max constraint violation

Step size

Optimality measure

- For use in entering problem data, insert a new section by clicking the **Section Break** button on the **Insert** tab. New sections appear above and below the task.
- In the new section above the task, enter the following code to define the initial point and objective function.

```
x0 = [2.1 1.7];
fun = @ps_example;
```

- To place these variables into the workspace, run the section by pressing **Ctrl + Enter**.
- In the **Specify problem type** section of the task, click the **Objective > Nonsmooth** button.
- Ensure that the selected solver is `patternsearch`.
- In the **Select problem data** section of the task, select **Objective function > Function handle** and then choose `fun`.
- Select **Initial point (x0) > x0**.

- In the **Display progress** section of the task, select the **Best value** and **Mesh size** plots.

Solver patternsearch - Pattern search (recommended)

Select problem data

Objective function Function handle ▼ fun ▼ ?

Initial point (x0) x0 ▼

Specify solver options

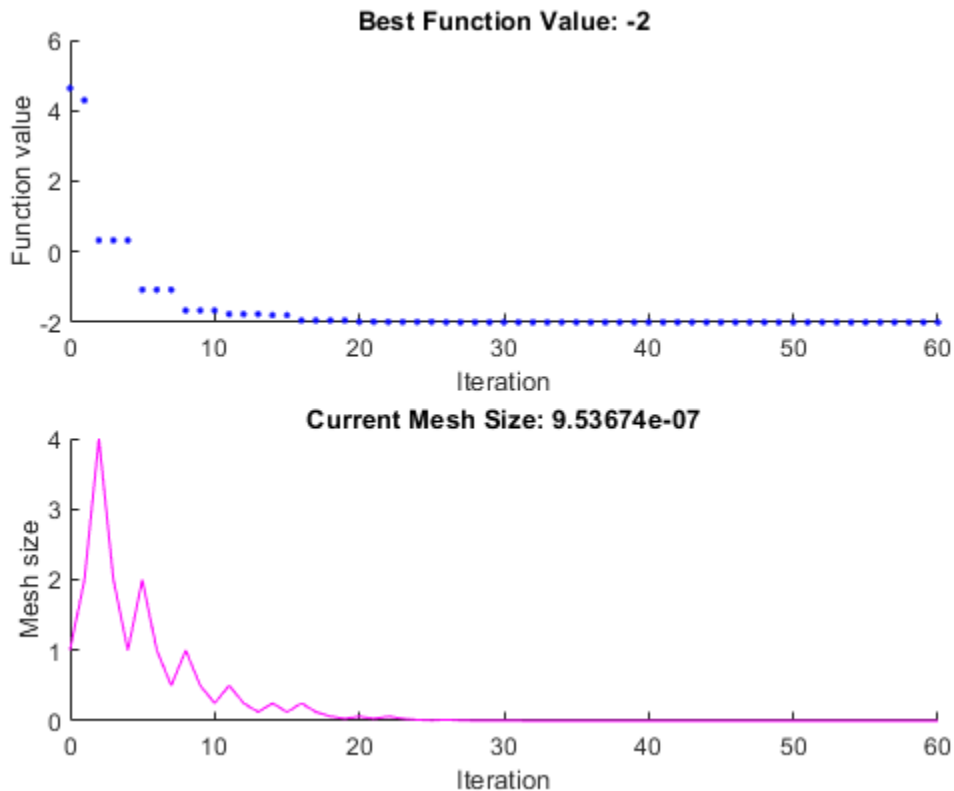
Display progress

Text display Final output ▼

Plot Best value Mesh size Function count Best point

Max constraint

- To run the solver, click the options button **:** at the top right of the task window, and select **Run Section**. The plots appear in a separate figure window and in the task output area.



The upper plot shows the objective function value of the best point at each iteration. Typically, the objective function values improve rapidly at the early iterations and then level off as they approach the optimal value.

The lower plot shows the mesh size at each iteration. The mesh size increases after each successful iteration and decreases after each unsuccessful iteration. For details, see “How Pattern Search Polling Works” on page 6-27.

The optimization stops because the mesh size becomes smaller than the mesh size tolerance value, defined by the `MeshTolerance` option. The minimum function value is approximately -2.

To see the solution and objective function value, look at the top of the task.

Optimize

```
solution, objectiveValue = Minimize fun using patternsearch solver
```

The `Optimize` task puts the variables `solution` and `objectiveValue` in the workspace. View these values by placing a new section below the task, and include this code.

```
disp(solution)
disp(objectiveValue)
```

Run the section by pressing **Ctrl+Enter**.

The `Optimize` Live Editor task appears next in its final state.

Optimize

```
solution, objectiveValue = Minimize fun using patternsearch solver
```

▼ Specify problem type

Objective

Linear Quadratic Least squares Nonlinear Nonsmooth

Examples: $f(x) = |x|$, $f(x) = \max(x)$, piecewise,...

Constraints

Unconstrained Lower bounds Upper bounds Linear inequality

Linear equality Second-order cone Nonlinear Integer

Select constraint types to see example formulas

Solver patternsearch - Pattern search (recommended) ?

▼ Select problem data

Objective function Function handle fun ?

Initial point (x0) x0

► Specify solver options

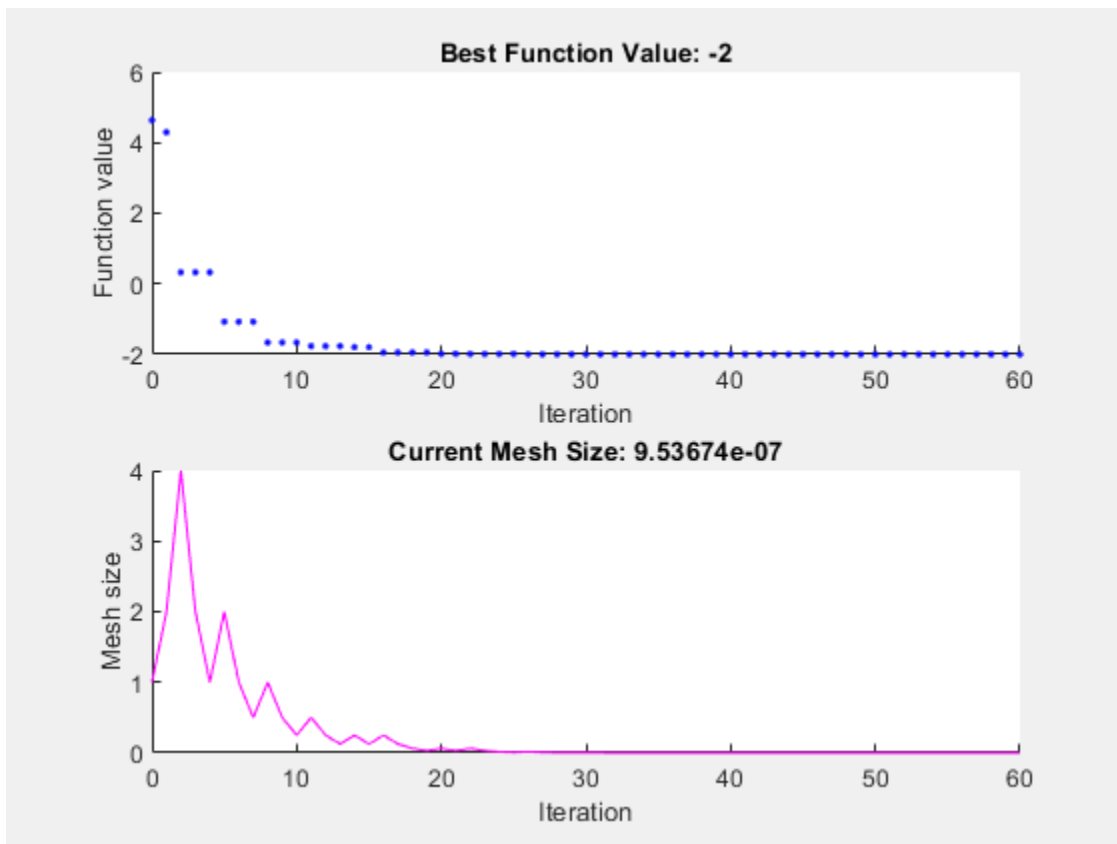
▼ Display progress

Text display Final output

Plot Best value Mesh size Evaluation count Best point

Max constraint violation

Optimization terminated: mesh size less than options.MeshTolerance.



```
disp(solution)
```

```
-4.7124 -0.0000
```

```
disp(objectiveValue)
```

```
-2.0000
```

See Also

[patternsearch](#) | [Optimize](#)

More About

- “Constrained Minimization Using patternsearch and Optimize Live Editor Task” on page 6-71
- “How Pattern Search Polling Works” on page 6-27
- “Add Interactive Tasks to a Live Script”

Coding and Minimizing an Objective Function Using Pattern Search

This example shows how to create and minimize an objective function using pattern search.

Objective Function

For this problem, the objective function to minimize is a simple function of a 2-D variable x .

```
simple_objective(x) = (4 - 2.1*x(1)^2 + x(1)^4/3)*x(1)^2 + x(1)*x(2) + (-4 + 4*x(2)^2)*x(2)^2;
```

This function is known as "cam," as described in L.C.W. Dixon and G.P. Szego [1].

Code the Objective Function

Create a MATLAB® file named `simple_objective.m` containing the following code:

```
type simple_objective

function y = simple_objective(x)
%SIMPLE_OBJECTIVE Objective function for PATTERNSEARCH solver

% Copyright 2004 The MathWorks, Inc.

x1 = x(1);
x2 = x(2);
y = (4-2.1.*x1.^2+x1.^4./3).*x1.^2+x1.*x2+(-4+4.*x2.^2).*x2.^2;
```

Solvers such as `patternsearch` accept a single input x , where x has as many elements as the number of variables in the problem. The objective function computes the scalar value of the objective function and returns it in its single output argument y .

Minimize Using `patternsearch`

Specify the objective function as a function handle.

```
ObjectiveFunction = @simple_objective;
```

Specify an initial point for the solver.

```
x0 = [0.5 0.5]; % Starting point
```

Call the solver, requesting the optimal point x and the function value at the optimal point `fval`.

```
[x,fval] = patternsearch(ObjectiveFunction,x0)
```

```
Optimization terminated: mesh size less than options.MeshTolerance.
```

```
x = 1x2
```

```
    -0.0898    0.7127
```

```
fval = -1.0316
```

Minimize Using Additional Arguments

Sometimes your objective function has extra arguments that act as constants during the optimization. For example, in `simple_objective`, you might want to specify the constants 4, 2.1, and 4 as variable parameters to create a family of objective functions.

Rewrite `simple_objective` to take three additional parameters (`p1`, `p2`, and `p3`) that act as constants during the optimization (they are not varied as part of the minimization). To implement the objective function calculation, the MATLAB file `parameterized_objective.m` contains the following code:

```
type parameterized_objective

function y = parameterized_objective(x,p1,p2,p3)
%PARAMETERIZED_OBJECTIVE Objective function for PATTERNSEARCH solver

% Copyright 2004 The MathWorks, Inc.

x1 = x(1);
x2 = x(2);
y = (p1-p2.*x1.^2+x1.^4./3).*x1.^2+x1.*x2+(-p3+p3.*x2.^2).*x2.^2;
```

`patternsearch` calls the objective function with just one argument `x`, but the parameterized objective function has four arguments: `x`, `p1`, `p2`, and `p3`. Use an anonymous function to capture the values of the additional arguments `p1`, `p2`, and `p3`. Create a function handle `ObjectiveFunction` to an anonymous function that takes one input `x`, but calls `parameterized_objective` with `x`, `p1`, `p2`, and `p3`. When you create the function handle `ObjectiveFunction`, the variables `p1`, `p2`, and `p3` have values that are stored in the anonymous function. For details, see “Passing Extra Parameters”.

```
p1 = 4; p2 = 2.1; p3 = 4; % Define constant values
ObjectiveFunction = @(x) parameterized_objective(x,p1,p2,p3);
[x,fval] = patternsearch(ObjectiveFunction,x0)

Optimization terminated: mesh size less than options.MeshTolerance.

x = 1x2
    -0.0898    0.7127

fval = -1.0316
```

Vectorize the Objective Function

By default, `patternsearch` passes in one point at a time to the objective function. Sometimes, you can speed the solver by *vectorizing* the objective function to take a set of points and return a set of function values.

For the solver to evaluate a set of five points in one call to the objective function, for example, the solver calls the objective on a matrix of size 5-by-2 (where 2 is the number of variables). For details, see “Vectorize the Objective and Constraint Functions” on page 6-83.

To vectorize `parameterized_objective`, use the following code:

```
type vectorized_objective

function y = vectorized_objective(x,p1,p2,p3)
%VECTORIZED_OBJECTIVE Objective function for PATTERNSEARCH solver
```

```
% Copyright 2004-2018 The MathWorks, Inc.  
x1 = x(:,1); % First column of x  
x2 = x(:,2);  
y = (p1 - p2.*x1.^2 + x1.^4./3).*x1.^2 + x1.*x2 + (-p3 + p3.*x2.^2).*x2.^2;
```

This vectorized version of the objective function takes a matrix x with an arbitrary number of points (the rows of x) and returns a column vector y whose length is the number of rows of x .

To take advantage of the vectorized objective function, set the `UseVectorized` option to `true` and the `UseCompletePoll` option to `true`. `patternsearch` requires both of these options to compute in a vectorized manner.

```
options = optimoptions(@patternsearch,'UseVectorized',true,'UseCompletePoll',true);
```

Specify the objective function and call `patternsearch`, including the `options` argument. Use `tic/toc` to evaluate the solution time.

```
ObjectiveFunction = @(x) vectorized_objective(x,4,2.1,4);  
tic  
[x,fval] = patternsearch(ObjectiveFunction,x0,[],[],[],[],[],[],[],options)
```

```
Optimization terminated: mesh size less than options.MeshTolerance.
```

```
x = 1x2  
    -0.0898    0.7127
```

```
fval = -1.0316
```

```
toc
```

```
Elapsed time is 0.027503 seconds.
```

Evaluate the nonvectorized solution time for comparison.

```
tic  
[x,fval] = patternsearch(ObjectiveFunction,x0)
```

```
Optimization terminated: mesh size less than options.MeshTolerance.
```

```
x = 1x2  
    -0.0898    0.7127
```

```
fval = -1.0316
```

```
toc
```

```
Elapsed time is 0.027502 seconds.
```

In this case, the vectorization does not have a significant impact on the solution time.

References

[1] Dixon, L. C. W., and G .P. Szego (eds.). *Towards Global Optimisation 2*. North-Holland: Elsevier Science Ltd., Amsterdam, 1978.

See Also

More About

- “Passing Extra Parameters”
- “Vectorize the Objective and Constraint Functions” on page 6-83

Constrained Minimization Using Pattern Search, Solver-Based

This example shows how to minimize an objective function, subject to nonlinear inequality constraints and bounds, using pattern search. For a problem-based version of this example, see “Constrained Minimization Using Pattern Search, Problem-Based” on page 7-4.

Constrained Minimization Problem

For this problem, the objective function to minimize is a simple function of a 2-D variable x .

$$\text{simple_objective}(x) = (4 - 2.1*x(1)^2 + x(1)^4/3)*x(1)^2 + x(1)*x(2) + (-4 + 4*x(2)^2)*x(2)^2;$$

This function is known as "cam," as described in L.C.W. Dixon and G.P. Szego [1].

Additionally, the problem has nonlinear constraints and bounds.

$x(1)*x(2) + x(1) - x(2) + 1.5 \leq 0$	(nonlinear constraint)
$10 - x(1)*x(2) \leq 0$	(nonlinear constraint)
$0 \leq x(1) \leq 1$	(bound)
$0 \leq x(2) \leq 13$	(bound)

Code the Objective Function

Create a MATLAB® file named `simple_objective.m` containing the following code:

```
type simple_objective
function y = simple_objective(x)
%SIMPLE_OBJECTIVE Objective function for PATTERNSEARCH solver
% Copyright 2004 The MathWorks, Inc.
x1 = x(1);
x2 = x(2);
y = (4-2.1.*x1.^2+x1.^4./3).*x1.^2+x1.*x2+(-4+4.*x2.^2).*x2.^2;
```

Solvers such as `patternsearch` accept a single input x , where x has as many elements as the number of variables in the problem. The objective function computes the scalar value of the objective function and returns it in its single output argument y .

Coding the Constraint Function

Create a MATLAB file named `simple_constraint.m` containing the following code:

```
type simple_constraint
function [c, ceq] = simple_constraint(x)
%SIMPLE_CONSTRAINT Nonlinear inequality constraints.
% Copyright 2005-2007 The MathWorks, Inc.
c = [1.5 + x(1)*x(2) + x(1) - x(2);
     -x(1)*x(2) + 10];
% No nonlinear equality constraints:
ceq = [];
```


The constraint function computes the values of all the inequality and equality constraints and returns the vectors `c` and `ceq`, respectively. The value of `c` represents nonlinear inequality constraints that the solver attempts to make less than or equal to zero. The value of `ceq` represents nonlinear equality constraints that the solver attempts to make equal to zero. This example has no nonlinear equality constraints, so `ceq = []`. For details, see “Nonlinear Constraints”.

Minimize Using patternsearch

Specify the objective function as a function handle.

```
ObjectiveFunction = @simple_objective;
```

Specify the problem bounds.

```
lb = [0 0]; % Lower bounds
ub = [1 13]; % Upper bounds
```

Specify the nonlinear constraint function as a function handle.

```
ConstraintFunction = @simple_constraint;
```

Specify an initial point for the solver.

```
x0 = [0.5 0.5]; % Starting point
```

Call the solver, requesting the optimal point `x` and the function value at the optimal point `fval`.

```
[x,fval] = patternsearch(ObjectiveFunction,x0,[],[],[],[],lb,ub, ...
    ConstraintFunction)
```

```
Optimization finished: mesh size less than options.MeshTolerance
and constraint violation is less than options.ConstraintTolerance.
```

```
x = 1x2
```

```
    0.8122    12.3122
```

```
fval = 9.1324e+04
```

Add Visualization

To observe the solver's progress, specify options that select two plot functions. The plot function `psplotbestf` plots the best objective function value at every iteration, and the plot function `psplotmaxconstr` plots the maximum constraint violation at every iteration. Set these two plot functions in a cell array. Also, display information about the solver's progress in the Command Window by setting the `Display` option to `'iter'`.

```
options = optimoptions(@patternsearch,'PlotFcn',{@psplotbestf,@psplotmaxconstr}, ...
    'Display','iter');
```

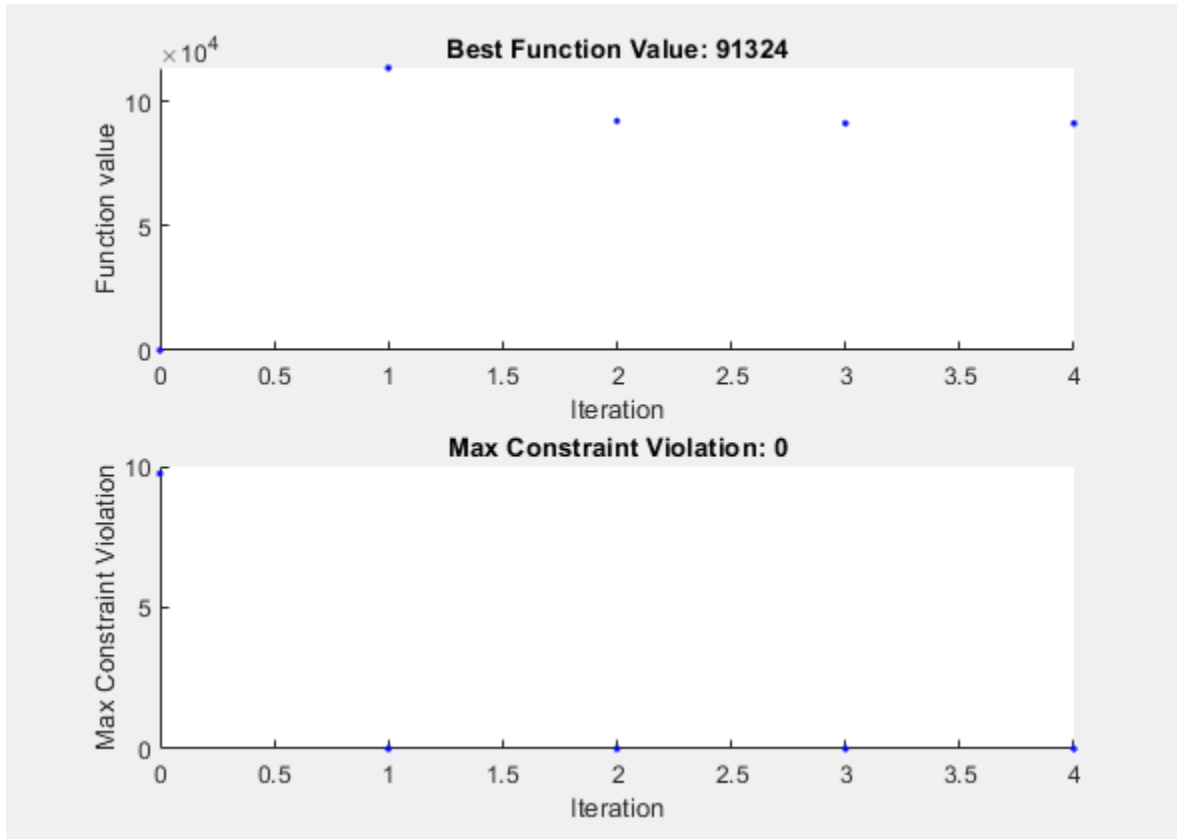
Run the solver, including the options argument.

```
[x,fval] = patternsearch(ObjectiveFunction,x0,[],[],[],[],lb,ub, ...
    ConstraintFunction,options)
```

Iter	Func-count	f(x)	Max Constraint	MeshSize	Method
0	1	0.373958	9.75	0.9086	

1	18	113581	1.617e-10	0.001	Increase penalty
2	147	92267	0	1e-05	Increase penalty
3	373	91333.2	0	1e-07	Increase penalty
4	638	91324	0	1e-09	Increase penalty

Optimization finished: mesh size less than options.MeshTolerance
and constraint violation is less than options.ConstraintTolerance.



$x = 1 \times 2$

0.8122 12.3122

fval = 9.1324e+04

Nonlinear constraints cause `patternsearch` to solve many subproblems at each iteration. As shown in both the plots and the iterative display, the solution process has few iterations. However, the `Func-count` column in the iterative display shows many function evaluations per iteration. Both the plots and the iterative display show that the initial point is infeasible, and that the objective function is low at the initial point. During the solution process, the objective function value initially increases, then decreases to its final value.

References

[1] Dixon, L. C. W., and G .P. Szego (eds.). *Towards Global Optimisation 2*. North-Holland: Elsevier Science Ltd., Amsterdam, 1978.

See Also

More About

- “Write Constraints” on page 2-6

Effects of Pattern Search Options

This example shows the effects of some options for pattern search. The options include plotting, stopping criteria, and other algorithmic controls for speeding a solution. For a list of available options for patternsearch algorithms, see “Options Table for Pattern Search Algorithms” on page 17-21.

Set Up a Problem for Pattern Search

The problem to minimize is a quadratic function of six variables subject to linear equality and inequality constraints. The objective function, `lincontest7`, is available when you run this example.

type `lincontest7`

```
function y = lincontest7(x)
%LINCONTEST7 objective function.
% y = LINCONTEST7(X) evaluates y for the input X. Make sure that x is a column
% vector, whereas objective function gets a row vector.

% Copyright 2003-2017 The MathWorks, Inc.
x = x(:);

%Define a quadratic problem in terms of H and f
H = [36 17 19 12 8 15;
     17 33 18 11 7 14;
     19 18 43 13 8 16;
     12 11 13 18 6 11;
     8 7 8 6 9 8;
     15 14 16 11 8 29];

f = [ 20 15 21 18 29 24 ]';

y = 0.5*x'*H*x + f'*x;
```

Specify the function handle `@lincontest7` as the objective function.

```
objectiveFcn = @lincontest7;
```

The objective function accepts a row vector of length six. Specify an initial point for the optimization.

```
x0 = [2 1 0 9 1 0];
```

Create linear constraint matrices representing the constraints $A_{ineq}x \leq B_{ineq}$ and $A_{eq}x = B_{eq}$. For details, see “Linear Constraints”.

```
Aineq = [-8 7 3 -4 9 0 ];
Bineq = [7];
Aeq = [7 1 8 3 3 3; 5 0 5 1 5 8; 2 6 7 1 1 8; 1 0 0 0 0 0];
Beq = [84 62 65 1];
```

Run the `patternsearch` solver, and note the number of iterations and function evaluations required to reach the solution.

```
[X1,Fval,Exitflag,Output] = patternsearch(objectiveFcn,x0,Aineq,Bineq,Aeq,Beq);
```

Optimization terminated: mesh size less than options.MeshTolerance.

```
fprintf('The number of iterations is: %d\n', Output.iterations);
```

The number of iterations is: 156

```
fprintf('The number of function evaluations is: %d\n', Output.funccount);
```

The number of function evaluations is: 934

```
fprintf('The best function value found is: %g\n', Fval);
```

The best function value found is: 2189.18

Add Visualization

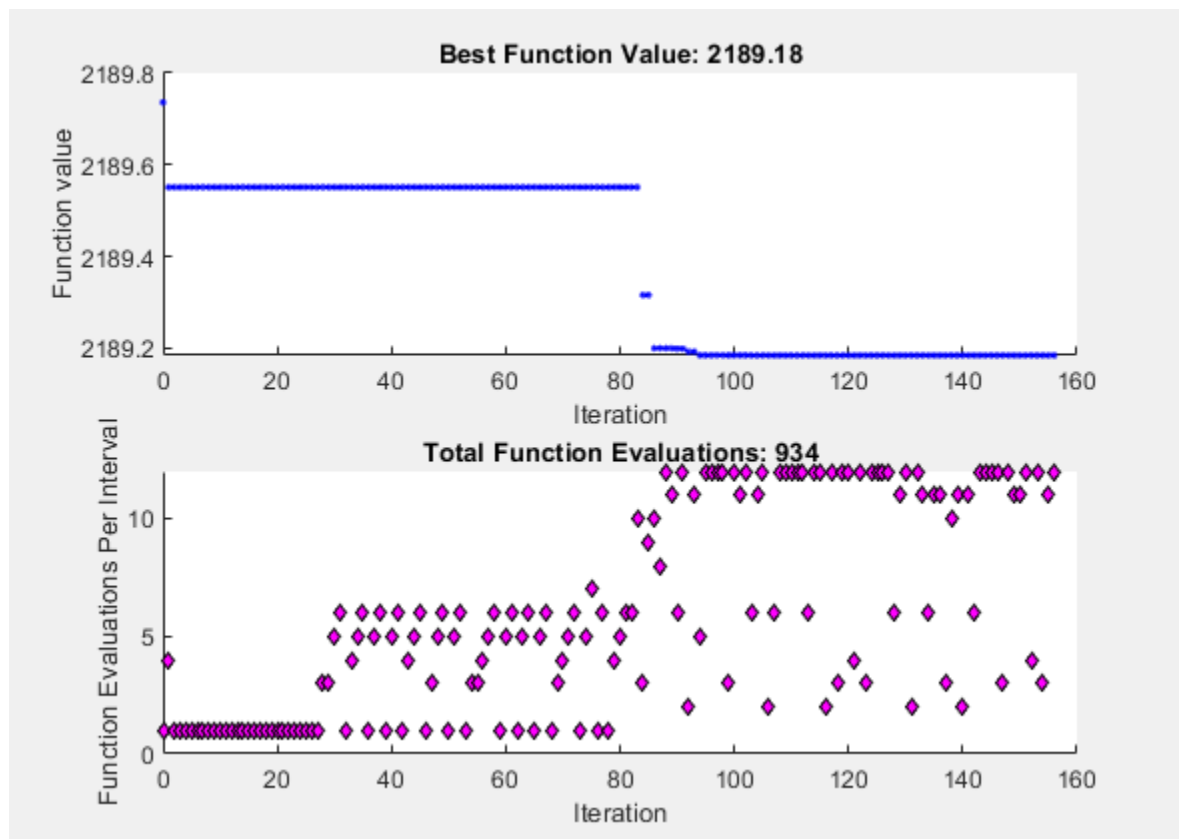
Monitor the optimization process by specifying options that select two plot functions. The plot function `psplotbestf` plots the best objective function value at every iteration, and the plot function `psplotfunccount` plots the number of times the objective function is evaluated at each iteration. Set these two plot functions in a cell array.

```
opts = optimoptions(@patternsearch, 'PlotFcn',{@psplotbestf,@psplotfunccount});
```

Run the `patternsearch` solver, including the `opts` argument. Because the problem has no upper or lower bound constraints and no nonlinear constraints, pass empty arrays (`[]`) for the seventh, eighth, and ninth arguments.

```
[X1,Fval,ExitFlag,Output] = patternsearch(objectiveFcn,x0,Aineq,Bineq, ...
    Aeq,Beq,[],[],[],opts);
```

Optimization terminated: mesh size less than options.MeshTolerance.



Mesh Options

Pattern search involves evaluating the objective function at points in a mesh. The size of the mesh can influence the speed of the solution. You can control the size of the mesh using options.

Initial Mesh Size

The mesh at each iteration is the span of a set of search directions that are added to the current point, scaled by the current mesh size. The solver starts with an initial mesh size of 1 by default. To start the initial mesh size at 1/2, set the `InitialMeshSize` option.

```
opts = optimoptions(opts, 'InitialMeshSize', 1/2);
```

Mesh Scaling

You can scale the mesh to improve the minimization of a poorly scaled optimization problem. Scaling rotates the pattern by some degree and scales along the search directions. The `ScaleMesh` option is on (`true`) by default, but you can turn it off if the problem is well scaled. In general, if the problem is poorly scaled, setting this option to `true` can reduce the number of function evaluations. For this problem, set `ScaleMesh` to `false`, because `lincontest7` is a well-scaled objective function.

```
opts = optimoptions(opts, 'ScaleMesh', false);
```

Mesh Accelerator

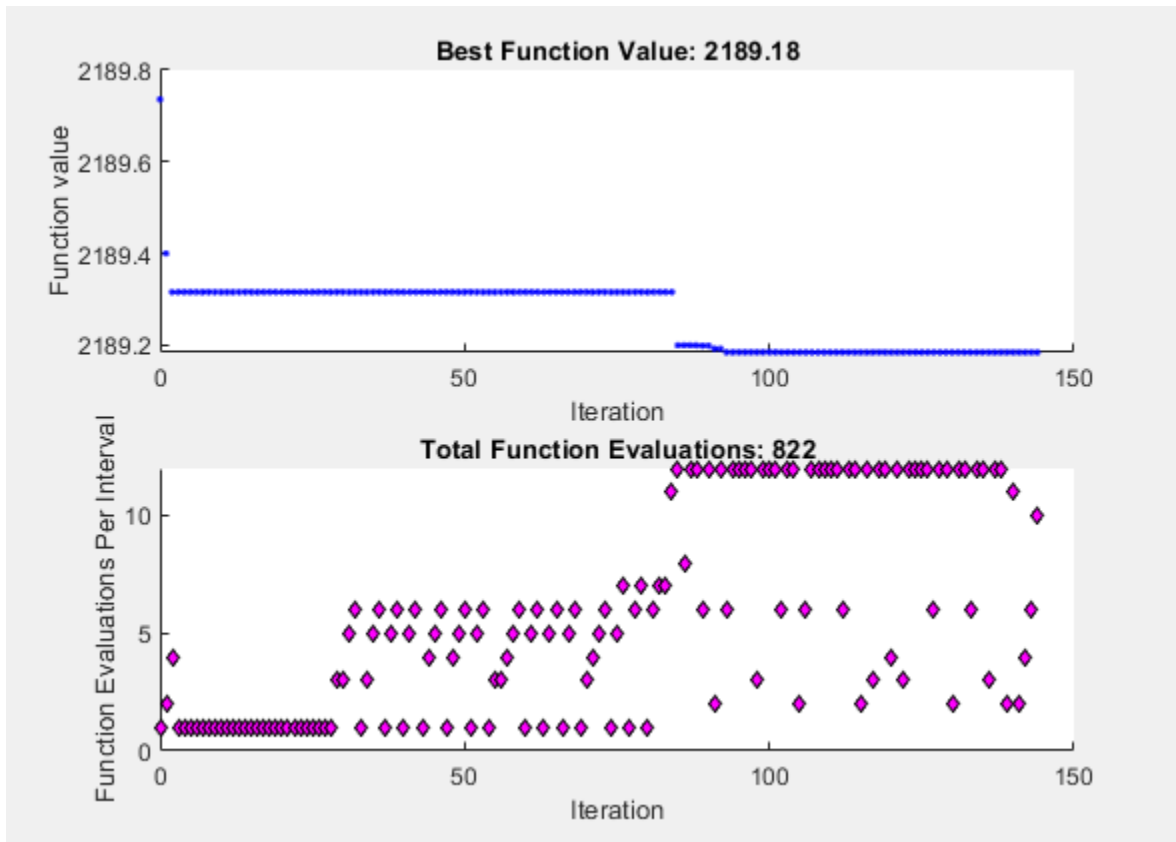
Direct search methods require many function evaluations compared to derivative-based optimization methods. The pattern search algorithm can quickly find the neighborhood of an optimum point, but can be slow in detecting the minimum itself. The `patternsearch` solver can reduce the number of function evaluations by using an accelerator. When the accelerator is on (`opts.AccelerateMesh = true`), the solver contracts the mesh size rapidly after reaching a minimum mesh size. This option is recommended only for smooth problems; in other types of problems, you can lose some accuracy. The `AccelerateMesh` option is off (`false`) by default. For this problem, set `AccelerateMesh` to `true` because the objective function is smooth.

```
opts = optimoptions(opts, 'AccelerateMesh', true);
```

Run the `patternsearch` solver.

```
[X2, Fval, ExitFlag, Output] = patternsearch(objectiveFcn, x0, Aineq, Bineq, ...  
    Aeq, Beq, [], [], [], opts);
```

```
Optimization terminated: mesh size less than options.MeshTolerance.
```



```
fprintf('The number of iterations is: %d\n', Output.iterations);
```

```
The number of iterations is: 144
```

```
fprintf('The number of function evaluations is: %d\n', Output.funccount);
```

```
The number of function evaluations is: 822
```

```
fprintf('The best function value found is: %g\n', Fval);
```

```
The best function value found is: 2189.18
```

The mesh option settings reduce the number of iterations and the number of function evaluations, and with no apparent loss of accuracy.

Stopping Criteria and Tolerances

`MeshTolerance` is a tolerance on the mesh size. If the mesh size is less than `MeshTolerance`, the solver stops. `StepTolerance` is the minimum tolerance on the change in the current point to the next point. `FunctionTolerance` is the minimum tolerance on the change in the function value from the current point to the next point.

Set the `MeshTolerance` to $1e-7$, which is ten times smaller than the default value. This setting can increase the number of function evaluations and iterations, and can lead to a more accurate solution.

```
opts.MeshTolerance = 1e-7;
```

Search Methods in Pattern Search

The pattern search algorithm can use an additional search method at every iteration, based on the value of the `SearchFcn` option. When you specify a search method using `SearchFcn`, `patternsearch` performs the specified search first, before the mesh search. If the search method is successful, `patternsearch` skips the mesh search, commonly called the poll function, for that iteration. If the search method is unsuccessful in improving the current point, `patternsearch` performs the mesh search.

You can specify different search methods for `SearchFcn`, including `searchga` and `searchneldermead`, which are optimization algorithms. Use these two search methods only for the first iteration, which is the default setting. Using either of these methods at every iteration might not improve the results and can be computationally expensive. However, you can use the `searchlhs` method, which generates Latin hypercube points, at every iteration or possibly every 10 iterations.

Other choices for search methods include poll methods such as positive basis $N+1$ or positive basis $2N$. A recommended strategy is to use positive basis $N+1$ (which requires at most $N+1$ points to create a pattern) as a search method and positive basis $2N$ (which requires $2N$ points to create a pattern) as a poll method.

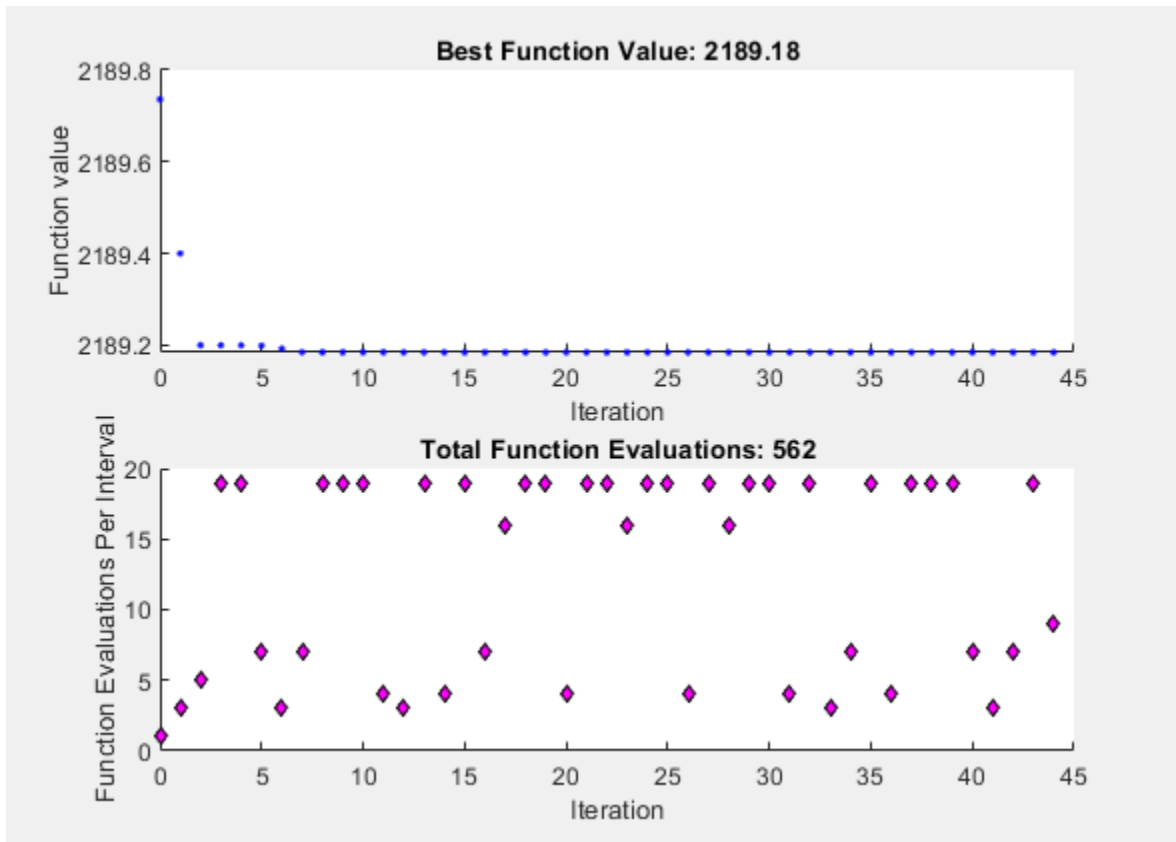
Update the options structure to use `positivebasisnpl` as the search method. Because positive basis $2N$ is the default for the `PollFcn` option, do not set that option.

```
opts.SearchFcn = @positivebasisnpl;
```

Run the `patternsearch` solver.

```
[X5,Fval,ExitFlag,Output] = patternsearch(objectiveFcn,x0,Aineq,Bineq,Aeq,Beq, ...  
    [],[],[],opts);
```

Optimization terminated: change in X less than options.StepTolerance.



```
fprintf('The number of iterations is: %d\n', Output.iterations);
```

The number of iterations is: 44

```
fprintf('The number of function evaluations is: %d\n', Output.funccount);
```

The number of function evaluations is: 562

```
fprintf('The best function value found is: %g\n', Fval);
```

The best function value found is: 2189.18

The total number of iterations and function evaluations decreases, even though the mesh tolerance is smaller than its previous value and is the stopping criterion that halts the solver.

See Also

More About

- “Set Mesh Options” on page 6-66
- “Pattern Search Options” on page 17-7
- “Custom Plot Function” on page 6-48

Pattern Search Terminology

In this section...

“Patterns” on page 6-24

“Meshes” on page 6-24

“Polling” on page 6-25

“Expanding and Contracting” on page 6-25

Patterns

A *pattern* is a set of vectors $\{v_i\}$ that the pattern search algorithm uses to determine which points to search at each iteration. The set $\{v_i\}$ is defined by the number of independent variables in the objective function, N , and the positive basis set. Two commonly used positive basis sets in pattern search algorithms are the maximal basis, with $2N$ vectors, and the minimal basis, with $N+1$ vectors.

With GPS, the collection of vectors that form the pattern are fixed-direction vectors. For example, if there are three independent variables in the optimization problem, the default for a $2N$ positive basis consists of the following pattern vectors:

$$\begin{aligned} v_1 &= [1 \ 0 \ 0] & v_2 &= [0 \ 1 \ 0] & v_3 &= [0 \ 0 \ 1] \\ v_4 &= [-1 \ 0 \ 0] & v_5 &= [0 \ -1 \ 0] & v_6 &= [0 \ 0 \ -1] \end{aligned}$$

An $N+1$ positive basis consists of the following default pattern vectors.

$$\begin{aligned} v_1 &= [1 \ 0 \ 0] & v_2 &= [0 \ 1 \ 0] & v_3 &= [0 \ 0 \ 1] \\ v_4 &= [-1 \ -1 \ -1] \end{aligned}$$

With GSS, the pattern is identical to the GPS pattern, except when there are linear constraints and the current point is near a constraint boundary. For a description of the way in which GSS forms a pattern with linear constraints, see Kolda, Lewis, and Torczon [1]. The GSS algorithm is more efficient than the GPS algorithm when you have linear constraints. For an example showing the efficiency gain, see “Compare the Efficiency of Poll Options” on page 6-61.

With MADS, the collection of vectors that form the pattern are randomly selected by the algorithm. Depending on the poll method choice, the number of vectors selected will be $2N$ or $N+1$. As in GPS, $2N$ vectors consist of N vectors and their N negatives, while $N+1$ vectors consist of N vectors and one that is the negative of the sum of the others.

References

- [1] Kolda, Tamara G., Robert Michael Lewis, and Virginia Torczon. “A generating set direct search augmented Lagrangian algorithm for optimization with a combination of general and linear constraints.” Technical Report SAND2006-5315, Sandia National Laboratories, August 2006.

Meshes

At each step, `patternsearch` searches a set of points, called a *mesh*, for a point that improves the objective function. `patternsearch` forms the mesh by

- 1 Generating a set of vectors $\{d_i\}$ by multiplying each pattern vector v_i by a scalar Δ^m . Δ^m is called the *mesh size*.
- 2 Adding the $\{d_i\}$ to the *current point*—the point with the best objective function value found at the previous step.

For example, using the GPS algorithm, suppose that:

- The current point is [1.6 3.4].
- The pattern consists of the vectors

$$v_1 = [1 \ 0]$$

$$v_2 = [0 \ 1]$$

$$v_3 = [-1 \ 0]$$

$$v_4 = [0 \ -1]$$

- The current mesh size Δ^m is 4.

The algorithm multiplies the pattern vectors by 4 and adds them to the current point to obtain the following mesh.

$$\begin{aligned} [1.6 \ 3.4] + 4*[1 \ 0] &= [5.6 \ 3.4] \\ [1.6 \ 3.4] + 4*[0 \ 1] &= [1.6 \ 7.4] \\ [1.6 \ 3.4] + 4*[-1 \ 0] &= [-2.4 \ 3.4] \\ [1.6 \ 3.4] + 4*[0 \ -1] &= [1.6 \ -0.6] \end{aligned}$$

The pattern vector that produces a mesh point is called its *direction*.

Polling

At each step, the algorithm polls the points in the current mesh by computing their objective function values. When the **Complete poll** option has the (default) setting `Off`, the algorithm stops polling the mesh points as soon as it finds a point whose objective function value is less than that of the current point. If this occurs, the poll is called *successful* and the point it finds becomes the current point at the next iteration.

The algorithm only computes the mesh points and their objective function values up to the point at which it stops the poll. If the algorithm fails to find a point that improves the objective function, the poll is called *unsuccessful* and the current point stays the same at the next iteration.

When the **Complete poll** option has the setting `On`, the algorithm computes the objective function values at all mesh points. The algorithm then compares the mesh point with the smallest objective function value to the current point. If that mesh point has a smaller value than the current point, the poll is successful.

Expanding and Contracting

After polling, the algorithm changes the value of the mesh size Δ^m . The default is to multiply Δ^m by 2 after a successful poll, and by 0.5 after an unsuccessful poll.

See Also

More About

- “How Pattern Search Polling Works” on page 6-27
- “Searching and Polling” on page 6-37
- “Effects of Pattern Search Options” on page 6-18

How Pattern Search Polling Works

In this section...

“Context” on page 6-27

“Successful Polls” on page 6-27

“An Unsuccessful Poll” on page 6-29

“Successful and Unsuccessful Polls in MADS” on page 6-30

“Displaying the Results at Each Iteration” on page 6-31

“More Iterations” on page 6-31

“Poll Method” on page 6-32

“Complete Poll” on page 6-33

“Stopping Conditions for the Pattern Search” on page 6-33

“Robustness of Pattern Search” on page 6-34

Context

`patternsearch` finds a sequence of points, x_0, x_1, x_2, \dots , that approach an optimal point. The value of the objective function either decreases or remains the same from each point in the sequence to the next. This section explains how pattern search works for the function described in “Optimize Using the GPS Algorithm” on page 6-3.

To simplify the explanation, this section describes how the generalized pattern search (GPS) works using the default maximal positive basis of $2N$, with the `ScaleMesh` option set to `false`.

This section does not show how the `patternsearch` algorithm works with bounds or linear constraints. For bounds and linear constraints, `patternsearch` modifies poll points to be feasible at every iteration, meaning to satisfy all bounds and linear constraints.

This section does not encompass nonlinear constraints. To understand how `patternsearch` works with nonlinear constraints, see “Nonlinear Constraint Solver Algorithm for Pattern Search” on page 6-46.

Successful Polls

The pattern search begins at the initial point x_0 that you provide. In this example, $x_0 = [2.1 \ 1.7]$.

Iteration 1

At the first iteration, the mesh size is 1 and the GPS algorithm adds the pattern vectors to the initial point $x_0 = [2.1 \ 1.7]$ to compute the following mesh points:

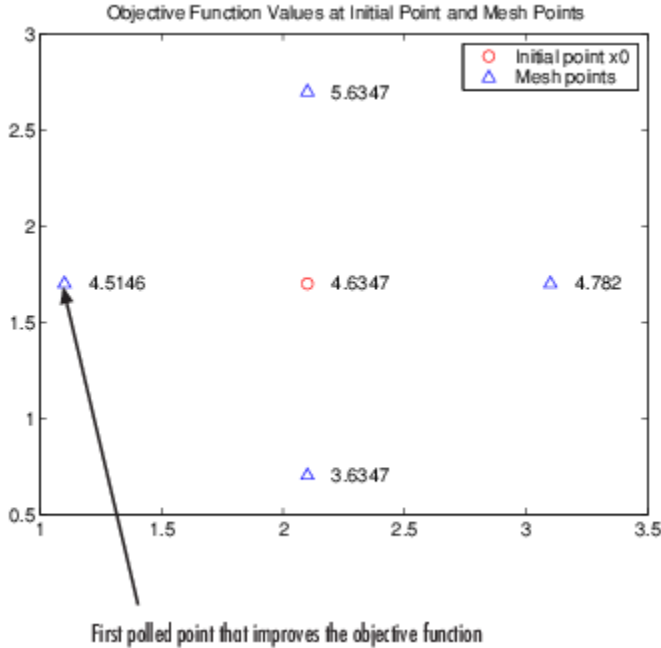
$$[1 \ 0] + x_0 = [3.1 \ 1.7]$$

$$[0 \ 1] + x_0 = [2.1 \ 2.7]$$

$$[-1 \ 0] + x_0 = [1.1 \ 1.7]$$

$$[0 \ -1] + x_0 = [2.1 \ 0.7]$$

The algorithm computes the objective function at the mesh points in the order shown above. The following figure shows the value of the objective function at the initial point and mesh points.



The algorithm polls the mesh points by computing their objective function values until it finds one whose value is smaller than 4.6347, the value at x_0 . In this case, the first such point it finds is [1.1 1.7], at which the value of the objective function is 4.5146, so the poll at iteration 1 is *successful*. The algorithm sets the next point in the sequence equal to

$$x_1 = [1.1 \ 1.7]$$

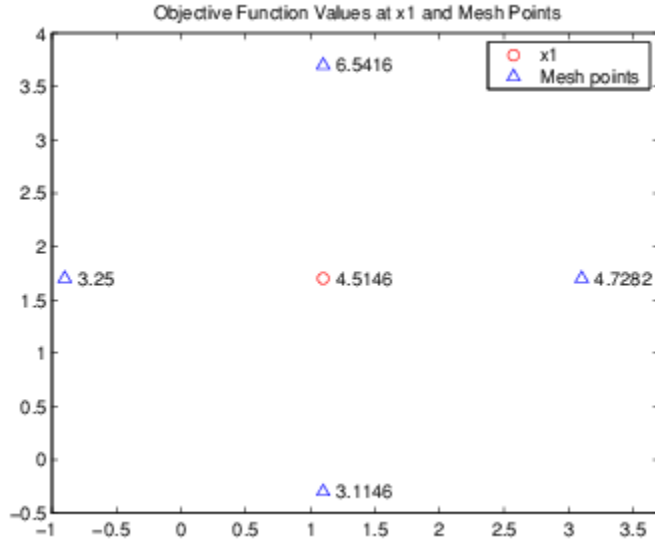
Note By default, the GPS pattern search algorithm stops the current iteration as soon as it finds a mesh point whose fitness value is smaller than that of the current point. Consequently, the algorithm might not poll all the mesh points. You can make the algorithm poll all the mesh points by setting the `UseCompletePoll` option to `true`.

Iteration 2

After a successful poll, the algorithm multiplies the current mesh size by 2, the default value of the `MeshExpansionFactor` options. Because the initial mesh size is 1, at the second iteration the mesh size is 2. The mesh at iteration 2 contains the following points:

$$\begin{aligned} 2*[1 \ 0] + x_1 &= [3.1 \ 1.7] \\ 2*[0 \ 1] + x_1 &= [1.1 \ 3.7] \\ 2*[-1 \ 0] + x_1 &= [-0.9 \ 1.7] \\ 2*[0 \ -1] + x_1 &= [1.1 \ -0.3] \end{aligned}$$

The following figure shows the point x_1 and the mesh points, together with the corresponding values of the objective function.



The algorithm polls the mesh points until it finds one whose value is smaller than 4.5146, the value at x_1 . The first such point it finds is $[-0.9 \ 1.7]$, at which the value of the objective function is 3.25, so the poll at iteration 2 is again successful. The algorithm sets the second point in the sequence equal to

$$x_2 = [-0.9 \ 1.7]$$

Because the poll is successful, the algorithm multiplies the current mesh size by 2 to get a mesh size of 4 at the third iteration.

An Unsuccessful Poll

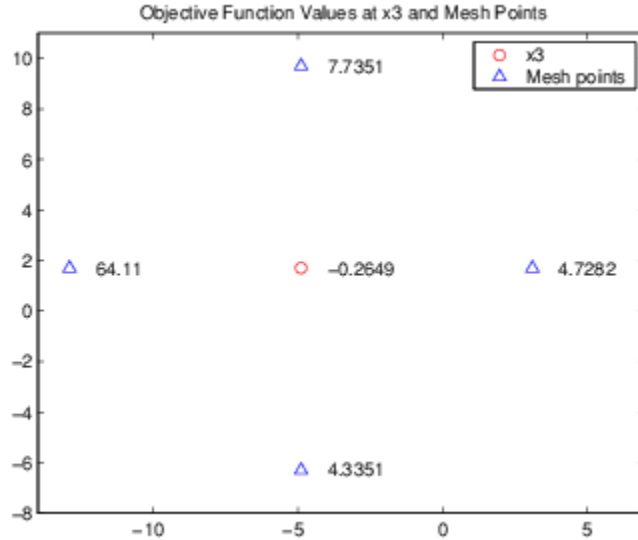
By the fourth iteration, the current point is

$$x_3 = [-4.9 \ 1.7]$$

and the mesh size is 8, so the mesh consists of the points

$$\begin{aligned} 8*[1 \ 0] + x_3 &= [3.1 \ 1.7] \\ 8*[0 \ 1] + x_3 &= [-4.9 \ 9.7] \\ 8*[-1 \ 0] + x_3 &= [-12.9 \ 1.7] \\ 8*[0 \ -1] + x_3 &= [-4.9 \ -1.3] \end{aligned}$$

The following figure shows the mesh points and their objective function values.



At this iteration, none of the mesh points has a smaller objective function value than the value at x_3 , so the poll is *unsuccessful*. In this case, the algorithm does not change the current point at the next iteration. That is,

$$x_4 = x_3;$$

At the next iteration, the algorithm multiplies the current mesh size by 0.5, the default value of the `MeshContractionFactor` option, so that the mesh size at the next iteration is 4. The algorithm then polls with a smaller mesh size.

Successful and Unsuccessful Polls in MADS

Setting the `PollMethod` option to 'MADSPositiveBasis2N' or 'MADSPositiveBasisNp1' causes `patternsearch` to use both a different poll type and to react to polling differently than the other polling algorithms.

A MADS poll uses newly generated pseudorandom mesh vectors at each iteration. The vectors are randomly shuffled components from the columns of a random lower-triangular matrix. The components of the matrix have integer sizes up to $1/\sqrt{\text{mesh size}}$. In the poll, the mesh vectors are multiplied by the mesh size, so the poll points can be up to $\sqrt{\text{mesh size}}$ from the current point.

Unsuccessful polls contract the mesh by a factor of 4, ignoring the `MeshContractionFactor` option. Similarly, successful polls expand the mesh by a factor of 4, ignoring the `MeshExpansionFactor` option. The maximum mesh size is 1, despite any setting of the `MaxMeshSize` option.

In addition, when there is a successful poll, `patternsearch` starts at the successful point and polls again. This extra poll uses the same mesh vectors, expanded by a factor of 4 while staying below size 1. The extra poll looks again along the same directions that were just successful.

Displaying the Results at Each Iteration

You can display the results of the pattern search at each iteration by setting the `Display` option to `'iter'`. This enables you to evaluate the progress of the pattern search and to make changes to options if necessary.

With this setting, the pattern search displays information about each iteration at the command line. The first four iterations are

Iter	f-count	f(x)	MeshSize	Method
0	1	4.63474	1	
1	4	4.51464	2	Successful Poll
2	7	3.25	4	Successful Poll
3	10	-0.264905	8	Successful Poll
4	14	-0.264905	4	Refine Mesh

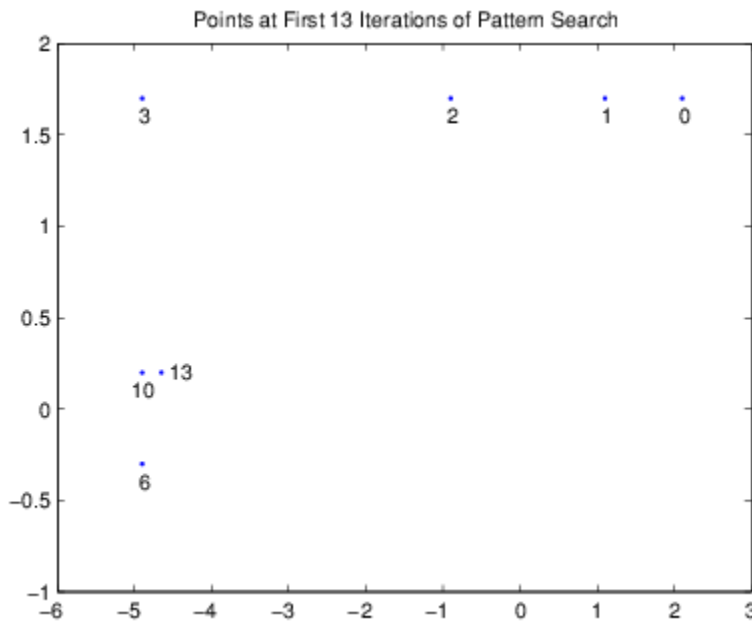
The entry `Successful Poll` below `Method` indicates that the current iteration was successful. For example, the poll at iteration 2 is successful. As a result, the objective function value of the point computed at iteration 2, displayed below $f(x)$, is less than the value at iteration 1.

At iteration 4, the entry `Refine Mesh` tells you that the poll is unsuccessful. As a result, the function value at iteration 4 remains unchanged from iteration 3.

By default, the pattern search doubles the mesh size after each successful poll and halves it after each unsuccessful poll.

More Iterations

The pattern search performs 60 iterations before stopping. The following plot shows the points in the sequence computed in the first 13 iterations of the pattern search.



The numbers below the points indicate the first iteration at which the algorithm finds the point. The plot only shows iteration numbers corresponding to successful polls, because the best point doesn't change after an unsuccessful poll. For example, the best point at iterations 4 and 5 is the same as at iteration 3.

Poll Method

At each iteration, the pattern search polls the points in the current mesh—that is, it computes the objective function at the mesh points to see if there is one whose function value is less than the function value at the current point. “How Pattern Search Polling Works” on page 6-27 provides an example of polling. You can specify the pattern that defines the mesh by the `PollMethod` option. The default pattern, `'GPSPositiveBasis2N'`, consists of the following $2N$ directions, where N is the number of independent variables for the objective function.

```
[1 0 0...0]
[0 1 0...0]
...
[0 0 0...1]
[-1 0 0...0]
[0 -1 0...0]
[0 0 0...-1].
```

For example, if the objective function has three independent variables, the `GPS Positive basis 2N`, consists of the following six vectors.

```
[1 0 0]
[0 1 0]
[0 0 1]
[-1 0 0]
[0 -1 0]
[0 0 -1].
```

Alternatively, you can set the `PollMethod` option to `'GPSPositiveBasisNp1'`, the pattern consisting of the following $N + 1$ directions.

```
[1 0 0...0]
[0 1 0...0]
...
[0 0 0...1]
[-1 -1 -1...-1].
```

For example, if objective function has three independent variables, the `'GPSPositiveBasisNp1'` consists of the following four vectors.

```
[1 0 0]
[0 1 0]
[0 0 1]
[-1 -1 -1].
```

A pattern search will sometimes run faster using `'GPSPositiveBasisNp1'` rather than the `'GPSPositiveBasis2N'` as the `PollMethod`, because the algorithm searches fewer points at each iteration. Although not being addressed in this example, the same is true when using the `'MADSPositiveBasisNp1'` over the `'MADSPositiveBasis2N'`, and similarly for GSS. For

example, if you run a pattern search on the linearly constrained example in “Constrained Minimization Using patternsearch and Optimize Live Editor Task” on page 6-71, the algorithm performs 1588 function evaluations with 'GPSPositiveBasis2N', the default PollMethod, but only 877 function evaluations using 'GPSPositiveBasisNp1'. For more detail, see “Compare the Efficiency of Poll Options” on page 6-61.

However, if the objective function has many local minima, using 'GPSPositiveBasis2N' as the PollMethod might avoid finding a local minimum that is not the global minimum, because the search explores more points around the current point at each iteration.

Complete Poll

By default, if the pattern search finds a mesh point that improves the value of the objective function, it stops the poll and sets that point as the current point for the next iteration. When this occurs, some mesh points might not get polled. Some of these unpolled points might have an objective function value that is even lower than the first one the pattern search finds.

For problems in which there are several local minima, it is sometimes preferable to make the pattern search poll *all* the mesh points at each iteration and choose the one with the best objective function value. This enables the pattern search to explore more points at each iteration and thereby potentially avoid a local minimum that is not the global minimum. Have the pattern search poll the entire mesh setting the UseCompletePoll option to true.

Stopping Conditions for the Pattern Search

The algorithm stops when any of the following conditions occurs:

- The mesh size is less than the MeshTolerance option.
- The number of iterations performed by the algorithm reaches the value of the MaxIterations option.
- The total number of objective function evaluations performed by the algorithm reaches the value of the MaxFunctionEvaluations option.
- The time, in seconds, the algorithm runs until it reaches the value of the MaxTime option.
- After a successful poll, the distance between the point found in the previous two iterations and the mesh size are both less than the StepTolerance option.
- After a successful poll, the change in the objective function in the previous two iterations is less than the FunctionTolerance option and the mesh size is less than the StepTolerance option.

The ConstraintTolerance option is not used as stopping criterion. It determines the feasibility with respect to nonlinear constraints.

The MADS algorithm uses an additional parameter called the poll parameter, Δ_p , in the mesh size stopping criterion:

$$\Delta_p = \begin{cases} N\sqrt{\Delta_m} & \text{for positive basis } N + 1 \text{ poll} \\ \sqrt{\Delta_m} & \text{for positive basis } 2N \text{ poll,} \end{cases}$$

where Δ_m is the mesh size. The MADS stopping criterion is:

$$\Delta_p \leq \text{MeshTolerance.}$$

Robustness of Pattern Search

The pattern search algorithm is robust in relation to objective function failures. This means `patternsearch` tolerates function evaluations resulting in NaN, Inf, or complex values. When the objective function at the initial point x_0 is a real, finite value, `patternsearch` treats poll point failures as if the objective function values are large, and ignores them.

For example, if all points in a poll evaluate to NaN, `patternsearch` considers the poll unsuccessful, shrinks the mesh, and reevaluates. If even one point in a poll evaluates to a smaller value than any seen yet, `patternsearch` considers the poll successful, and expands the mesh.

See Also

More About

- “Optimize Using the GPS Algorithm” on page 6-3
- “Constrained Minimization Using `patternsearch` and Optimize Live Editor Task” on page 6-71
- “Vectorize the Objective and Constraint Functions” on page 6-83
- “Search and Poll” on page 6-42

Nonuniform Pattern Search (NUPS) Algorithm

When you set the `Algorithm` option for the `patternsearch` function to "nups", the algorithm performs search and poll steps as described in "Searching and Polling" on page 6-37 with the following modifications.

- The poll ignores the settings of the options `MeshContractionFactor`, `MeshExpansionFactor`, `PollMethod`, `PollOrderAlgorithm`, `UseCompletePoll`, and `MeshRotate`. The mesh expansion factor depends on the iterations (as described in the sufficient decrease bullet later in this list). The mesh contraction factor is always $1/2$.
- The "nups" algorithm works in a cycle of 16 iterations. In each cycle, the algorithm uses GPS (generalized pattern search) for the first part of the cycle and MADS (mesh adaptive direct search) for the remainder. Each poll method can be used up to 14 iterations in a cycle. The number of iterations used for a poll method in a future cycle depends on the improvement of the objective function per function evaluation in the current cycle.
 - When the problem has no linear constraints or the current point is not at a linear constraint boundary, each poll uses $N+1$ points. Alternately the N coordinate directions and the point $[1,1,1,\dots,1]$; the next poll uses the negative of these directions. MADS uses these same poll points with an orthogonal set of directions chosen uniformly at random. (MADS does not apply when the problem has linear equality constraints.)
 - When the problem has linear constraints and the current point is at a constraint boundary, the poll uses the tangent directions from the linear constraints, plus the appropriate directions from the remaining dimensions in the poll. Here, appropriate directions are those that are feasible from the current point.
- When the problem has linear constraints, the algorithm uses a line search, if necessary, to ensure that all points in a poll are feasible with respect to the constraints and bounds. In contrast, if a potential poll point is infeasible in the "classic" algorithm, that point is simply eliminated from the poll. So, the "nups" algorithm can use more poll points, potentially resulting in a more effective poll.
- Sufficient decrease means a poll finds a point with an objective function value that is sufficiently lower than the current value. Let F_c be the objective function value at the current point, and F_n be the objective function value at the next proposed point. The value F_n satisfies the sufficient decrease condition when:

$$(F_c - F_n) / \max(1, \text{abs}(F_c)) \geq \text{ImprovementTolerance}.$$

The poll continues until a proposed point satisfies the sufficient decrease condition or until all points are examined. Then the algorithm takes the first applicable action in this list:

- If the poll finds no point that improves the current value, the poll is deemed unsuccessful and the mesh size decreases by a factor of $1/2$.
- If the poll finds a point that satisfies the sufficient decrease condition, the poll is deemed successful and the mesh size increases by a factor of 2.
- If, for three successive iterations, the poll finds a point that improves the current value, but not enough to satisfy the sufficient decrease condition, the third poll is deemed successful and the mesh size increases by a factor of $3/2$.
- If the poll finds a point that improves the current value and the current mesh size is larger than a threshold, the poll is deemed successful and the mesh size increases by a factor of 2.
- Otherwise, the poll is deemed successful but the mesh size does not change.

The effect of the sufficient decrease condition is to keep the mesh size small when a step leads to a very small improvement in the objective function value. This behavior can help to keep the algorithm from overreacting to small changes in the objective function value, potentially resulting in faster convergence.

- You cannot use a poll method as a search method.

When you set the `Algorithm` option to "nups-gps", the algorithm is the same as "nups" except it performs all polling using GPS. When you set the `Algorithm` option to "nups-mads", the algorithm is the same as "nups" except it performs all polling using OrthoMADS.

`patternsearch` handles nonlinear constraints in the same way for all algorithms, as described in "Nonlinear Constraint Solver Algorithm for Pattern Search" on page 6-46.

When `UseParallel` is true, `patternsearch` performs a complete poll for all algorithms.

See Also

`patternsearch`

Related Examples

- "Explore `patternsearch` Algorithms" on page 6-103
- "Explore `patternsearch` Algorithms in Optimize Live Editor Task" on page 6-107
- "Direct Search"
- "Pattern Search Options" on page 17-7

Searching and Polling

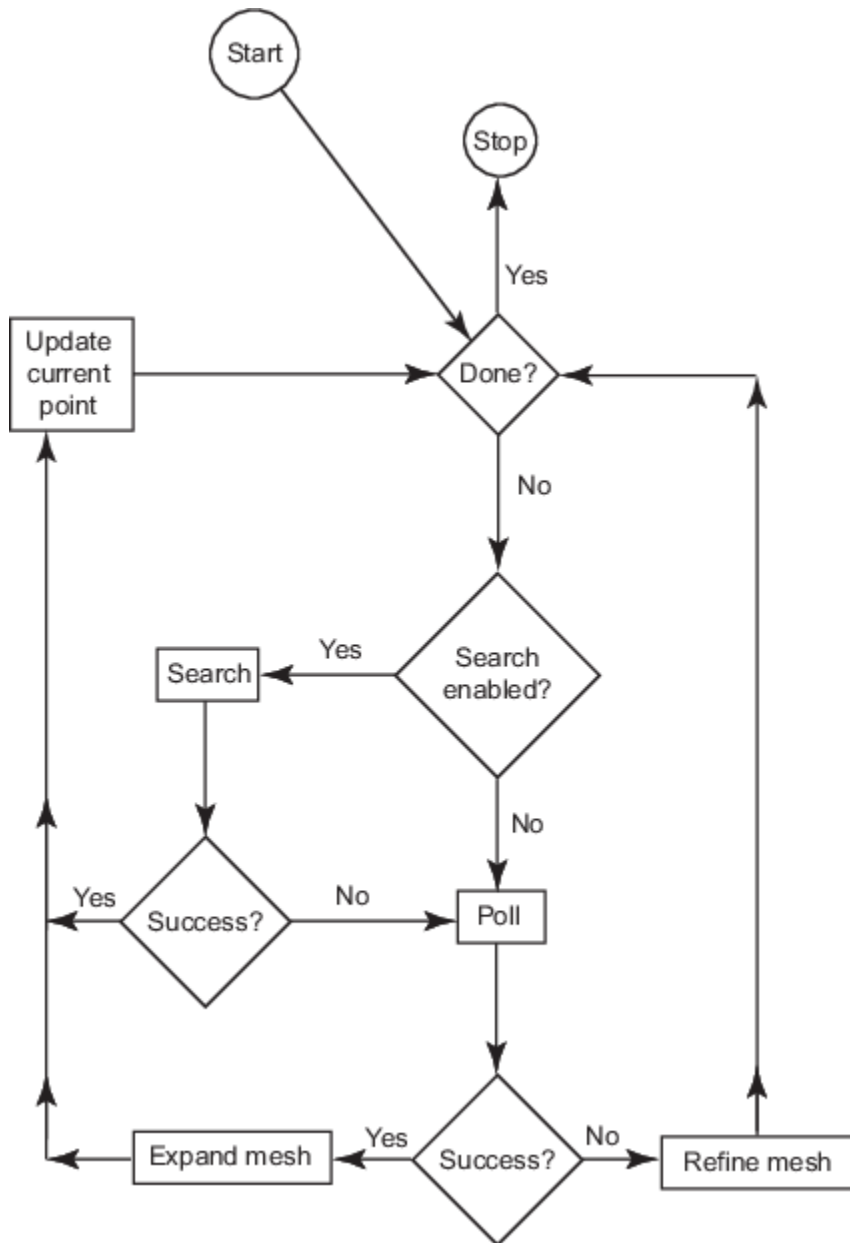
In this section...
“Definition of Search” on page 6-37
“How to Use a Search Method” on page 6-38
“Built-In Search Types” on page 6-39
“When to Use Search” on page 6-40

Definition of Search

In `patternsearch`, a search is an algorithm that runs before a poll. The search attempts to locate a better point than the current point. (Better means one with lower objective function value.) If the search finds a better point, the better point becomes the current point, and no polling is done at that iteration. If the search does not find a better point, `patternsearch` performs a poll.

By default, `patternsearch` does not use search. To search, see “How to Use a Search Method” on page 6-38.

The figure “`patternsearch` With a Search Method” on page 6-38 contains a flow chart of direct search including using a search method.



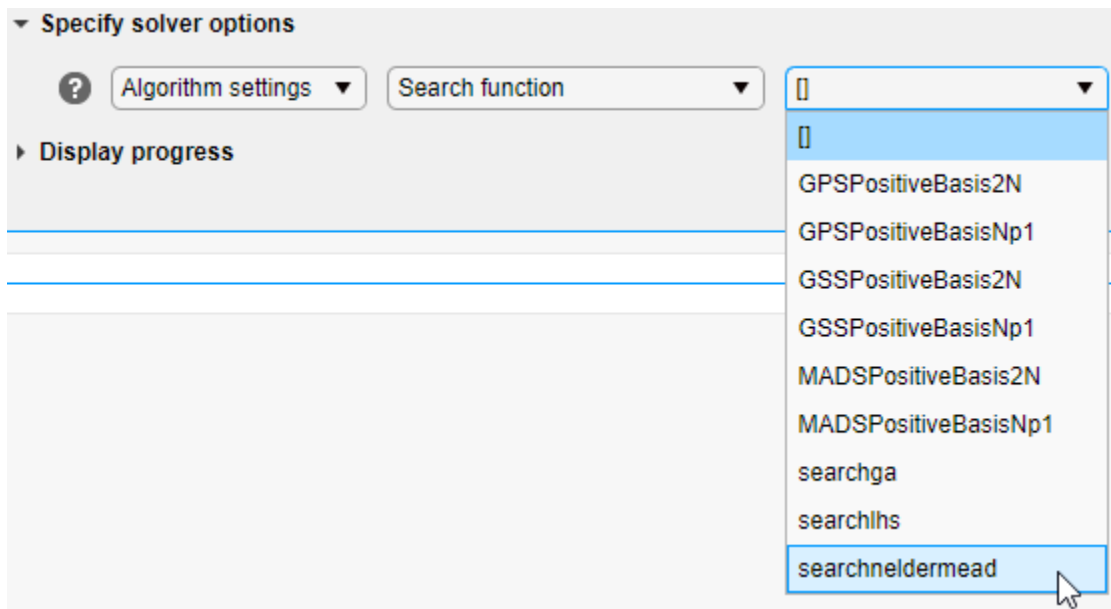
patternsearch With a Search Method

Iteration limit applies to all built-in search methods except those that are poll methods. If you select an iteration limit for the search method, the search is enabled until the iteration limit is reached. Afterward, patternsearch stops searching and only polls.

How to Use a Search Method

To use search in patternsearch:

- In the **Optimize** Live Editor task, select a search function in **Specify solver options > Algorithm settings > Search function**.



- At the command line, create options with a search method using `optimoptions`. For example, to use Latin hypercube search:

```
opts = optimoptions("patternsearch", "SearchFcn", @searchlhs);
```

For more information, including a list of all built-in search methods, consult the `patternsearch` function reference page, and the “Search Options” on page 17-12 section of the options reference.

You can write your own search method. Use the syntax described in “Structure of the Search Function” on page 17-14. To use your search method in a pattern search, give its function handle as the Custom **Function** (`SearchFcn`) option.

Built-In Search Types

- Poll methods — You can use any poll method as a search algorithm in the “classic” algorithm. `patternsearch` conducts one poll step as a search. For this type of search to be beneficial, your search type should be different from your poll type. (`patternsearch` does not search if the selected search method is the same as the poll type.) Therefore, use a MADS search with a GSS or GPS poll, or use a GSS or GPS search with a MADS poll.

Note You can use a poll method as a search method only when the `Algorithm` option is “classic”.

- `fminsearch`, also called Nelder-Mead — `fminsearch` is for unconstrained problems only. `fminsearch` runs to its natural stopping criteria; it does not take just one step. Therefore, use `fminsearch` for just one iteration. This is the default setting. To change settings, see “Search Options” on page 17-12.
- `ga` — `ga` runs to its natural stopping criteria; it does not take just one step. Therefore, use `ga` for just one iteration. This is the default setting. To change settings, see “Search Options” on page 17-12.

- Latin hypercube search — Described in “Search Options” on page 17-12. By default, searches $15n$ points, where n is the number of variables, and only searches during the first iteration. To change settings, see “Search Options” on page 17-12.
- "rbfsurrogate" — As described in “Search Options” on page 17-12, searches using a radial basis function surrogate, similar to the `surrogateopt` surrogate (see “Surrogate Optimization Algorithm” on page 11-3). This search can lower the number of function evaluations, but is relatively time-consuming, so is best suited for time-consuming objective functions.

When to Use Search

There are two main reasons to use a search method:

- To speed an optimization (see “Search Methods for Increased Speed” on page 6-40)
- To obtain a better local solution, or to obtain a global solution on page 1-24 (see “Search Methods for Better Solutions” on page 6-40)

Search Methods for Increased Speed

Generally, you do not know beforehand whether a search method speeds an optimization or not. So try a search method when:

- You are performing repeated optimizations on similar problems, or on the same problem with different parameters.
- You can experiment with different search methods to find a lower solution time.

Search does not always speed an optimization. For one example where it does, see “Search and Poll” on page 6-42.

Search Methods for Better Solutions

Since search methods run before poll methods, using search can be equivalent to choosing a different starting point for your optimization. This comment holds for the Nelder-Mead, `ga`, and Latin hypercube search methods, all of which, by default, run once at the beginning of an optimization. `ga` and Latin hypercube searches are stochastic, and can search through several basins of attraction on page 1-25.

See Also

More About

- “Search and Poll” on page 6-42
- “Polling Types” on page 6-59
- “Setting Solver Tolerances” on page 6-41

Setting Solver Tolerances

Tolerance refers to how small a parameter, such as a mesh size, can become before the search is halted or changed in some way. You can specify the value of the following tolerances using `optimoptions` or the **Optimize** Live Editor task.

- `MeshTolerance` — When the current mesh size is less than the value of `MeshTolerance`, the algorithm halts.
- `StepTolerance` — After a successful poll, if the distance from the previous best point to the current best point is less than the value of `StepTolerance`, the algorithm halts.
- `FunctionTolerance` — After a successful poll, if the difference between the function value at the previous best point and function value at the current best point is less than the value of `FunctionTolerance`, the algorithm halts.
- `ConstraintTolerance` (not a stopping condition) — The algorithm treats a point to be feasible if nonlinear constraint violation is less than `ConstraintTolerance`.

See Also

More About

- “Set Options” on page 6-57
- “How Pattern Search Polling Works” on page 6-27

Search and Poll

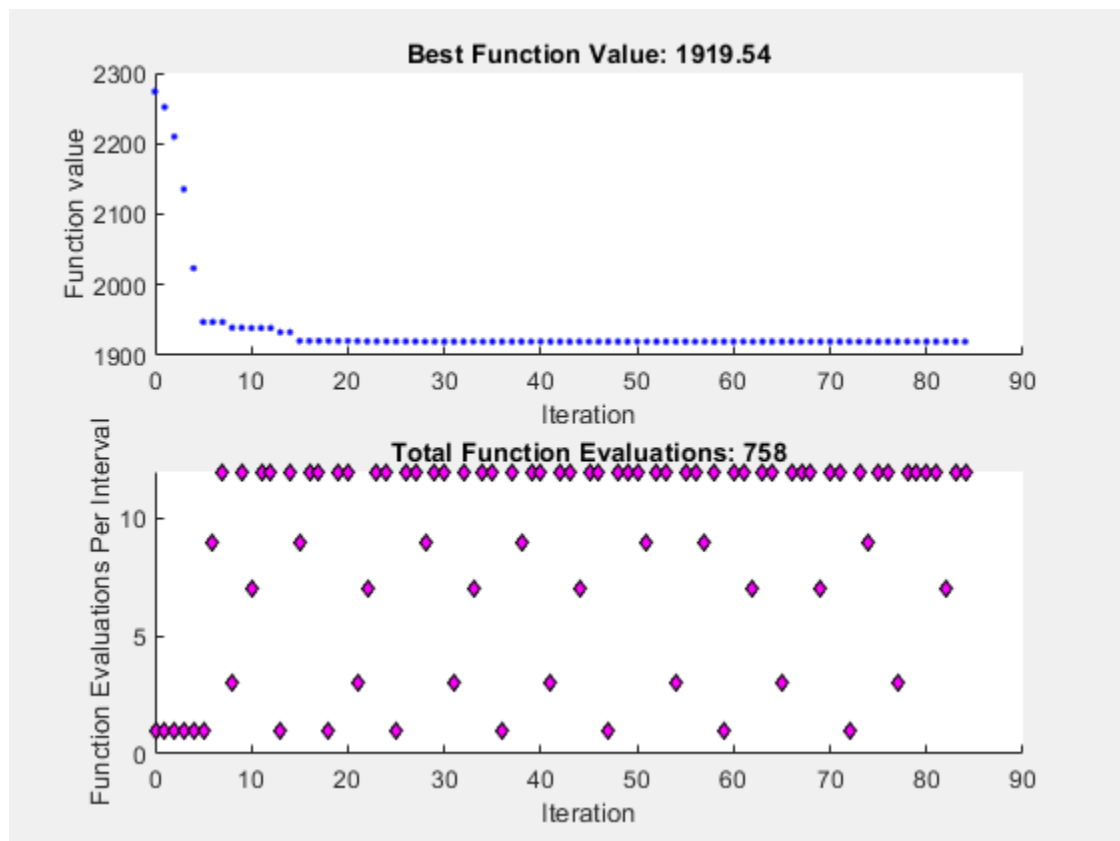
In addition to polling the mesh points, the pattern search algorithm can perform an optional step at every iteration, called search. At each iteration, the search step applies another optimization method to the current point. If this search does not improve the current point, the poll step is performed. The objective function, `lincontest7`, is available when you run this example.

Search Using a Poll Method

The following example illustrates the use of a search method on the problem described in “Constrained Minimization Using `patternsearch` and Optimize Live Editor Task” on page 6-71. In this case, the search method is the GSS Positive Basis 2N poll. For comparison, first run the problem without a search method.

```
x0 = [2 1 0 9 1 0];
Aineq = [-8 7 3 -4 9 0];
bineq = 7;
Aeq = [7 1 8 3 3 3; 5 0 -5 1 -5 8; -2 -6 7 1 1 9; 1 -1 2 -2 3 -3];
beq = [84 62 65 1];
options = optimoptions('patternsearch',...
    'PlotFcn',{@psplotbestf,@psplotfunccount});
[x,fval,exitflag,output] = patternsearch(@lincontest7,x0,...
    Aineq,bineq,Aeq,beq,[],[],[],options);
```

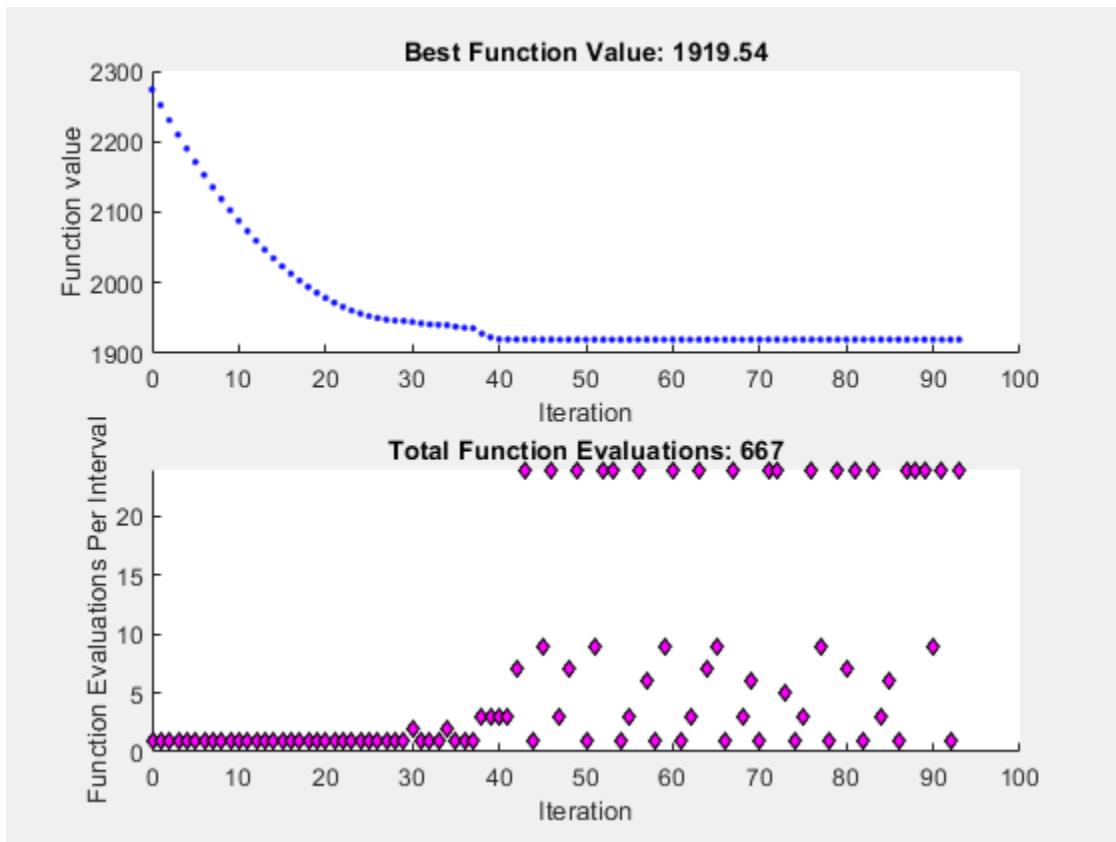
Optimization terminated: mesh size less than options.MeshTolerance.



To use the GSS Positive Basis 2N poll as a search method, change the SearchFcn option.

```
rng default % For reproducibility
options.SearchFcn = @GSSPositiveBasis2N;
[x2,fval2,exitflag2,output2] = patternsearch(@lincontest7,x0,...
    Aineq,bineq,Aeq,beq,[],[],[],options);
```

Optimization terminated: mesh size less than options.MeshTolerance.



Both optimizations reached the same objective function value. Using the search method reduces the number of function evaluations and the number of iterations.

```
table([output.funccount;output2.funccount],[output.iterations;output2.iterations],...
    'VariableNames',["Function Evaluations" "Iterations"],...
    'RowNames',["Without Search" "With Search"])
```

ans=2x2 table

	Function Evaluations	Iterations
Without Search	758	84
With Search	667	93

Search Using a Different Solver

patternsearch takes a long time to minimize Rosenbrock's function. The function is

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2.$$

Rosenbrock's function is described and plotted in “Constrained Nonlinear Problem Using Optimize Live Editor Task or Solver”. The minimum of Rosenbrock's function is 0, attained at the point [1, 1]. Because `patternsearch` is not efficient at minimizing this function, use a different search method to help.

Create the objective function.

```
dejong2fcn = @(x)100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
```

The default maximum number of iterations for `patternsearch` with two variables is 200, and the default maximum number of function evaluations is 4000. Increase these values to `MaxFunctionEvaluations = 5000` and `MaxIterations = 2000`.

```
opts = optimoptions('patternsearch','MaxFunctionEvaluations',5000,'MaxIterations',2000);
```

Run `patternsearch` starting from [-1.9 2].

```
[x,feval,eflag,output] = patternsearch(dejong2fcn,...
    [-1.9,2],[],[],[],[],[],[],[],[],opts);
```

Maximum number of function evaluations exceeded: increase options.MaxFunctionEvaluations.

```
disp(feval)
```

```
0.8560
```

```
disp(output.funccount)
```

```
5000
```

The optimization did not complete even after 5000 function evaluations, and so the result is not very close to the optimal value of 0.

Set the options to use `fminsearch` as the search method, using the default number of function evaluations and iterations.

```
opts = optimoptions('patternsearch',opts,'SearchFcn',@searchneldermead);
```

Rerun the optimization.

```
[x2,feval2,eflag2,output2] = patternsearch(dejong2fcn,...
    [-1.9,2],[],[],[],[],[],[],[],[],opts);
```

Optimization terminated: mesh size less than options.MeshTolerance.

```
disp(feval2)
```

```
4.0686e-10
```

```
disp(output2.funccount)
```

```
291
```

The objective function value at the solution is much better (lower) when using this search method, and the number of function evaluations is much lower. `fminsearch` is more efficient at getting close to the minimum of Rosenbrock's function.

See Also

More About

- “Polling Types” on page 6-59
- “Vectorize the Objective and Constraint Functions” on page 6-83

Nonlinear Constraint Solver Algorithm for Pattern Search

The pattern search algorithm uses the Augmented Lagrangian Pattern Search (ALPS) algorithm to solve nonlinear constraint problems. The optimization problem solved by the ALPS algorithm is

$$\min_x f(x)$$

such that

$$\begin{aligned} c_i(x) &\leq 0, \quad i = 1 \dots m \\ ceq_i(x) &= 0, \quad i = m + 1 \dots mt \\ A \cdot x &\leq b \\ Aeq \cdot x &= beq \\ lb &\leq x \leq ub, \end{aligned}$$

where $c(x)$ represents the nonlinear inequality constraints, $ceq(x)$ represents the equality constraints, m is the number of nonlinear inequality constraints, and mt is the total number of nonlinear constraints.

The ALPS algorithm attempts to solve a nonlinear optimization problem with nonlinear constraints, linear constraints, and bounds. In this approach, bounds and linear constraints are handled separately from nonlinear constraints. A subproblem is formulated by combining the objective function and nonlinear constraint function using the Lagrangian and the penalty parameters. A sequence of such optimization problems are approximately minimized using a pattern search algorithm such that the linear constraints and bounds are satisfied.

Each subproblem solution represents one iteration. The number of function evaluations per iteration is therefore much higher when using nonlinear constraints than otherwise.

A subproblem formulation is defined as

$$\Theta(x, \lambda, s, \rho) = f(x) - \sum_{i=1}^m \lambda_i s_i \log(s_i - c_i(x)) + \sum_{i=m+1}^{mt} \lambda_i ceq_i(x) + \frac{\rho}{2} \sum_{i=m+1}^{mt} ceq_i(x)^2,$$

where

- The components λ_i of the vector λ are nonnegative and are known as Lagrange multiplier estimates
- The elements s_i of the vector s are nonnegative shifts
- ρ is the positive penalty parameter.

The algorithm begins by using an initial value for the penalty parameter (`InitialPenalty`).

The pattern search minimizes a sequence of subproblems, each of which is an approximation of the original problem. Each subproblem has a fixed value of λ , s , and ρ . When the subproblem is minimized to a required accuracy and satisfies feasibility conditions, the Lagrangian estimates are updated. Otherwise, the penalty parameter is increased by a penalty factor (`PenaltyFactor`). This results in a new subproblem formulation and minimization problem. These steps are repeated until the stopping criteria are met.

Each subproblem solution represents one iteration. The number of function evaluations per iteration is therefore much higher when using nonlinear constraints than otherwise.

For a complete description of the algorithm, see the following references:

References

- [1] Kolda, Tamara G., Robert Michael Lewis, and Virginia Torczon. "A generating set direct search augmented Lagrangian algorithm for optimization with a combination of general and linear constraints." Technical Report SAND2006-5315, Sandia National Laboratories, August 2006.
- [2] Conn, A. R., N. I. M. Gould, and Ph. L. Toint. "A Globally Convergent Augmented Lagrangian Algorithm for Optimization with General Constraints and Simple Bounds," *SIAM Journal on Numerical Analysis*, Volume 28, Number 2, pages 545-572, 1991.
- [3] Conn, A. R., N. I. M. Gould, and Ph. L. Toint. "A Globally Convergent Augmented Lagrangian Barrier Algorithm for Optimization with General Inequality Constraints and Simple Bounds," *Mathematics of Computation*, Volume 66, Number 217, pages 261-288, 1997.

See Also

More About

- "Constrained Minimization Using Pattern Search, Solver-Based" on page 6-14
- "Constrained Minimization Using patternsearch and Optimize Live Editor Task" on page 6-71

Custom Plot Function

In this section...

“About Custom Plot Functions” on page 6-48
 “Creating the Custom Plot Function” on page 6-48
 “Set Up the Problem” on page 6-49
 “Run the Optimization with Custom Plot Function” on page 6-49
 “How the Plot Function Works” on page 6-50

About Custom Plot Functions

To use a `patternsearch` plot function other than those included with the software, you can write your own custom plot function that is called at each iteration of the pattern search to create the plot. This example shows how to create a plot function that displays the logarithmic change in the best objective function value from the previous iteration to the current iteration. More plot function details are available in “Plot Options” on page 17-23.

Creating the Custom Plot Function

To create the plot function for this example, copy and paste the following code into a new function file in the MATLAB Editor:

```
function stop = psplotchange(optimvalues, flag)
% PSLOTCHANGE Plots the change in the best objective function
% value from the previous iteration.

% Best objective function value in the previous iteration
persistent last_best

stop = false;
if(strcmp(flag,'init'))
    set(gca,'yscale','log'); %Set up the plot
    hold on;
    xlabel('Iteration');
    ylabel('Log Change in Values');
    title(['Change in Best Function Value']);
end

% Best objective function value in the current iteration
best = min(optimvalues.fval);

% Set last_best to best
if optimvalues.iteration == 0
    last_best = best;
else
    %Change in objective function value
    change = last_best - best;
    plot(optimvalues.iteration, change, 'r');
end
```

Save the file as `psplotchange.m` in a folder on the MATLAB path. The code is explained in “How the Plot Function Works” on page 6-50.

Set Up the Problem

The problem is the same as “Constrained Minimization Using patternsearch and Optimize Live Editor Task” on page 6-71. To set up the problem:

- 1 Enter the following at the MATLAB command line:

```
x0 = [2 1 0 9 1 0];
Aineq = [-8 7 3 -4 9 0];
bineq = 7;
Aeq = [7 1 8 3 3 3; 5 0 -5 1 -5 8; -2 -6 7 1 1 9; 1 -1 2 -2 3 -3];
beq = [84 62 65 1];
H = [36 17 19 12 8 15;
     17 33 18 11 7 14;
     19 18 43 13 8 16;
     12 11 13 18 6 11;
     8 7 8 6 9 8;
     15 14 16 11 8 29];
```

```
f = [ 20 15 21 18 29 24 ]';
```

```
F = @(x)0.5*x'*H*x + f'*x;
```

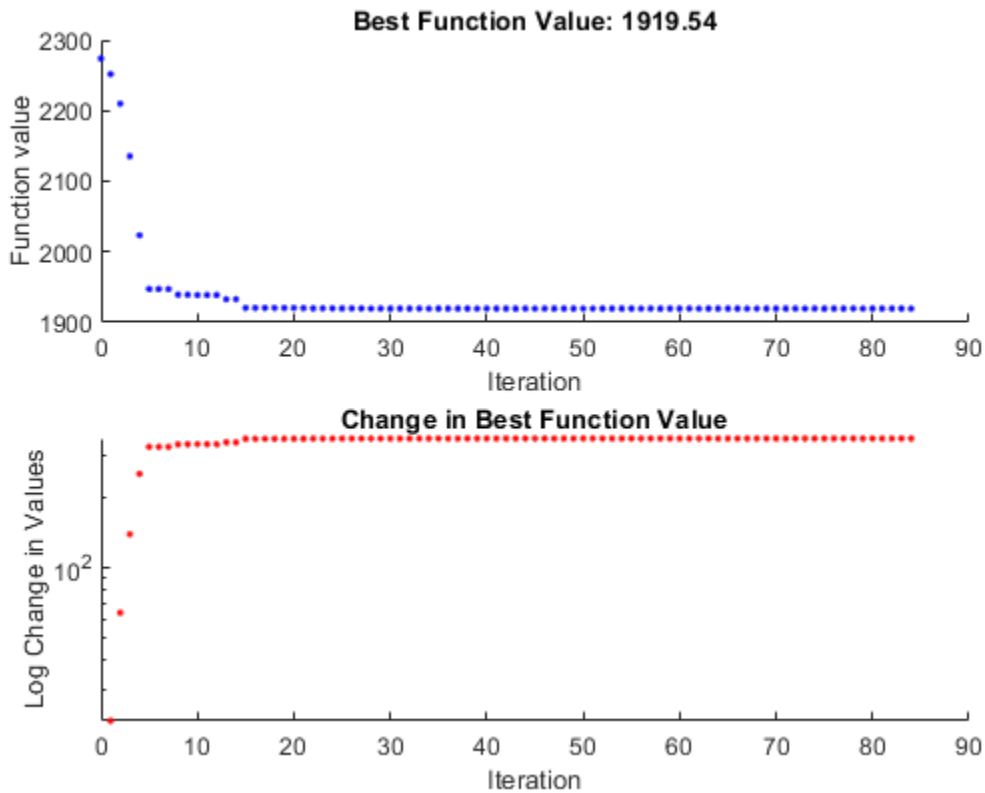
- 2 Because this is a linearly constrained problem, set the `PollMethod` option to `'GSSPositiveBasis2N'`. Include both the `@psplotbestf` built-in plot function and the custom plot function `@psplotchange` in the options.

```
options = optimoptions('patternsearch',...
    'PlotFcn',{@psplotbestf,@psplotchange},...
    'PollMethod','GSSPositiveBasis2N');
```

Run the Optimization with Custom Plot Function

Run the example by calling `patternsearch` starting from `x0`.

```
[x,fval] = patternsearch(F,x0,...
    Aineq,bineq,Aeq,beq,[],[],[],options);
```



Because the scale of the y-axis in the lower custom plot is logarithmic, the plot shows only changes that are greater than 0.

How the Plot Function Works

The plot function uses information contained in the following structures.

- `optimvalues` — Current state of the solver, a structure
- `flag` — Current status of the algorithm, a character vector

The most important statements of the custom plot function, `psplotchange.m`, are summarized in the following table.

Custom Plot Function Statements

Statement	Description
<code>persistent last_best</code>	Creates the persistent variable <code>last_best</code> , the best objective function value in the previous generation. Persistent variables are preserved over multiple calls to the plot function.
<code>set(gca, 'Yscale', 'log')</code>	Sets up the plot before the algorithm starts.
<code>best = min(optimvalues.fval)</code>	Sets <code>best</code> equal to the minimum objective function value. The field <code>optimvalues.fval</code> contains the objective function value in the current iteration. The variable <code>best</code> is the minimum objective function value. For a complete description of the fields of the structure <code>optimvalues</code> , see “Structure of the Plot Functions” on page 17-8.
<code>change = last_best - best</code>	Sets the variable <code>change</code> to the best objective function value at the previous iteration minus the best objective function value in the current iteration.
<code>plot(optimvalues.iteration, change, 'r')</code>	Plots the variable <code>change</code> at the current objective function value, for the current iteration contained in <code>optimvalues.iteration</code> .

See Also**More About**

- “Plot Options” on page 17-23

Pattern Search Climbs Mount Washington

This example shows visually how pattern search optimizes a function. The function is the height of the terrain near Mount Washington, as a function of the x-y location. In order to find the top of Mount Washington, we minimize the objective function that is the negative of the height. (The Mount Washington in this example is the highest peak in the northeastern United States.)

The US Geological Survey provides data on the height of the terrain as a function of the x-y location on a grid. In order to be able to evaluate the height at an arbitrary point, the objective function interpolates the height from nearby grid points.

It would be faster, of course, to simply find the maximum value of the height as specified on the grid, using the `max` function. The point of this example is to show how the pattern search algorithm operates; it works on functions defined over continuous regions, not just grid points. Also, if it is computationally expensive to evaluate the objective function, then performing this evaluation on a complete grid, as required by the `max` function, will be much less efficient than using the pattern search algorithm, which samples a small subset of grid points.

How Pattern Search Works

Pattern search finds a local minimum of an objective function by the following method, called polling. In this description, words describing pattern search quantities are in bold. The search starts at an initial point, which is taken as the **current point** in the first step:

1. Generate a **pattern** of points, typically plus and minus the coordinate directions, times a **mesh size**, and center this pattern on the **current point**.
2. Evaluate the objective function at every point in the **pattern**.
3. If the minimum objective in the **pattern** is lower than the value at the **current point**, then the poll is **successful**, and the following happens:
 - 3a. The minimum point found becomes the **current point**.
 - 3b. The **mesh size** is doubled.
 - 3c. The algorithm proceeds to Step 1.
4. If the poll is not **successful**, then the following happens:
 - 4a. The **mesh size** is halved.
 - 4b. If the **mesh size** is below a threshold, the iterations stop.
 - 4c. Otherwise, the **current point** is retained, and the algorithm proceeds at Step 1.

This simple algorithm, with some minor modifications, provides a robust and straightforward method for optimization. It requires no gradients of the objective function. It lends itself to constraints, too, but this example and description deal only with unconstrained problems.

Preparing the Pattern Search

To prepare the pattern search, load the data in `mtWashington.mat`, which contains the USGS data on a 472-by-345 grid. The elevation, Z , is given in feet. The vectors x and y contain the base values of

the grid spacing in the east and north directions respectively. The data file also contains the starting point for the search, X0.

```
load mtWashington
```

There are three MATLAB® files that perform the calculation of the objective function, and the plotting routines. They are:

1. `terrainfun`, which evaluates the negative of height at any x-y position. `terrainfun` uses the MATLAB function `interp2` to perform two-dimensional linear interpolation. It takes the Z data and enables evaluation of the negative of the height at all x-y points.
2. `psoutputwashington`, which draws a 3-d rendering of Mt. Washington. In addition, as the run progresses, it draws spheres around each point that is better (higher) than previously-visited points.
3. `psplotwashington`, which draws a contour map of Mt. Washington, and monitors a slider that controls the speed of the run. It shows where the pattern search algorithm looks for optima by drawing + signs at those points. It also draws spheres around each point that is better than previously-visited points.

In the example, `patternsearch` uses `terrainfun` as its objective function, `psoutputwashington` as an output function, and `psplotwashington` as a plot function. We prepare the functions to be passed to `patternsearch` in anonymous function syntax:

```
mtWashObjectiveFcn = @(xx) terrainfun(xx, x, y, Z);
mtWashOutputFcn    = @(xx,arg1,arg2) psoutputwashington(xx,arg1,arg2, x, y, Z);
mtWashPlotFcn      = @(xx,arg1) psplotwashington(xx,arg1, x, y, Z);
```

Pattern Search Options Settings

Next, we create options for pattern search. This set of options has the algorithm halt when the mesh size shrinks below 1, keeps the mesh unscaled (the same size in each direction), sets the initial mesh size to 10, and sets the output function and plot function:

```
options = optimoptions(@patternsearch, 'MeshTolerance',1, 'ScaleMesh',false, ...
    'InitialMeshSize',10, 'UseCompletePoll',true, 'PlotFcn',mtWashPlotFcn, ...
    'OutputFcn',mtWashOutputFcn, 'UseVectorized',true);
```

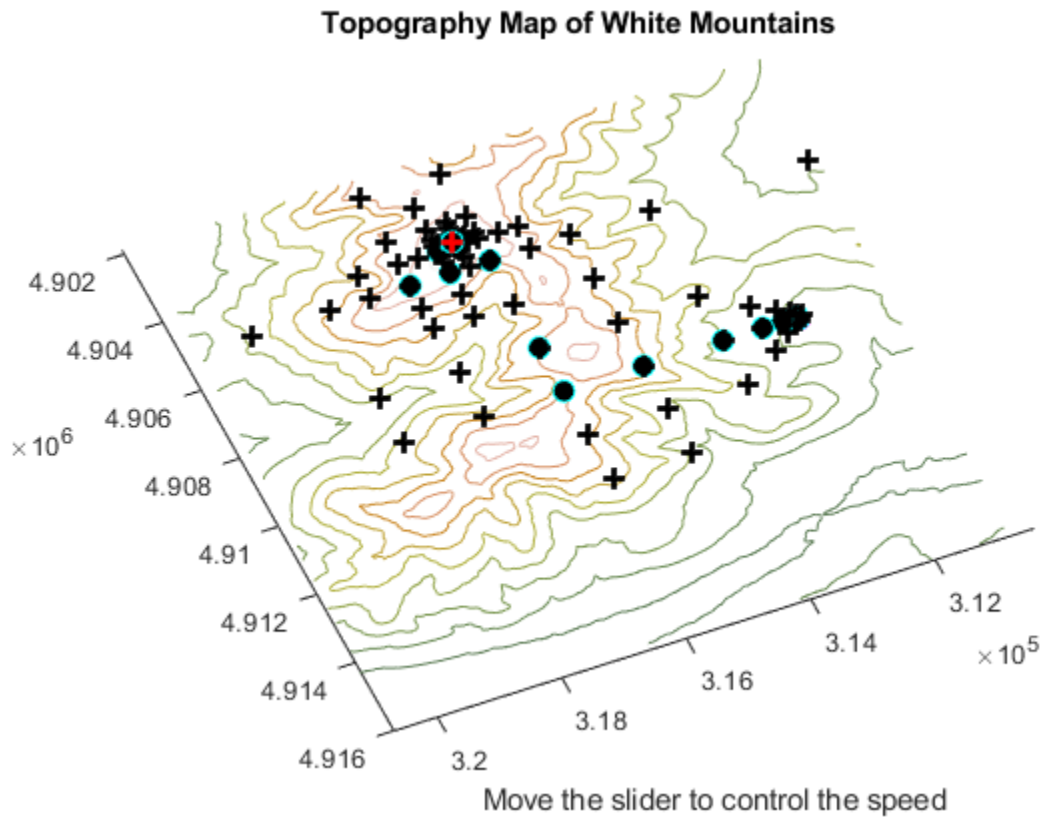
Observing the Progress of Pattern Search

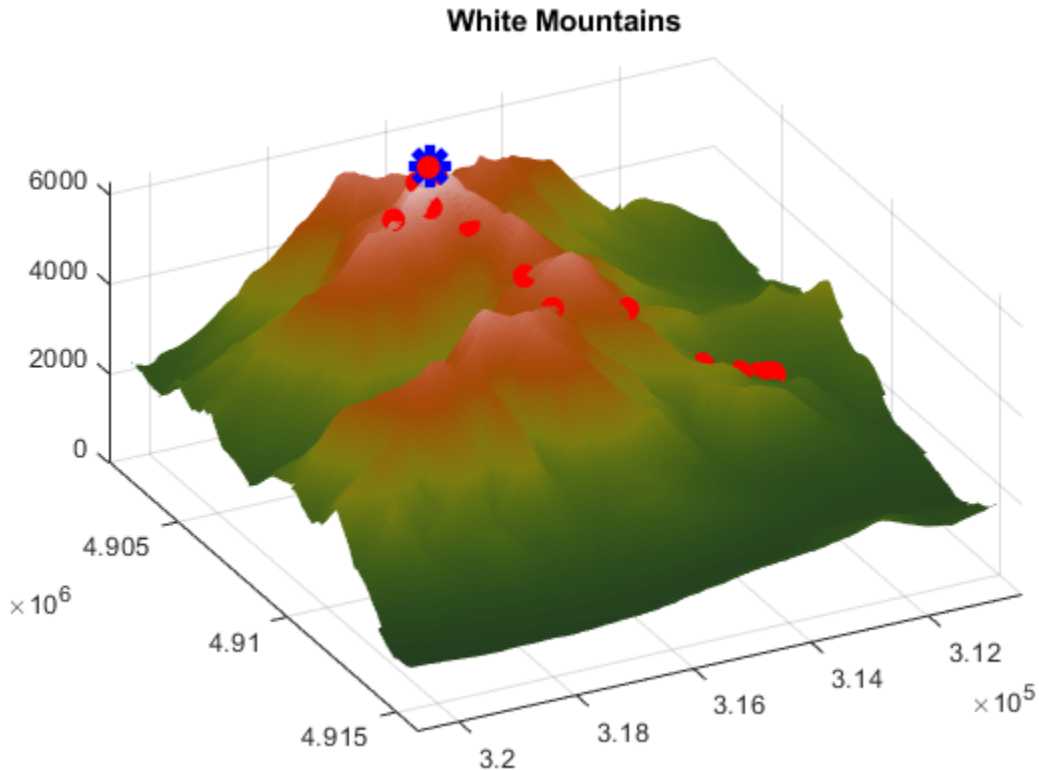
When you run this example you see two windows. One shows the points the pattern search algorithm chooses on a two-dimensional contour map of Mount Washington. This window has a slider that controls the delay between iterations of the algorithm (when it returns to Step 1 in the description of how pattern search works). Set the slider to a low position to speed the run, or to a high position to slow the run.

The other window shows a three-dimensional plot of Mount Washington, along with the steps the pattern search algorithm makes. You can rotate this plot while the run progresses to get different views.

```
[xfinal ffinal] = patternsearch(mtWashObjectiveFcn,X0,[],[],[],[],[], ...
    [],[],options)
```

```
Optimization terminated: mesh size less than options.MeshTolerance.
```





```
xfinal = 1x2
```

```
    316130    4904295
```

```
ffinal = -6280
```

The final point, `xfinal`, shows where the pattern search algorithm finished; this is the x-y location of the top of Mount Washington. The final objective function, `ffinal`, is the negative of the height of Mount Washington, 6280 feet. (This should be 6288 feet according to the Mount Washington Observatory).

Examine the files `terrainfun.m`, `psoutputwashington.m`, and `psplotwashington.m` to see how the interpolation and graphics work.

There are many options available for the pattern search algorithm. For example, the algorithm can take the first point it finds that is an improvement, rather than polling all the points in the pattern. It can poll the points in various orders. And it can use different patterns for the poll, both deterministic and random. Consult the Global Optimization Toolbox User's Guide for details.

See Also

More About

- “How Pattern Search Polling Works” on page 6-27

- “Custom Plot Function” on page 6-48

Set Options

You can specify any available `patternsearch` options by passing `options` as an input argument to `patternsearch` using the syntax

```
[x,fval] = patternsearch(@fitnessfun,nvars, ...
    A,b,Aeq,beq,lb,ub,nonlcon,options)
```

Pass in empty brackets `[]` for any constraints that do not appear in the problem.

Create options using the `optimoptions` function.

```
options = optimoptions(@patternsearch)
```

```
options =
```

```
patternsearch options:
```

```
Set properties:
```

```
No options set.
```

```
Default properties:
```

```
AccelerateMesh: 0
ConstraintTolerance: 1.0000e-06
    Display: 'final'
FunctionTolerance: 1.0000e-06
InitialMeshSize: 1
MaxFunctionEvaluations: '2000*numberOfVariables'
    MaxIterations: '100*numberOfVariables'
    MaxTime: Inf
MeshContractionFactor: 0.5000
MeshExpansionFactor: 2
MeshTolerance: 1.0000e-06
    OutputFcn: []
    PlotFcn: []
    PollMethod: 'GPSPositiveBasis2N'
PollOrderAlgorithm: 'consecutive'
    ScaleMesh: 1
    SearchFcn: []
StepTolerance: 1.0000e-06
UseCompletePoll: 0
UseCompleteSearch: 0
UseParallel: 0
UseVectorized: 0
```

The `patternsearch` function uses these default values if you do not pass in `options` as an input argument.

The value of each option is stored in a field of `options`, such as `options.MeshExpansionFactor`. You can display any of these values by entering `options` followed by the name of the field. For example, to display the mesh expansion factor for the pattern search, enter

```
options.MeshExpansionFactor
```

```
ans =
    2
```

To create `options` with a field value that is different from the default, use `optimoptions`. For example, to change the mesh expansion factor to 3 instead of its default value 2, enter

```
options = optimoptions('patternsearch','MeshExpansionFactor',3);
```

This creates `options` with all values set to defaults except for `MeshExpansionFactor`, which is set to 3.

If you now call `patternsearch` with the argument `options`, the pattern search uses a mesh expansion factor of 3.

If you subsequently decide to change another field in `options`, such as setting `PlotFcn` to `@psplotmeshsize`, which plots the mesh size at each iteration, call `optimoptions` with the syntax

```
options = optimoptions(options,'PlotFcn',@psplotmeshsize)
```

This preserves the current values of all fields of `options` except for `PlotFcn`, which is changed to `@psplotmeshsize`. Note that if you omit the `options` input argument, `optimoptions` resets `MeshExpansionFactor` to its default value, which is 2.

You can also set both `MeshExpansionFactor` and `PlotFcn` with the single command

```
options = optimoptions('patternsearch','MeshExpansionFactor',3,'PlotFcn',@psplotmeshsize)
```

See Also

`patternsearch` | `optimoptions`

More About

- “Pattern Search Options” on page 17-7

Polling Types

In this section...

“Using a Complete Poll in a Generalized Pattern Search” on page 6-59

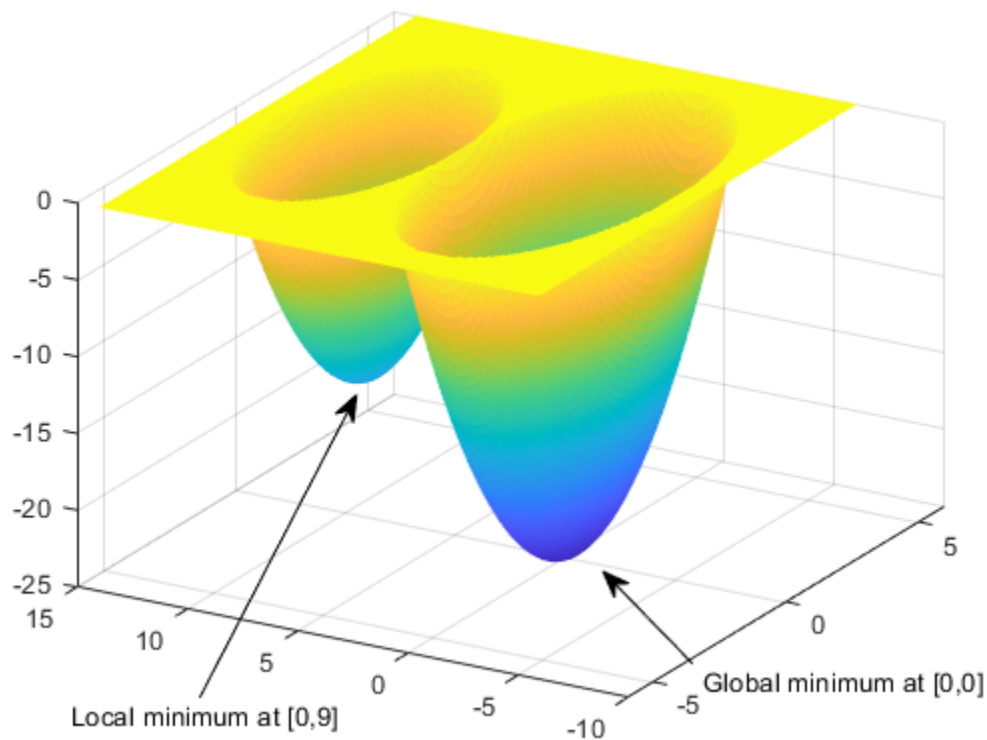
“Compare the Efficiency of Poll Options” on page 6-61

Using a Complete Poll in a Generalized Pattern Search

As an example, consider the following function.

$$f(x_1, x_2) = \begin{cases} x_1^2 + x_2^2 - 25 & \text{for } x_1^2 + x_2^2 \leq 25 \\ x_1^2 + (x_2 - 9)^2 - 16 & \text{for } x_1^2 + (x_2 - 9)^2 \leq 16 \\ 0 & \text{otherwise.} \end{cases}$$

The following figure shows a plot of the function.



The global minimum of the function occurs at (0, 0), where its value is -25. However, the function also has a local minimum at (0, 9), where its value is -16.

To create a file that computes the function, copy and paste the following code into a new file in the MATLAB Editor.

```

function z = poll_example(x)
if x(1)^2 + x(2)^2 <= 25
    z = x(1)^2 + x(2)^2 - 25;
elseif x(1)^2 + (x(2) - 9)^2 <= 16
    z = x(1)^2 + (x(2) - 9)^2 - 16;
else z = 0;
end

```

Save the file as `poll_example.m` in a folder on the MATLAB path.

To run a pattern search on the function, enter the following.

```

options = optimoptions('patternsearch','Display','iter');
[x,fval] = patternsearch(@poll_example,[0,5],...
    [],[],[],[],[],[],[],options)

```

MATLAB returns a table of iterations and the solution.

```

x =
    0    9

fval =
   -16

```

The algorithm begins by evaluating the function at the initial point, $f(0, 5) = 0$. The poll evaluates the following during its first iterations.

$$f((0, 5) + (1, 0)) = f(1, 5) = 0$$

$$f((0, 5) + (0, 1)) = f(0, 6) = -7$$

As soon as the search polls the mesh point (0, 6), at which the objective function value is less than at the initial point, it stops polling the current mesh and sets the current point at the next iteration to (0, 6). Consequently, the search moves toward the local minimum at (0, 9) at the first iteration. You see this by looking at the first two lines of the command line display.

Iter	f-count	f(x)	MeshSize	Method
0	1	0	1	
1	3	-7	2	Successful Poll

Note that the pattern search performs only two evaluations of the objective function at the first iteration, increasing the total function count from 1 to 3.

Next, set `UseCompletePoll` to `true` and rerun the optimization.

```

options.UseCompletePoll = true;
[x,fval] = patternsearch(@poll_example,[0,5],...
    [],[],[],[],[],[],[],options);

```

This time, the pattern search finds the global minimum at (0, 0). The difference between this run and the previous one is that with `UseCompletePoll` set to `true`, at the first iteration the pattern search polls all four mesh points.

$$f((0, 5) + (1, 0)) = f(1, 5) = 0$$

$$f(0, 5) + (0, 1) = f(0, 6) = -6$$

$$f(0, 5) + (-1, 0) = f(-1, 5) = 0$$

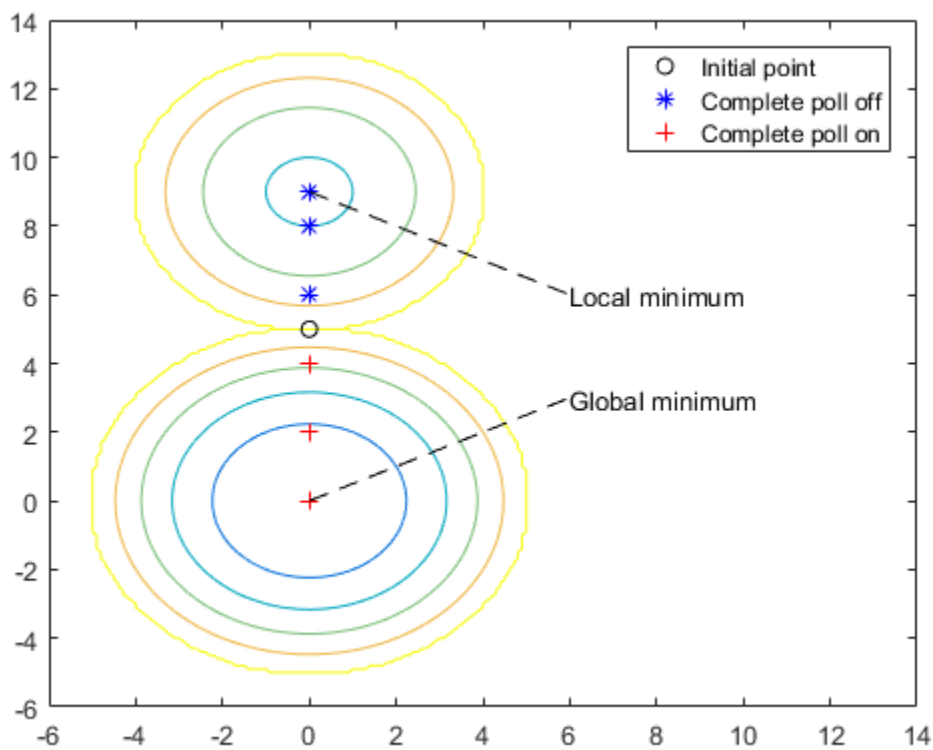
$$f(0, 5) + (0, -1) = f(0, 4) = -9$$

Because the last mesh point has the lowest objective function value, the pattern search selects it as the current point at the next iteration. The first two lines of the command-line display show this.

Iter	f-count	f(x)	MeshSize	Method
0	1	0	1	
1	5	-9	2	Successful Poll

In this case, the objective function is evaluated four times at the first iteration. As a result, the pattern search moves toward the global minimum at (0, 0).

The following figure compares the sequence of points returned when **Complete poll** is set to 0ff with the sequence when **Complete poll** is 0n.



Compare the Efficiency of Poll Options

This example shows how several poll options interact in terms of iterations and total function evaluations. The main results are:

- GSS is more efficient than GPS or MADS for linearly constrained problems.

- Whether setting `UseCompletePoll` to `true` increases efficiency or decreases efficiency is unclear, although it affects the number of iterations.
- Similarly, whether having a 2N poll is more or less efficient than having an Np1 poll is also unclear. The most efficient poll is GSS Positive Basis Np1 with **Complete poll** set to on. The least efficient is MADS Positive Basis Np1 with **Complete poll** set to on.

Note The efficiency of an algorithm depends on the problem. GSS is efficient for linearly constrained problems. However, predicting the efficiency implications of the other poll options is difficult, as is knowing which poll type works best with other constraints.

Problem setup

The problem is the same as in “Solve Using patternsearch in Optimize Live Editor Task” on page 6-72. This linearly constrained problem has a quadratic objective function.

- 1 Enter the following into your MATLAB workspace.

```
x0 = [2 1 0 9 1 0];
Aineq = [-8 7 3 -4 9 0];
bineq = 7;
Aeq = [7 1 8 3 3 3; 5 0 -5 1 -5 8; -2 -6 7 1 1 9; 1 -1 2 -2 3 -3];
beq = [84 62 65 1];
H = [36 17 19 12 8 15;
     17 33 18 11 7 14;
     19 18 43 13 8 16;
     12 11 13 18 6 11;
     8 7 8 6 9 8;
     15 14 16 11 8 29];
```

```
f = [ 20 15 21 18 29 24 ]';
```

```
fun = @(x)0.5*x'*H*x + f'*x;
```

- 2 Set the initial options and objective function.

```
options = optimoptions('patternsearch',...
    'PollMethod','GPSPositiveBasis2N',...
    'PollOrderAlgorithm','consecutive',...
    'UseCompletePoll',false);
```

- 3 Run the optimization, naming the output structure `outputgps2noff`.

```
[x,fval,exitflag,outputgps2noff] = patternsearch(fun,x0,...
    Aineq,bineq,Aeq,beq,[],[],[],options);
```

- 4 Set options to use a complete poll.

```
options.UseCompletePoll = true;
```

- 5 Run the optimization, naming the output structure `outputgps2non`.

```
[x,fval,exitflag,outputgps2non] = patternsearch(fun,x0,...
    Aineq,bineq,Aeq,beq,[],[],[],options);
```

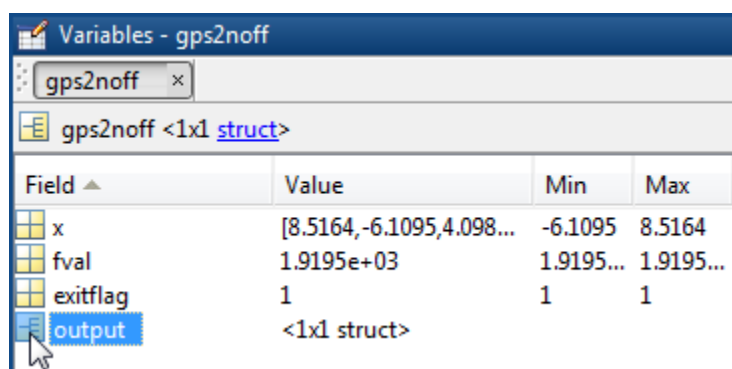
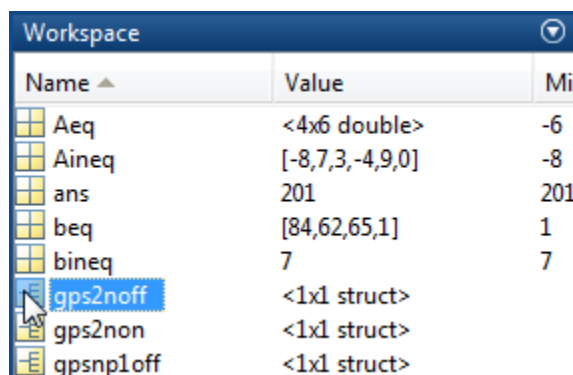
- 6 Continue in a like manner to create output structures for the other poll methods with `UseCompletePoll` set `true` and `false`: `outputgss2noff`, `outputgss2non`, `outputgssnploff`, `outputgssnp1on`, `outputmads2noff`, `outputmads2non`, `outputmadsnploff`, and `outputmadsnp1on`.

Examine the Results

You have the results of 12 optimization runs. The following table shows the efficiency of the runs, measured in total function counts and in iterations. Your MADS results could differ, since MADS is a stochastic algorithm.

Algorithm	Function Count	Iterations
GPS2N, complete poll off	1462	136
GPS2N, complete poll on	1396	96
GPSNp1, complete poll off	864	118
GPSNp1, complete poll on	1007	104
GSS2N, complete poll off	758	84
GSS2N, complete poll on	889	74
GSSNp1, complete poll off	533	94
GSSNp1, complete poll on	491	70
MADS2N, complete poll off	922	162
MADS2N, complete poll on	2285	273
MADSNp1, complete poll off	1155	201
MADSNp1, complete poll on	1651	201

To obtain, say, the first row in the table, enter `gps2noff.output.funccount` and `gps2noff.output.iterations`. You can also examine options in the Variables editor by double-clicking the options in the Workspace Browser, and then double-clicking the output structure.



Field	Value	Min	Max
function	@lincontest7		
problemtype	'linearconstraints'		
pollmethod	'gpspositivebasis2n'		
searchmethod	[]		
iterations	136	136	136
funccount	1462	1462	1462
meshsize	9.5367e-07	9.5367...	9.5367...
maxconstraint	9.9945e-04	9.9945...	9.9945...
message	'Optimization termin...		

Alternatively, you can access the data from the output structures.

```
[outputgps2noff.funccount,outputgps2noff.iterations]
```

```
ans =
```

```
1462    136
```

The main results gleaned from the table are:

- Setting `UseCompletePoll` to `true` generally lowers the number of iterations for GPS and GSS, but the change in number of function evaluations is unpredictable.
- Setting `UseCompletePoll` to `true` does not necessarily change the number of iterations for MADs, but substantially increases the number of function evaluations.
- The most efficient algorithm/options settings, with efficiency meaning lowest function count:
 - 1 'GSSPositiveBasisNp1' with `UseCompletePoll` set to `true` (function count 491)
 - 2 'GSSPositiveBasisNp1' with `UseCompletePoll` set to `false` (function count 533)
 - 3 'GSSPositiveBasis2N' with `UseCompletePoll` set to `false` (function count 758)
 - 4 'GSSPositiveBasis2N' with `UseCompletePoll` set to `true` (function count 889)

The other poll methods had function counts exceeding 900.

- For this problem, the most efficient poll is 'GSSPositiveBasisNp1' with `UseCompletePoll` set to `true`, although the `UseCompletePoll` setting makes only a small difference. The least efficient poll is 'MADSPositiveBasis2N' with `UseCompletePoll` set to `true`. In this case, the `UseCompletePoll` setting makes a substantial difference.

See Also

More About

- "How Pattern Search Polling Works" on page 6-27
- "Searching and Polling" on page 6-37

- “Search and Poll” on page 6-42

Set Mesh Options

In this section...

“Mesh Expansion and Contraction” on page 6-66

“Mesh Accelerator” on page 6-69

Mesh Expansion and Contraction

The `MeshExpansionFactor` and `MeshContractionFactor` options control how much the mesh size is expanded or contracted at each iteration. With the default `MeshExpansionFactor` value of 2, the pattern search multiplies the mesh size by 2 after each successful poll. With the default `MeshContractionFactor` value of 0.5, the pattern search multiplies the mesh size by 0.5 after each unsuccessful poll.

You can view the expansion and contraction of the mesh size during the pattern search by setting `@psplotmeshsize` as the `PlotFcn` option. To also display the values of the mesh size and objective function at the command line, set the `Display` option to `'iter'`.

For example, set up the problem described in “Constrained Minimization Using patternsearch and Optimize Live Editor Task” on page 6-71 as follows:

- 1 Enter the following at the command line:

```
x0 = [2 1 0 9 1 0];
Aineq = [-8 7 3 -4 9 0];
bineq = 7;
Aeq = [7 1 8 3 3 3; 5 0 -5 1 -5 8; -2 -6 7 1 1 9; 1 -1 2 -2 3 -3];
beq = [84 62 65 1];
H = [36 17 19 12 8 15;
     17 33 18 11 7 14;
     19 18 43 13 8 16;
     12 11 13 18 6 11;
     8 7 8 6 9 8;
     15 14 16 11 8 29];

f = [ 20 15 21 18 29 24 ]';

F = @(x)0.5*x'*H*x + f'*x;
```

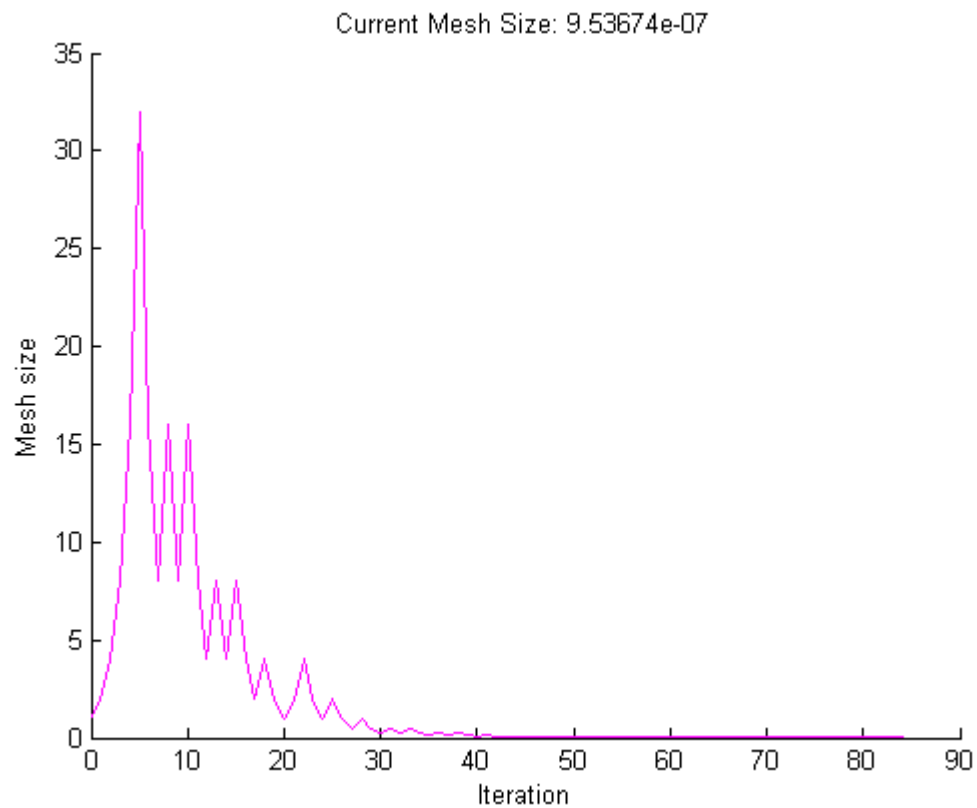
- 2 Create options to use the `GSSPositiveBasis2N` poll method, give iterative display, and plot the mesh size.

```
options = optimoptions('patternsearch',...
    'PollMethod','GSSPositiveBasis2N',...
    'PlotFcn',@psplotmeshsize,...
    'Display','iter');
```

- 3 Run the optimization.

```
[x,fval,exitflag,output] = patternsearch(F,x0,...
    Aineq,bineq,Aeq,beq,[],[],[],options);
```

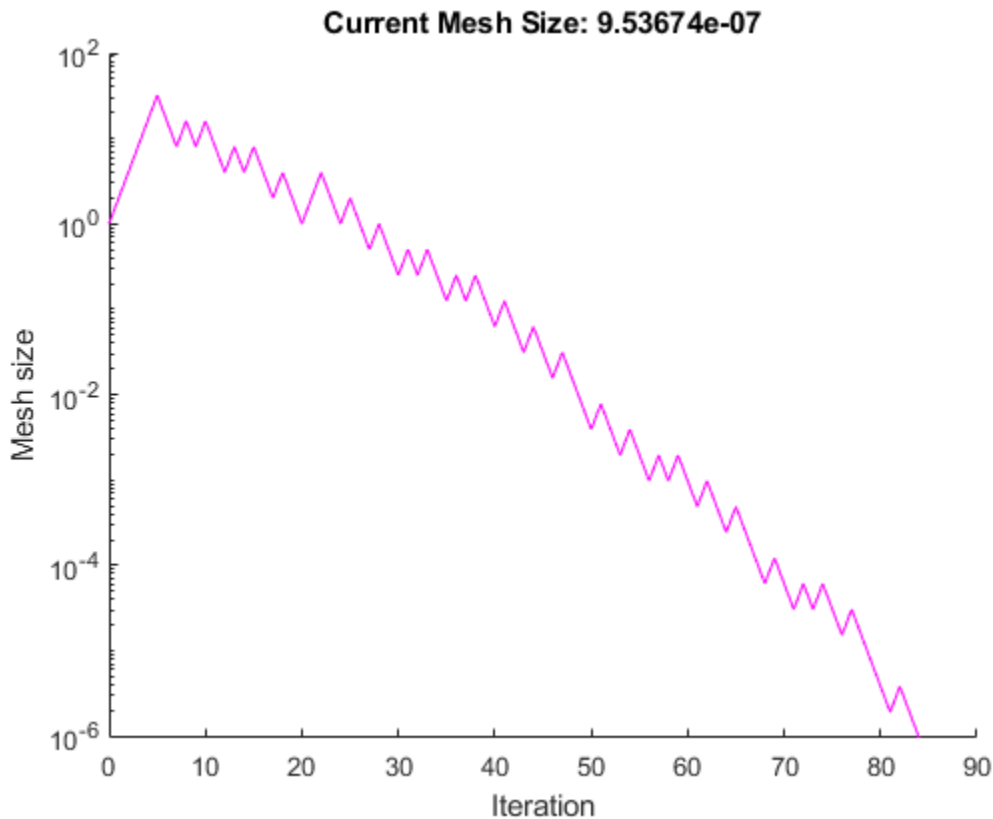
4



To see the changes in mesh size more clearly, change the y-axis to logarithmic scaling as follows:

- 1 Select **Axes Properties** from the **Edit** menu in the plot window.
- 2 In the Properties Editor, select the **Rulers** tab.
- 3 Set **YScale** to **Log**.

Updating these settings in the MATLAB Property Editor shows the plot in the following figure.



The first 5 iterations result in successful polls, so the mesh sizes increase steadily during this time. You can see that the first unsuccessful poll occurs at iteration 6 by looking at the command-line display.

Iter	f-count	f(x)	MeshSize	Method
0	1	2273.76	1	
1	2	2251.69	2	Successful Poll
2	3	2209.86	4	Successful Poll
3	4	2135.43	8	Successful Poll
4	5	2023.48	16	Successful Poll
5	6	1947.23	32	Successful Poll
6	15	1947.23	16	Refine Mesh

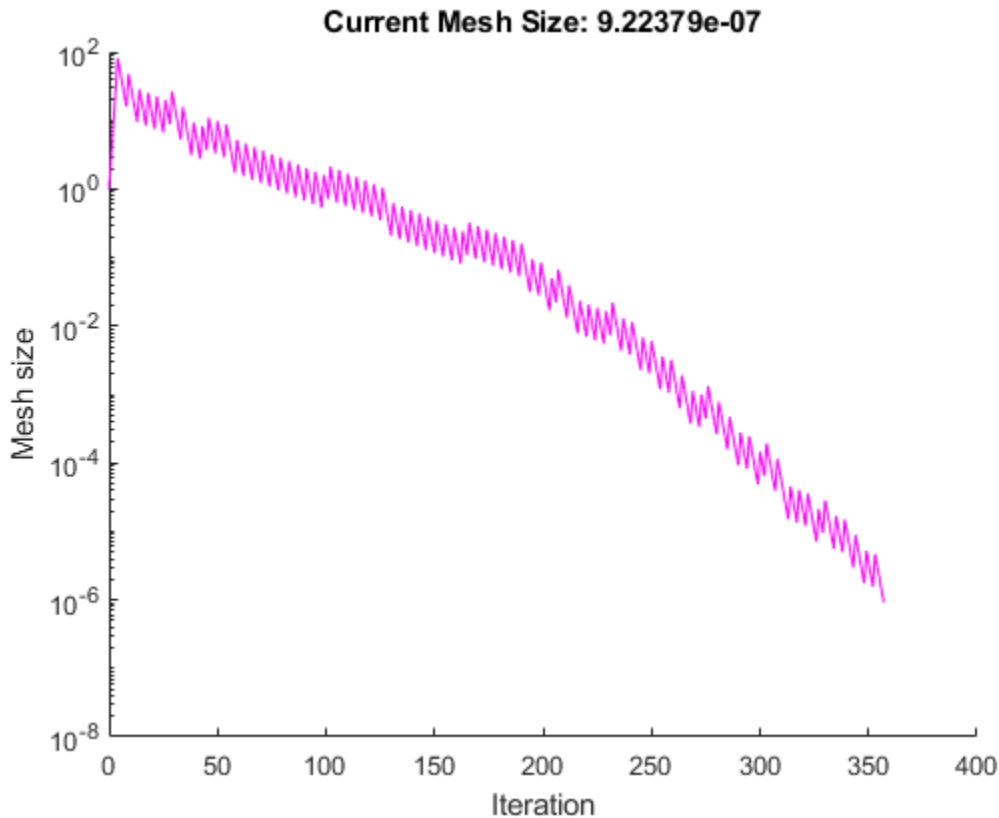
Note that at iteration 5, which is successful, the mesh size doubles for the next iteration. But at iteration 6, which is unsuccessful, the mesh size is multiplied 0.5.

To see how `MeshExpansionFactor` and `MeshContractionFactor` affect the pattern search, set `MeshExpansionFactor` to 3.0 and set `MeshContractionFactor` to 2/3.

```
options = optimoptions(options, 'MeshExpansionFactor', 3.0, ...
    'MeshContractionFactor', 2/3);
[x, fval, exitflag, output] = patternsearch(F, x0, ...
    Aineq, bineq, Aeq, beq, [], [], [], options);
```

The final objective function value is approximately the same as with the previous settings, but the solver takes longer to reach that point.

When you change the scaling of the y-axis to logarithmic, the mesh size plot appears as shown in the following figure.



Note that the mesh size increases faster with `MeshExpansionFactor` set to 3.0, as compared with the default value of 2.0, and decreases more slowly with `MeshContractionFactor` set to 2/3, as compared with the default value of 0.5.

Mesh Accelerator

The mesh accelerator can make a pattern search converge faster to an optimal point by reducing the number of iterations required to reach the mesh tolerance. When the mesh size is below a certain value, the pattern search contracts the mesh size by a factor smaller than the `MeshContractionFactor` factor. Mesh accelerator applies only to the GPS and GSS algorithms.

Note For best results, use the mesh accelerator for problems in which the objective function is not too steep near the optimal point, or you might lose some accuracy. For differentiable problems, this means that the absolute value of the derivative is not too large near the solution.

To use the mesh accelerator, set the `AccelerateMesh` option to `true`.

For example, set up the problem described in “Constrained Minimization Using patternsearch and Optimize Live Editor Task” on page 6-71 as follows:

- 1 Enter the following at the command line:

```
x0 = [2 1 0 9 1 0];
Aineq = [-8 7 3 -4 9 0];
bineq = 7;
Aeq = [7 1 8 3 3 3; 5 0 -5 1 -5 8; -2 -6 7 1 1 9; 1 -1 2 -2 3 -3];
beq = [84 62 65 1];
H = [36 17 19 12 8 15;
     17 33 18 11 7 14;
     19 18 43 13 8 16;
     12 11 13 18 6 11;
     8 7 8 6 9 8;
     15 14 16 11 8 29];

f = [ 20 15 21 18 29 24 ]';
```

```
F = @(x)0.5*x'*H*x + f'*x;
```

- 2 Create options, including the mesh accelerator.

```
options = optimoptions('patternsearch',...
    'PollMethod','GSSPositiveBasis2N',...
    'Display','iter','AccelerateMesh',true);
```

- 3 Run the optimization.

```
[x,fval,exitflag,output] = patternsearch(F,x0,...
    Aineq,bineq,Aeq,beq,[],[],[],options);
```

`patternsearch` completes in 78 iterations, compared to 84 iterations when the mesh accelerator is not on. You can see the effect of the mesh accelerator in the iterative display. Run the example with and without mesh acceleration. The mesh sizes are the same until iteration 70, but differ at iteration 71. The MATLAB Command Window displays the following lines for iterations 70 and 71 without the accelerator.

Iter	f-count	f(x)	MeshSize	Method
70	618	1919.54	6.104e-05	Refine Mesh
71	630	1919.54	3.052e-05	Refine Mesh

Note that the mesh size is multiplied by 0.5, the default value of `MeshContractionFactor`.

For comparison, the Command Window displays the following lines for the same iteration numbers with the accelerator.

Iter	f-count	f(x)	MeshSize	Method
70	618	1919.54	6.104e-05	Refine Mesh
71	630	1919.54	1.526e-05	Refine Mesh

In this case the mesh size is multiplied by 0.25.

See Also

More About

- “Effects of Pattern Search Options” on page 6-18
- “Pattern Search Options” on page 17-7
- “How Pattern Search Polling Works” on page 6-27

Constrained Minimization Using patternsearch and Optimize Live Editor Task

In this section...

“Problem Description” on page 6-71

“Solve Using patternsearch in Optimize Live Editor Task” on page 6-72

“Solve Using patternsearch at the Command Line” on page 6-78

This example shows how to solve a constrained minimization problem using both the **Optimize** Live Editor task, which offers a visual approach, and the command line.

Problem Description

The problem involves using linear and nonlinear constraints when minimizing a nonlinear function with patternsearch. The objective function is

$$F(x) = \frac{1}{2}x^T H x + f^T x,$$

where

```
H = [36 17 19 12  8 15;
      17 33 18 11  7 14;
      19 18 43 13  8 16;
      12 11 13 18  6 11;
       8  7  8  6  9  8;
      15 14 16 11  8 29];
```

```
f = [ 20 15 21 18 29 24 ]';
```

```
F = @(x)0.5*x'*H*x + f'*x;
```

The linear constraints are

$$A \cdot x \leq b,$$

$$Aeq \cdot x = beq,$$

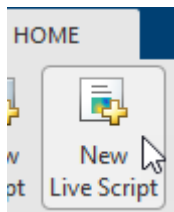
where

```
A = [-8 7 3 -4 9 0];
b = 7;
Aeq = [7 1 8 3 3 3;
       5 0 -5 1 -5 8;
       -2 -6 7 1 1 9;
       1 -1 2 -2 3 -3];
beq = [84 62 65 1]';
```

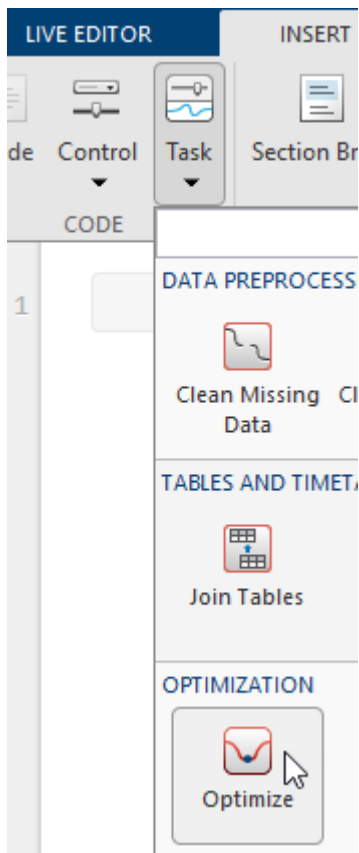
Enter the preceding code sections to get the problem variables into your workspace before proceeding.

Solve Using patternsearch in Optimize Live Editor Task

- 1 Create a new live script by clicking the **New Live Script** button in the **File** section on the **Home** tab.



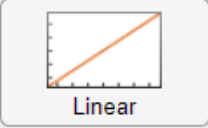
- 2 Insert an **Optimize** Live Editor task. Click the **Insert** tab and then, in the **Code** section, select **Task > Optimize**.



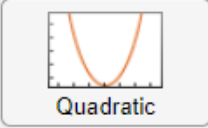
Optimize

Minimize a function with or without constraints

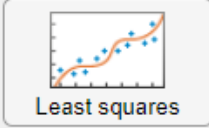
Specify problem type




Linear



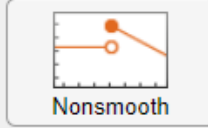
Quadratic



Least squares



Nonlinear



Nonsmooth

Select an objective type to see example functions

Unconstrained

Lower bounds

Upper bounds

Linear inequality

Select constraint types to see example formulas

Linear equality

Second-order cone

Nonlinear

Integer

Solver: fmincon - Constrained nonlinear minimization (recommended) ?

Select problem data

Objective function: From file Browse... New... ?

Initial point (x0): select

Specify solver options

Display progress

Text display: Final output

Plot:

Current point

Evaluation count

Objective value and feasibility

Objective value

Max constraint violation

Step size

Optimality measure

3 Specify Problem Type

In the **Specify problem type** section of the task, click the **Objective > Nonlinear** button.

4 Click the **Constraints > Linear inequality** and **Linear equality** buttons.

5 Select **Solver > patternsearch - Pattern search**.

6 Select Problem Data

Enter the problem variables in the **Select problem data** section of the task. To specify the objective function, select **Objective function > Function handle** and choose **F**.

7 Set the inequality constraints to **A** and **b**. Set the equality constraints to **Aeq** and **beq**.

8 To set the initial point, you first need to create a new section above the task. To do so, click the **Section Break** button on the **Insert** tab. In the new section above the task, enter the following code for the initial point.

```
x0 = [2 1 0 9 1 0]';
```

9 Run the section to place x_0 into the workspace. To run the section, place the cursor in the section and press **Ctrl+Enter** or click the blue striped bar to the left of the line number.

10 In the **Select problem data** section of the task, set x_0 as the initial point.

11 Specify Solver Options

Because this problem is linearly constrained, specify an additional solver option. Expand the **Specify solver options** section of the task, and then click the **Add** button. Set the **Poll settings** > **Poll method** to GSSPositiveBasis2N. For more information about the efficiency of the GSS poll methods for linearly constrained problems, see “Compare the Efficiency of Poll Options” on page 6-61.

12 Set Display Options

In the **Display progress** section of the task, select the **Best value** and **Mesh size** plot functions.

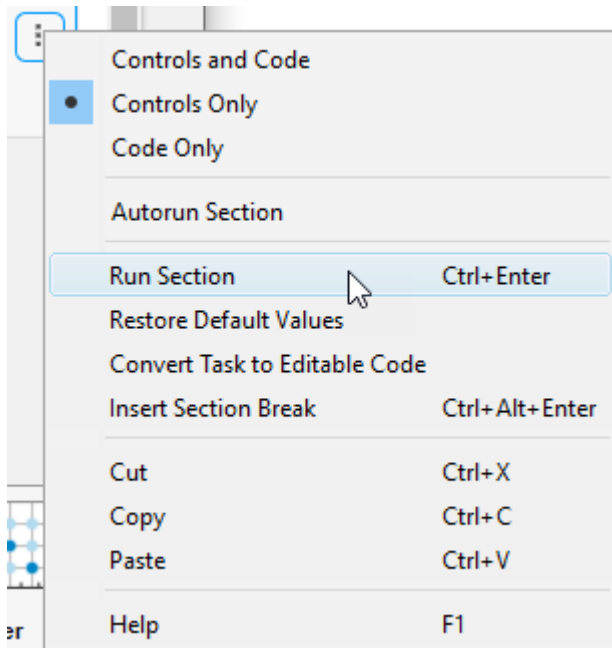
Your setup looks like this:

The screenshot shows the configuration window for the 'patternsearch - Pattern search' solver. The window is organized into several sections:

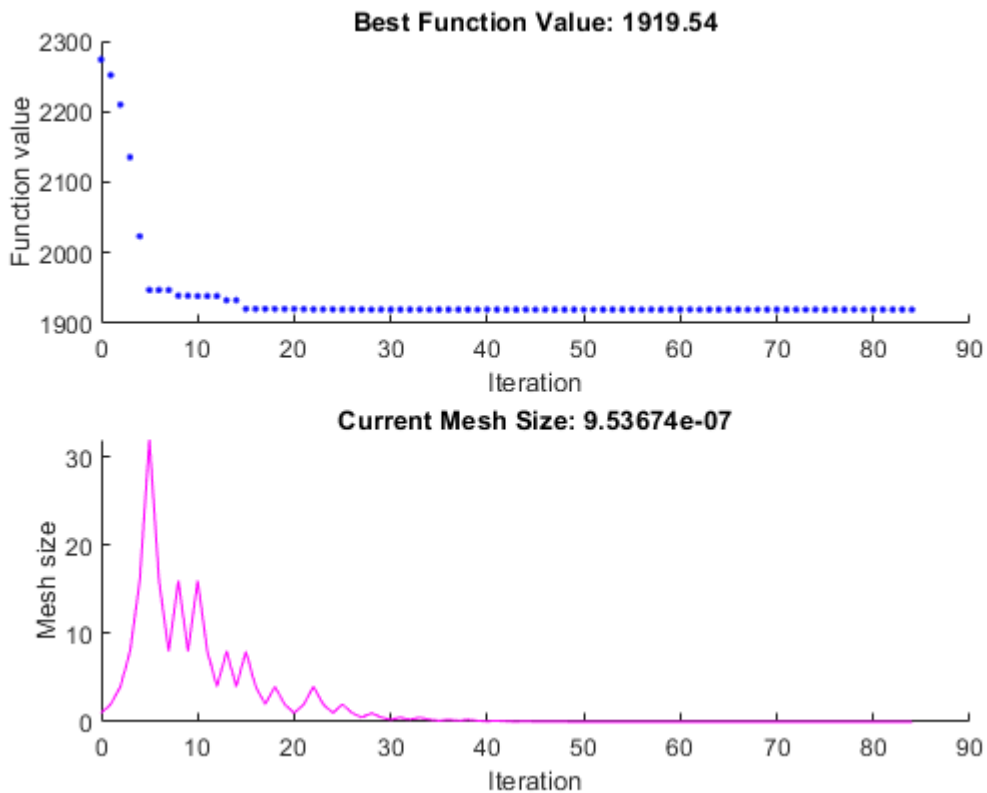
- Solver:** A dropdown menu set to 'patternsearch - Pattern search' with a help icon.
- Select problem data:**
 - Objective function:** 'Function handle' dropdown, 'F' dropdown, and a help icon.
 - Initial point (x0):** 'x0' dropdown.
 - Constraints:**
 - Linear inequality:** 'A' dropdown, '*x ≤ b', 'b' dropdown.
 - Linear equality:** 'Aeq' dropdown, '*x = beq', 'beq' dropdown.
- Specify solver options:**
 - Poll settings:** '?' icon, 'Poll settings' dropdown.
 - Polling algorithm:** 'Polling algorithm' dropdown.
 - Poll method:** 'GSSPositiveBasis2N' dropdown, with '-' and '+' buttons.
- Display progress:**
 - Text display:** 'Final output' dropdown.
 - Plot:**
 - Best value
 - Mesh size
 - Evaluation count
 - Best point
 - Max constraint violation

13 Run Solver and Examine Results

To run the solver, click the options button **:** at the top right of the task window, and select **Run Section**.



The plots appear in a separate figure window and in the task output area.



- 14 To obtain the solution point and objective function value at the solution, look at the top of the task.

Optimize

```
solution, objectiveValue = Minimize F using patternsearch solver
```

The **Optimize** Live Editor task returns the solution in a variable named `solution` and returns the objective function value in a variable named `objectiveValue`. View these values by entering the following code in the section below the task and then running the section, or by entering the code at the MATLAB command line.

```
disp(solution)
```

```
8.5165
-6.1094
4.0989
1.2877
-4.2348
2.1812
```

```
disp(objectiveValue)
```

```
1.9195e+03
```

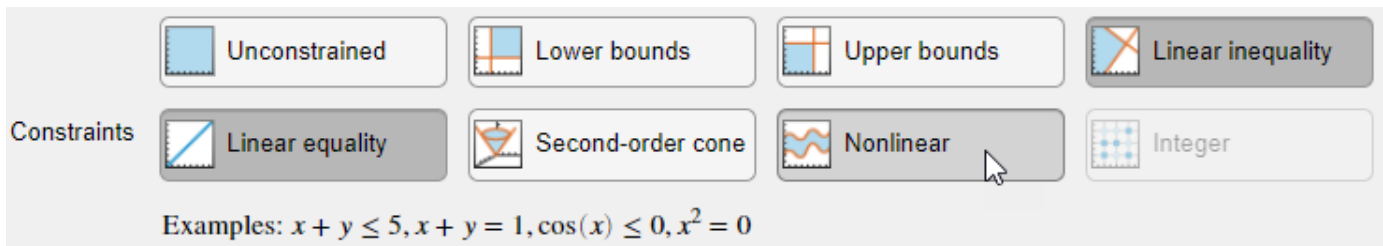
15 Include Nonlinear Constraints

Add the following nonlinear constraints to the problem.

$$-1.5 + x_1x_2 + x_1 - x_2 \leq 0$$

$$-x_1x_2 - 10 \leq 0.$$

To include these constraints, first click the **Constraints > Nonlinear** button.



- 16** In the **Select problem data** section, under **Constraints**, select **Nonlinear > Local function** and then click the **New** button. The function appears in a new section below the task. Edit the resulting code to contain the following lines.

```
function [c, ceq] = double_ineq(x)
c = [-1.5 + x(1)*x(2) + x(1) - x(2);
     -x(1)*x(2) - 10];
ceq = [];
end
```

- 17** In the **Nonlinear** constraints section, select **double_ineq**.

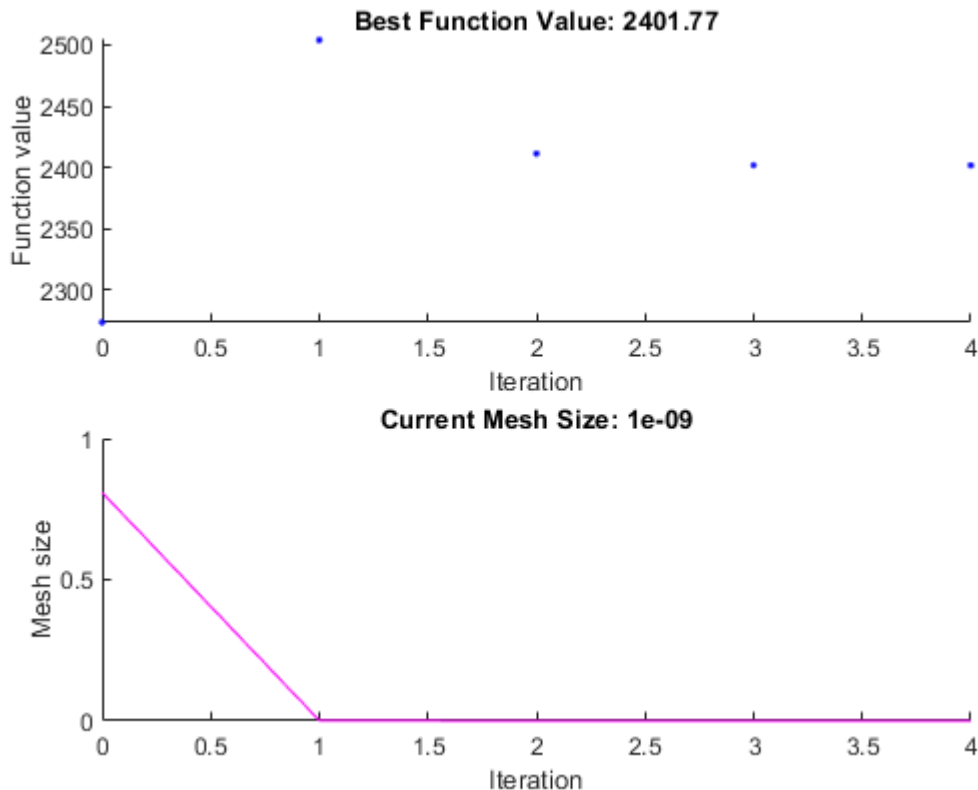
- 18** The nonlinear constraint algorithm causes `patternsearch` to take many function evaluations. In the **Specify solver options** section, click the plus sign to the right of the current options to display additional options. Then increase the maximum function evaluation limit to `5e4`.

Specify solver options

? Poll settings ▼ Polling algorithm ▼ GSSPositiveBasis2N ▼

Run time limits ▼ Max function evals ▼ 5e4

19 Run the task again to rerun the optimization.



20 View the solution and objective function value.

```
disp(solution)
```

```
7.2083
-1.3873
4.9579
-3.1393
-3.1843
4.7457
```

```
disp(objectiveValue)
```

```
2.4018e+03
```

The objective function value is higher than the value in the problem without nonlinear constraints. The previous solution is not feasible with respect to the nonlinear constraints.

The plots show many fewer iterations than before because the nonlinear constraint algorithm changes the patternsearch algorithm to include another outer loop to solve a modified problem.

The outer loop reduces the modification to the problem at each major iteration. In this case, the algorithm makes only four outer iterations. For algorithm details, see “Nonlinear Constraint Solver Algorithm for Pattern Search” on page 6-46.

Solve Using patternsearch at the Command Line

To solve the original problem (only linear constraints) at the command line, execute the following code.

```
x0 = [2 1 0 9 1 0]';
options = optimoptions('patternsearch',...
    'PollMethod','GSSPositiveBasis2N',...
    'PlotFcn',{'psplotbestf','psplotmeshsize'});
lb = [];
ub = [];
nonlcon = [];
[x,fval] = patternsearch(F,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)

Optimization terminated: mesh size less than options.MeshTolerance.

x =

    8.5165
   -6.1094
    4.0989
    1.2877
   -4.2348
    2.1812
```

```
fval =

    1.9195e+03
```

patternsearch generates the first pair of plots shown in the **Optimize** Live Editor task example.

To include the nonlinear constraints, save the following code to a file named `double_ineq.m` on the MATLAB path.

```
function [c, ceq] = double_ineq(x)
c = [-1.5 + x(1)*x(2) + x(1) - x(2);
     -x(1)*x(2) - 10];
ceq = [];
end
```

To allow the solver to run to completion with nonlinear constraints, increase the allowed number of function evaluations.

```
options.MaxFunctionEvaluations = 5e4;
```

Solve the problem including nonlinear constraints.

```
nonlcon = @double_ineq;
[x,fval] = patternsearch(F,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
```

```
Optimization terminated: mesh size less than options.MeshTolerance
and constraint violation is less than options.ConstraintTolerance.
```



```
x =  
    7.2083  
   -1.3873  
    4.9579  
   -3.1393  
   -3.1843  
    4.7457
```

```
fval =  
    2.4018e+03
```

patternsearch also generates the second pair of plots shown in the **Optimize** Live Editor task example.

Both the **Optimize** Live Editor task and the command line allow you to formulate and solve problems, and they give identical results. The command line is more streamlined, but provides less help for choosing a solver, setting up the problem, and choosing options such as plot functions. You can also start a problem using **Optimize**, and then generate code for command line use, as in “Constrained Nonlinear Problem Using Optimize Live Editor Task or Solver”.

See Also

patternsearch

More About

- “Constrained Minimization Using Pattern Search, Solver-Based” on page 6-14
- “Effects of Pattern Search Options” on page 6-18
- “Optimize ODEs in Parallel” on page 6-87
- “Add Interactive Tasks to a Live Script”

Use Cache

Typically, at any given iteration of a pattern search, some of the mesh points might coincide with mesh points at previous iterations. By default, the pattern search recomputes the objective function at these mesh points even though it has already computed their values and found that they are not optimal. If computing the objective function takes a long time, this can make the pattern search run significantly longer.

You can eliminate these redundant computations by using a cache, that is, by storing a history of the points that the pattern search has already visited. To do so, set **Cache** to **On** in **Cache** options. At each poll, the pattern search checks to see whether the current mesh point is within a specified tolerance, **Tolerance**, of a point in the cache. If so, the search does not compute the objective function for that point, but uses the cached function value and moves on to the next point.

Note When **Cache** is set to **On**, the pattern search might fail to identify a point in the current mesh that improves the objective function because it is within the specified tolerance of a point in the cache. As a result, the pattern search might run for more iterations with **Cache** set to **On** than with **Cache** set to **Off**. It is generally a good idea to keep the value of **Tolerance** very small, especially for highly nonlinear objective functions.

Note Cache does not work when you run the solver in parallel.

For example, set up the problem described in “Constrained Minimization Using patternsearch and Optimize Live Editor Task” on page 6-71 as follows:

- 1 Enter the following at the command line:

```
x0 = [2 1 0 9 1 0];
Aineq = [-8 7 3 -4 9 0];
bineq = 7;
Aeq = [7 1 8 3 3 3; 5 0 -5 1 -5 8; -2 -6 7 1 1 9; 1 -1 2 -2 3 -3];
beq = [84 62 65 1];
H = [36 17 19 12 8 15;
     17 33 18 11 7 14;
     19 18 43 13 8 16;
     12 11 13 18 6 11;
     8 7 8 6 9 8;
     15 14 16 11 8 29];
```

```
f = [ 20 15 21 18 29 24 ]';
```

```
F = @(x)0.5*x'*H*x + f'*x;
```

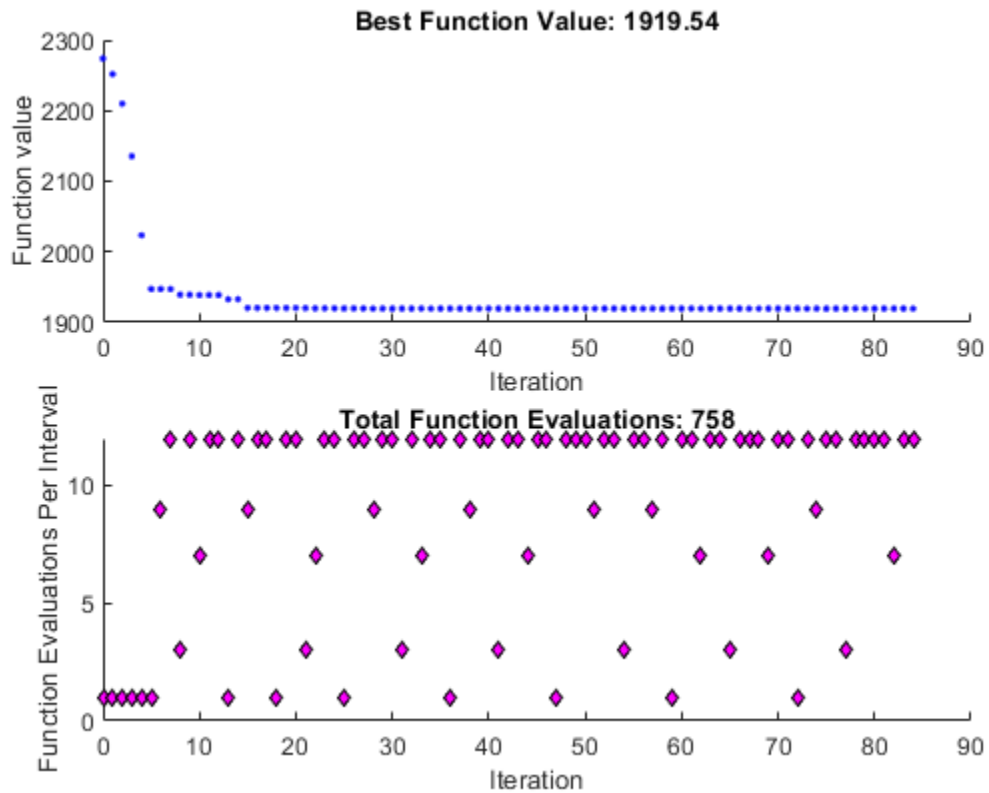
- 2 Create options to plot the best function value and function evaluations. Because the problem has linear constraints, use the 'GSSPositiveBasis2N' poll method. Turn off the display.

```
opts = optimoptions('patternsearch','PollMethod','GSSPositiveBasis2N',...
    'PlotFcn',{@psplotbestf,@psplotfuncount},'Display','none');
```

- 3 Run the optimization.

```
[x,fval,exitflag,output] = patternsearch(F,x0,...
    Aineq,bineq,Aeq,beq,[],[],[],opts);
```

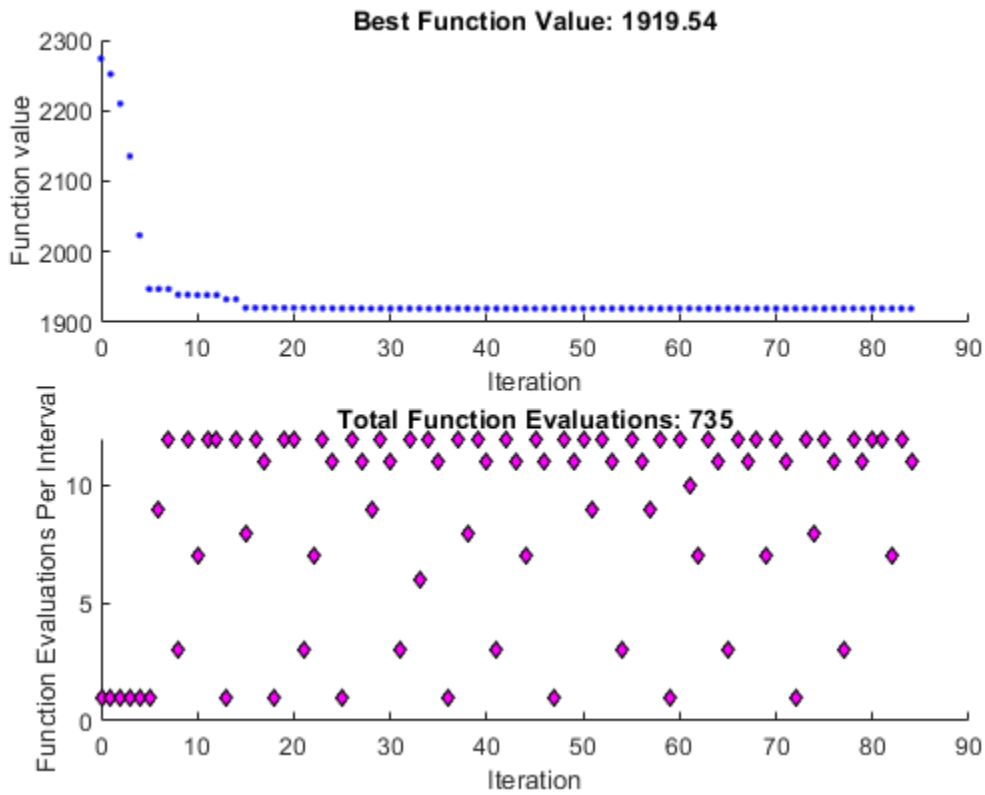
After the pattern search finishes, the plots appear as shown in the following figure.



Note that the total function count is 758.

Now, set the Cache option to 'On' and run the example again.

```
opts.Cache = 'on';
[x2,fval2,exitflag2,output2] = patternsearch(F,x0,...
    Aineq,bineq,Aeq,beq,[],[],[],opts);
```



The total function count is reduced to 735.

```
[output.funccount,output2.funccount]
```

```
ans =
```

```
758 735
```

See Also

patternsearch

More About

- "Pattern Search Options" on page 17-7

Vectorize the Objective and Constraint Functions

In this section...

“Vectorize for Speed” on page 6-83

“Vectorized Objective Function” on page 6-83

“Vectorized Constraint Functions” on page 6-85

“Example of Vectorized Objective and Constraints” on page 6-85

Vectorize for Speed

Direct search often runs faster if you *vectorize* the objective and nonlinear constraint functions. This means your functions evaluate all the points in a poll or search pattern at once, with one function call, without having to loop through the points one at a time. Therefore, the option `UseVectorized = true` works only when `UseCompletePoll` or `UseCompleteSearch` is also set to `true`. However, when you set `UseVectorized = true`, `patternsearch` checks that the objective and any nonlinear constraint functions give outputs of the correct shape for vectorized calculations, regardless of the setting of the `UseCompletePoll` or `UseCompleteSearch` options.

If there are nonlinear constraints, the objective function and the nonlinear constraints all need to be vectorized in order for the algorithm to compute in a vectorized manner.

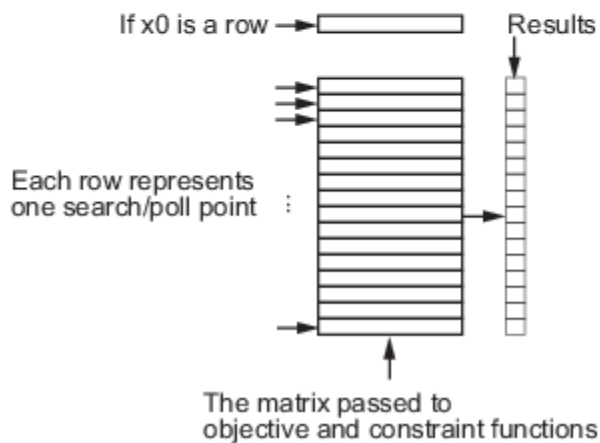
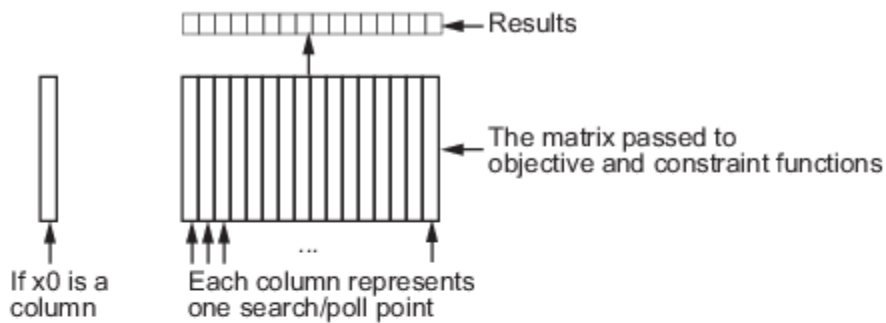
Note Write your vectorized objective function or nonlinear constraint function to accept a matrix with an arbitrary number of points. `patternsearch` sometimes evaluates a single point even during a vectorized calculation.

Vectorized Objective Function

A vectorized objective function accepts a matrix as input and generates a vector of function values, where each function value corresponds to one row or column of the input matrix. `patternsearch` resolves the ambiguity in whether the rows or columns of the matrix represent the points of a pattern as follows. Suppose the input matrix has m rows and n columns:

- If the initial point x_0 is a column vector of size m , the objective function takes each column of the matrix as a point in the pattern and returns a row vector of size n .
- If the initial point x_0 is a row vector of size n , the objective function takes each row of the matrix as a point in the pattern and returns a column vector of size m .
- If the initial point x_0 is a scalar, `patternsearch` assumes that x_0 is a row vector. Therefore, the input matrix has one column ($n = 1$, the input matrix is a vector), and each entry of the matrix represents one row for the objective function to evaluate. The output of the objective function in this case is a column vector of size m .

Pictorially, the matrix and calculation are represented by the following figure.



Structure of Vectorized Functions

For example, suppose the objective function is

$$f(x) = x_1^4 + x_2^4 - 4x_1^2 - 2x_2^2 + 3x_1 - x_2/2.$$

If the initial vector x_0 is a column vector, such as $[0; 0]$, a function for vectorized evaluation is

```
function f = vectorizedc(x)
```

```
f = x(1,:).^4+x(2,:).^4-4*x(1,:).^2-2*x(2,:).^2 ...
+3*x(1,:)-.5*x(2,:);
```

If the initial vector x_0 is a row vector, such as $[0, 0]$, a function for vectorized evaluation is

```
function f = vectorizedr(x)
```

```
f = x(:,1).^4+x(:,2).^4-4*x(:,1).^2-2*x(:,2).^2 ...
+3*x(:,1)-.5*x(:,2);
```

Tip If you want to use the same objective (fitness) function for both pattern search and genetic algorithm, write your function to have the points represented by row vectors, and write x_0 as a row vector. The genetic algorithm always takes individuals as the rows of a matrix. This was a design decision—the genetic algorithm does not require a user-supplied population, so needs to have a default format.

To minimize `vectorizedc`, enter the following commands:

```
options=optimoptions('patternsearch','UseVectorized',true,'UseCompletePoll',true);
x0=[0;0];
[x,fval]=patternsearch(@vectorizedc,x0,...
    [],[],[],[],[],[],[],options)
```

MATLAB returns the following output:

Optimization terminated: mesh size less than options.MeshTolerance.

```
x =
   -1.5737
    1.0575
```

```
fval =
   -10.0088
```

Vectorized Constraint Functions

Only nonlinear constraints need to be vectorized; bounds and linear constraints are handled automatically. If there are nonlinear constraints, the objective function and the nonlinear constraints all need to be vectorized in order for the algorithm to compute in a vectorized manner.

The same considerations hold for constraint functions as for objective functions: the initial point `x0` determines the type of points (row or column vectors) in the poll or search. If the initial point is a row vector of size k , the matrix `x` passed to the constraint function has k columns. Similarly, if the initial point is a column vector of size k , the matrix of poll or search points has k rows. The figure “Structure of Vectorized Functions” on page 6-84 may make this clear. If the initial point is a scalar, `patternsearch` assumes that it is a row vector.

Your nonlinear constraint function returns two matrices, one for inequality constraints, and one for equality constraints. Suppose there are n_c nonlinear inequality constraints and n_{ceq} nonlinear equality constraints. For row vector `x0`, the constraint matrices have n_c and n_{ceq} columns respectively, and the number of rows is the same as in the input matrix. Similarly, for a column vector `x0`, the constraint matrices have n_c and n_{ceq} rows respectively, and the number of columns is the same as in the input matrix. In figure “Structure of Vectorized Functions” on page 6-84, “Results” includes both n_c and n_{ceq} .

Example of Vectorized Objective and Constraints

Suppose that the nonlinear constraints are

$$\frac{x_1^2}{9} + \frac{x_2^2}{4} \leq 1 \text{ (the interior of an ellipse),}$$

$$x_2 \geq \cosh(x_1) - 1.$$

Write a function for these constraints for row-form `x0` as follows:

```
function [c ceq] = ellipsecosh(x)
c(:,1)=x(:,1).^2/9+x(:,2).^2/4-1;
c(:,2)=cosh(x(:,1))-x(:,2)-1;
ceq=[];
```

Minimize `vectorizedr` (defined in “Vectorized Objective Function” on page 6-83) subject to the constraints `ellipsecosh`:

```
x0=[0,0];
options = optimoptions('patternsearch','UseVectorized',true,'UseCompletePoll',true);
[x,fval] = patternsearch(@vectorizedr,x0,...
    [],[],[],[],[],[],[],@ellipsecosh,options)
```

MATLAB returns the following output:

```
Optimization terminated: mesh size less than options.MeshTolerance
and constraint violation is less than options.ConstraintTolerance.
```

```
x =
   -1.3516    1.0612
```

```
fval =
   -9.5394
```

See Also

More About

- “Optimize ODEs in Parallel” on page 6-87
- “Compute Objective Functions” on page 2-2

Optimize ODEs in Parallel

This example shows how to optimize parameters of an ODE.

It also shows how to avoid computing the objective and nonlinear constraint function twice when the ODE solution returns both. The example compares `patternsearch` and `ga` in terms of time to run the solver and the quality of the solutions.

You need a Parallel Computing Toolbox license to use parallel computing.

Step 1. Define the problem.

The problem is to change the position and angle of a cannon to fire a projectile as far as possible beyond a wall. The cannon has a muzzle velocity of 300 m/s. The wall is 20 m high. If the cannon is too close to the wall, it has to fire at too steep an angle, and the projectile does not travel far enough. If the cannon is too far from the wall, the projectile does not travel far enough either.

Air resistance slows the projectile. The resisting force is proportional to the square of the velocity, with proportionality constant 0.02. Gravity acts on the projectile, accelerating it downward with constant 9.81 m/s². Therefore, the equations of motion for the trajectory $x(t)$ are

$$\frac{d^2x(t)}{dt^2} = -0.02\|v(t)\|v(t) - (0, 9.81),$$

where $v(t) = dx(t)/dt$.

The initial position x_0 and initial velocity x_{p0} are 2-D vectors. However, the initial height $x_0(2)$ is 0, so the initial position depends only on the scalar $x_0(1)$. And the initial velocity x_{p0} has magnitude 300 (the muzzle velocity), so depends only on the initial angle, a scalar. For an initial angle th , $x_{p0} = 300 * (\cos(th), \sin(th))$. Therefore, the optimization problem depends only on two scalars, so it is a 2-D problem. Use the horizontal distance and the angle as the decision variables.

Step 2. Formulate the ODE model.

ODE solvers require you to formulate your model as a first-order system. Augment the trajectory vector $(x_1(t), x_2(t))$ with its time derivative $(x'_1(t), x'_2(t))$ to form a 4-D trajectory vector. In terms of this augmented vector, the differential equation becomes

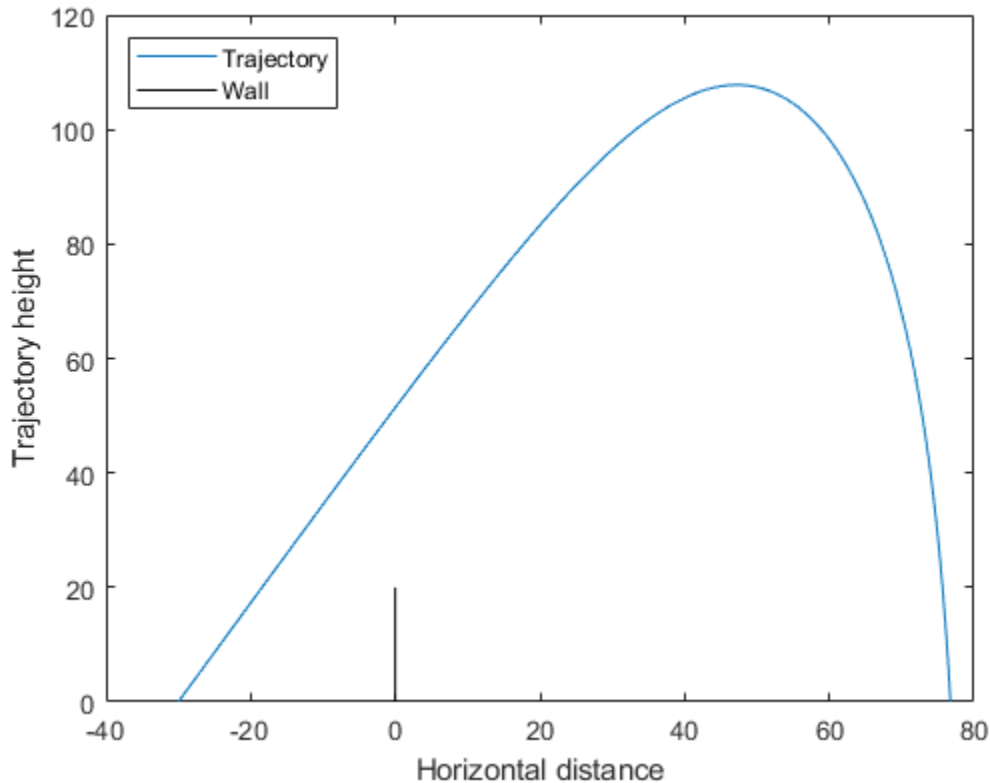
$$\frac{d}{dt}x(t) = \begin{bmatrix} x_3(t) \\ x_4(t) \\ -.02\|(x_3(t), x_4(t))\|x_3(t) \\ -.02\|(x_3(t), x_4(t))\|x_4(t) - 9.81 \end{bmatrix}.$$

Write the differential equation as a function file, and save it on your MATLAB path.

```
function f = cannonfodder(t,x)

f = [x(3);x(4);x(3);x(4)]; % Initial, gets f(1) and f(2) correct
nrm = norm(x(3:4)) * .02; % Norm of the velocity times constant
f(3) = -x(3)*nrm; % Horizontal acceleration
f(4) = -x(4)*nrm - 9.81; % Vertical acceleration
```

Visualize the solution of the ODE starting 30 m from the wall at an angle of $\pi/3$.



Code for generating the figure

```
x0 = [-30;0;300*cos(pi/3);300*sin(pi/3)];
sol = ode45(@cannonfodder,[0,10],x0);
% Find the time when the projectile lands
zerofnd = fzero(@(r)deval(sol,r,2),[sol.x(2),sol.x(end)]);
t = linspace(0,zerofnd); % equal times for plot
xs = deval(sol,t,1); % interpolated x values
ys = deval(sol,t,2); % interpolated y values
plot(xs,ys)
hold on
plot([0,0],[0,20],'k') % Draw the wall
xlabel('Horizontal distance')
ylabel('Trajectory height')
legend('Trajectory','Wall','Location','NW')
ylim([0 120])
hold off
```

Step 3. Solve using patternsearch.

The problem is to find initial position $x_0(1)$ and initial angle $x_0(2)$ to maximize the distance from the wall the projectile lands. Because this is a maximization problem, minimize the negative of the distance (see “Maximizing vs. Minimizing” on page 2-5).

To use `patternsearch` to solve this problem, you must provide the objective, constraint, initial guess, and options.

These two files are the objective and constraint functions. Copy them to a folder on your MATLAB path.

```
function f = cannonobjective(x)
x0 = [x(1);0;300*cos(x(2));300*sin(x(2))];

sol = ode45(@cannonfodder,[0,15],x0);

% Find the time t when y_2(t) = 0
zerofnd = fzero(@(r)deval(sol,r,2),[sol.x(2),sol.x(end)]);
% Then find the x-position at that time
f = deval(sol,zerofnd,1);

f = -f; % take negative of distance for maximization

function [c,ceq] = cannonconstraint(x)

ceq = [];
x0 = [x(1);0;300*cos(x(2));300*sin(x(2))];

sol = ode45(@cannonfodder,[0,15],x0);

if sol.y(1,end) <= 0 % Projectile never reaches wall
    c = 20 - sol.y(2,end);
else
    % Find when the projectile crosses x = 0
    zerofnd = fzero(@(r)deval(sol,r,1),[sol.x(2),sol.x(end)]);
    % Then find the height there, and subtract from 20
    c = 20 - deval(sol,zerofnd,2);
end
```

Notice that the objective and constraint functions set their input variable `x0` to a 4-D initial point for the ODE solver. The ODE solver does not stop if the projectile hits the wall. Instead, the constraint function simply becomes positive, indicating an infeasible initial value.

The initial position `x0(1)` cannot be above 0, and it is futile to have it be below -200. (It should be near -20 because, with no air resistance, the longest trajectory would start at -20 at an angle $\pi/4$.) Similarly, the initial angle `x0(2)` cannot be below 0, and cannot be above $\pi/2$. Set bounds slightly away from these initial values:

```
lb = [-200;0.05];
ub = [-1;pi/2-.05];
x0 = [-30,pi/3]; % Initial guess
```

Set the `UseCompletePoll` option to `true`. This gives a higher-quality solution, and enables direct comparison with parallel processing, because computing in parallel requires this setting.

```
opts = optimoptions('patternsearch','UseCompletePoll',true);
```

Call `patternsearch` to solve the problem.

```
tic % Time the solution
[xsolution,distance,eflag,outpt] = patternsearch(@cannonobjective,x0,...
    [],[],[],[],lb,ub,@cannonconstraint,opts)
toc
```

```
Optimization terminated: mesh size less than options.MeshTolerance
and constraint violation is less than options.ConstraintTolerance.
```

```
xsolution =
```

```
    -28.8123    0.6095
```

```
distance =
```

```
   -125.9880
```

```
eflag =
```

```
    1
```

```
outpt =
```

```
struct with fields:
```

```
    function: @cannonobjective
   problemtype: 'nonlinearconstr'
   pollmethod: 'gpspositivebasis2n'
  maxconstraint: 0
  searchmethod: []
   iterations: 5
   funccount: 269
   meshsize: 8.9125e-07
   rngstate: [1x1 struct]
   message: 'Optimization terminated: mesh size less than options.MeshTolerance, and constraint violation is less than options.ConstraintTolerance.'
```

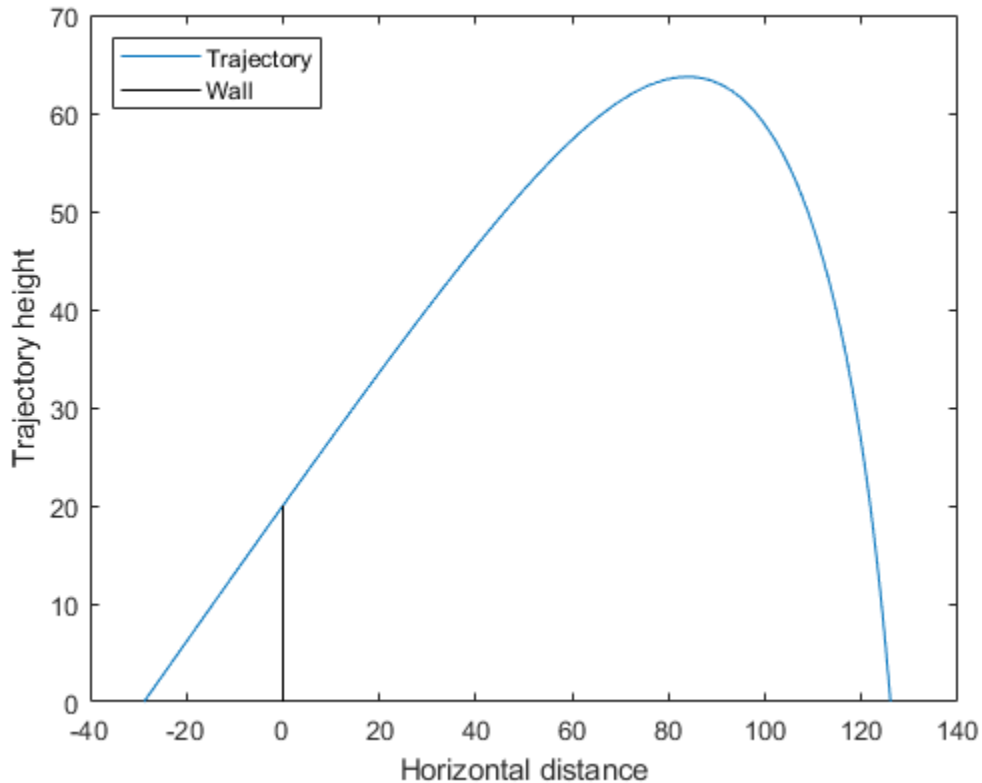
```
Elapsed time is 0.792152 seconds.
```

Starting the projectile about 29 m from the wall at an angle 0.6095 radian results in the farthest distance, about 126 m. The reported distance is negative because the objective function is the negative of the distance to the wall.

Visualize the solution.

```
x0 = [xsolution(1);0;300*cos(xsolution(2));300*sin(xsolution(2))];
```

```
sol = ode45(@cannonfodder,[0,15],x0);
% Find the time when the projectile lands
zerofnd = fzero(@(r)deval(sol,r,2),[sol.x(2),sol.x(end)]);
t = linspace(0,zerofnd); % equal times for plot
xs = deval(sol,t,1); % Interpolated x values
ys = deval(sol,t,2); % Interpolated y values
plot(xs,ys)
hold on
plot([0,0],[0,20],'k') % Draw the wall
xlabel('Horizontal distance')
ylabel('Trajectory height')
legend('Trajectory','Wall','Location','NW')
ylim([0 70])
hold off
```



Step 4. Avoid calling the expensive subroutine twice.

Both the objective and nonlinear constraint function call the ODE solver to calculate their values. Use the technique in “Objective and Nonlinear Constraints in the Same Function” to avoid calling the solver twice. The `runcannon` file implements this technique. Copy this file to a folder on your MATLAB path.

```
function [x,f,eflag,outpt] = runcannon(x0,opts)

if nargin == 1 % No options supplied
    opts = [];
end

xLast = []; % Last place ode solver was called
sol = []; % ODE solution structure

fun = @objfun; % The objective function, nested below
cfun = @constr; % The constraint function, nested below

lb = [-200;0.05];
ub = [-1;pi/2-.05];

% Call patternsearch
[x,f,eflag,outpt] = patternsearch(fun,x0,[],[],[],[],lb,ub,cfun,opts);

function y = objfun(x)
    if ~isequal(x,xLast) % Check if computation is necessary
```

```

        x0 = [x(1);0;300*cos(x(2));300*sin(x(2))];
        sol = ode45(@cannonfodder,[0,15],x0);
        xLast = x;
    end
    % Now compute objective function
    % First find when the projectile hits the ground
    zerofnd = fzero(@(r)deval(sol,r,2),[sol.x(2),sol.x(end)]);
    % Then compute the x-position at that time
    y = deval(sol,zerofnd,1);
    y = -y; % take negative of distance
end

function [c,ceq] = constr(x)
    ceq = [];
    if ~isequal(x,xLast) % Check if computation is necessary
        x0 = [x(1);0;300*cos(x(2));300*sin(x(2))];
        sol = ode45(@cannonfodder,[0,15],x0);
        xLast = x;
    end
    % Now compute constraint functions
    % First find when the projectile crosses x = 0
    zerofnd = fzero(@(r)deval(sol,r,1),[sol.x(1),sol.x(end)]);
    % Then find the height there, and subtract from 20
    c = 20 - deval(sol,zerofnd,2);
end

end
end

```

Reinitialize the problem and time the call to `runcannon`.

```

x0 = [-30;pi/3];
tic
[xsolution,distance,eflag,outpt] = runcannon(x0,opts);
toc

```

Optimization terminated: mesh size less than options.MeshTolerance and constraint violation is less than options.ConstraintTolerance. Elapsed time is 0.670715 seconds.

The solver ran faster than before. If you examine the solution, you see that the output is identical.

Step 5. Compute in parallel.

Try to save more time by computing in parallel. Begin by opening a parallel pool of workers.

```

parpool

Starting parpool using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).

```

```
ans =
```

```

ProcessPool with properties:

    Connected: true
    NumWorkers: 6
    Cluster: local
    AttachedFiles: {}
    AutoAddClientPath: true

```

```

IdleTimeout: 30 minutes (30 minutes remaining)
SpmEnabled: true

```

Set the options to use parallel computing, and rerun the solver.

```

opts = optimoptions('patternsearch',opts,'UseParallel',true);
x0 = [-30;pi/3];
tic
[xsolution,distance,eflag,outpt] = runcannon(x0,opts);
toc

```

```

Optimization terminated: mesh size less than options.MeshTolerance
and constraint violation is less than options.ConstraintTolerance.
Elapsed time is 1.894515 seconds.

```

In this case, parallel computing was slower. If you examine the solution, you see that the output is identical.

Step 6. Compare with the genetic algorithm.

You can also try to solve the problem using the genetic algorithm. However, the genetic algorithm is usually slower and less reliable.

As explained in “Objective and Nonlinear Constraints in the Same Function”, you cannot save time when using `ga` by the nested function technique used by `patternsearch` in Step 4. Instead, call `ga` in parallel by setting the appropriate options.

```

options = optimoptions('ga','UseParallel',true);
rng default % For reproducibility
tic % Time the solution
[xsolution,distance,eflag,outpt] = ga(@cannonobjective,2,...
    [],[],[],[],lb,ub,@cannonconstraint,options)
toc

```

```

Optimization terminated: average change in the fitness value less than options.FunctionTolerance
and constraint violation is less than options.ConstraintTolerance.

```

```

xsolution =

```

```

    -37.6217    0.4926

```

```

distance =

```

```

   -122.2181

```

```

eflag =

```

```

     1

```

```

outpt =

```

```

struct with fields:

```

```

    problemtype: 'nonlinearconstr'
        rngstate: [1x1 struct]

```

```
generations: 4
  funccount: 9874
    message: 'Optimization terminated: average change in the fitness value less than option
maxconstraint: 0
  hybridflag: []
```

Elapsed time is 12.529131 seconds.

The `ga` solution is not as good as the `patternsearch` solution: 122 m versus 126 m. `ga` takes more time: about 12 s versus under 2 s; `patternsearch` takes even less time in serial and nested, less than 1 s. Running `ga` serially takes even longer, about 30 s in one test run.

See Also

Related Examples

- “Objective and Nonlinear Constraints in the Same Function”

More About

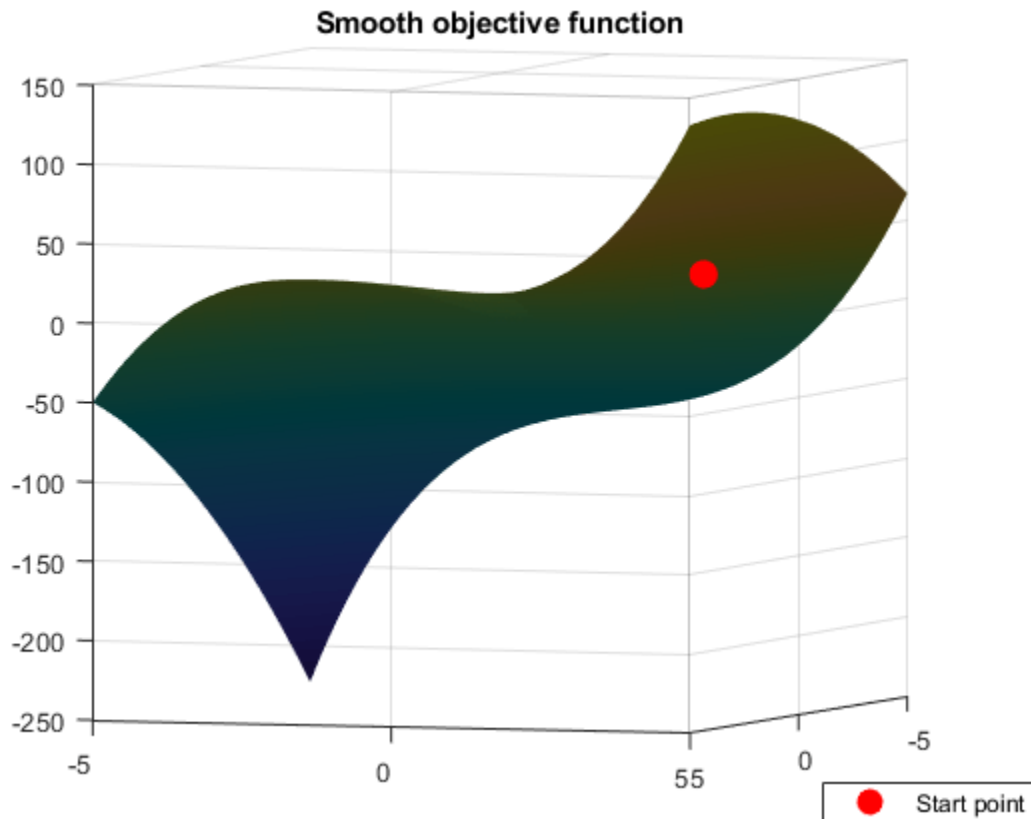
- “Parallel Computing”
- “Surrogate Optimization with Nonlinear Constraint” on page 11-41

Optimization of Stochastic Objective Function

This example shows how to find a minimum of a stochastic objective function using `patternsearch`. It also shows how Optimization Toolbox™ solvers are not suitable for this type of problem. The example uses a simple 2-dimensional objective function that is then perturbed by noise.

Initialization

```
X0 = [2.5 -2.5]; % Starting point.
LB = [-5 -5]; % Lower bound
UB = [5 5]; % Upper bound
range = [LB(1) UB(1); LB(2) UB(2)];
Objfcn = @smoothFcn; % Handle to the objective function.
% Plot the smooth objective function
fig = figure('Color','w');
showSmoothFcn(Objfcn,range);
hold on;
title('Smooth objective function');
ph = [];
ph(1) = plot3(X0(1),X0(2),Objfcn(X0)+30,'or','MarkerSize',10,'MarkerFaceColor','r');
hold off;
ax = gca;
ax.CameraPosition = [-31.0391 -85.2792 -281.4265];
ax.CameraTarget = [0 0 -50];
ax.CameraViewAngle = 6.7937;
% Add legend information
legendLabels = {'Start point'};
lh = legend(ph,legendLabels,'Location','SouthEast');
lp = lh.Position;
lh.Position = [1-lp(3)-0.005 0.005 lp(3) lp(4)];
```



Run `fmincon` on a Smooth Objective Function

The objective function is smooth (twice continuously differentiable). Solve the optimization problem using the Optimization Toolbox `fmincon` solver. `fmincon` finds a constrained minimum of a function of several variables. This function has a unique minimum at the point $x^* = [-5, -5]$ where it has a value $f(x^*) = -250$.

Set options to return iterative display.

```
options = optimoptions(@fmincon,'Algorithm','interior-point','Display','iter');
[Xop,Fop] = fmincon(Objfcn,X0,[],[],[],[],LB,UB,[],options)
figure(fig);
hold on;
```

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of step
0	3	-1.062500e+01	0.000e+00	2.004e+01	
1	6	-1.578420e+02	0.000e+00	5.478e+01	6.734e+00
2	9	-2.491310e+02	0.000e+00	6.672e+01	1.236e+00
3	12	-2.497554e+02	0.000e+00	2.397e-01	6.310e-03
4	15	-2.499986e+02	0.000e+00	5.065e-02	8.016e-03
5	18	-2.499996e+02	0.000e+00	9.632e-05	3.367e-05
6	21	-2.500000e+02	0.000e+00	1.502e-04	6.867e-06
7	24	-2.500000e+02	0.000e+00	1.159e-06	6.920e-08

Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in feasible directions, to within the value of the optimality tolerance, and constraints are satisfied to within the value of the constraint tolerance.

Xop =

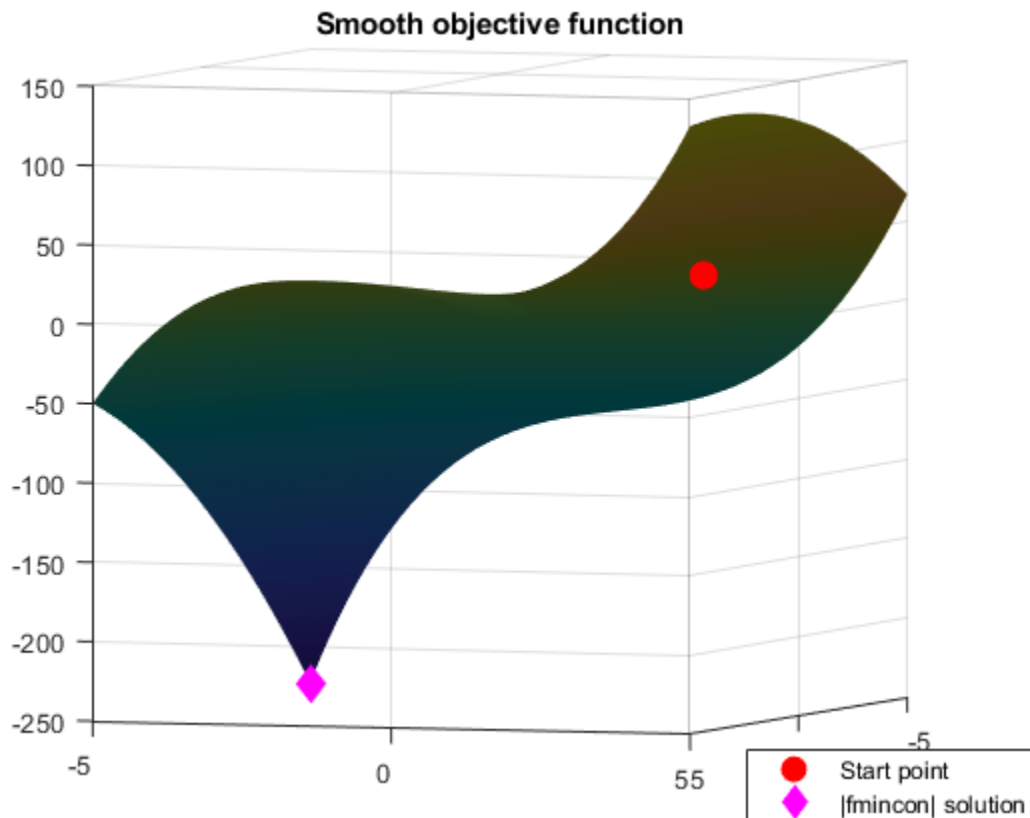
```
-5.0000  -5.0000
```

Fop =

```
-250.0000
```

Plot the final point

```
ph(2) = plot3(Xop(1),Xop(2),Fop,'dm','MarkerSize',10,'MarkerFaceColor','m');
% Add a legend to plot
legendLabels = [legendLabels, '|fmincon| solution'];
lh = legend(ph,legendLabels,'Location','SouthEast');
lp = lh.Position;
lh.Position = [1-lp(3)-0.005 0.005 lp(3) lp(4)];
hold off;
```



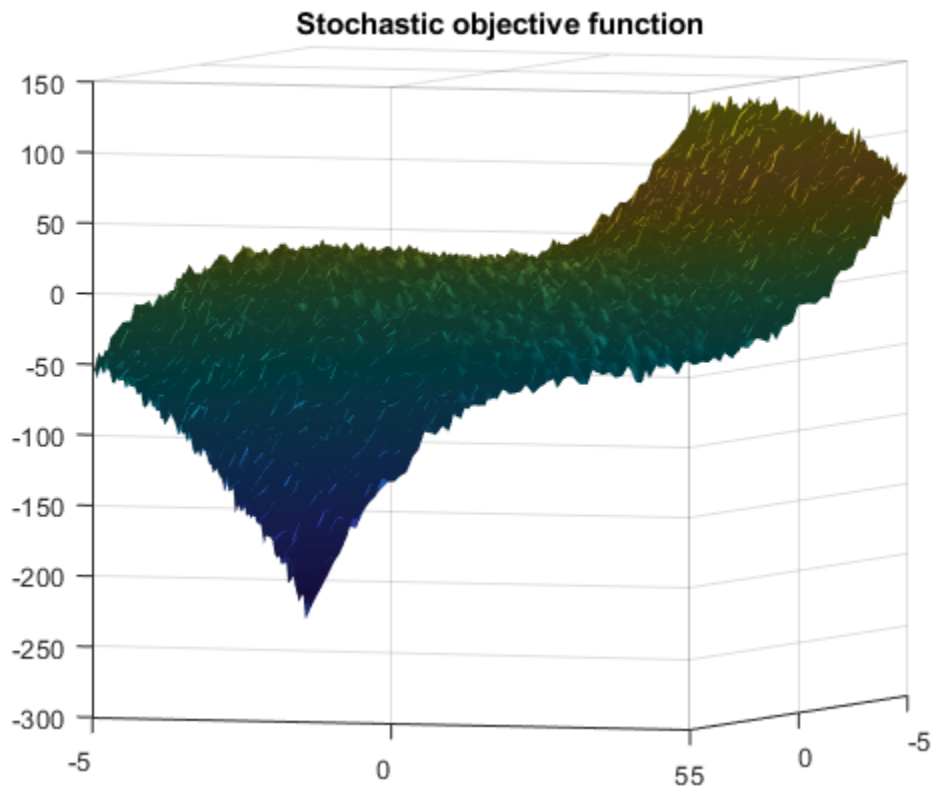
Stochastic Objective Function

Now perturb the objective function by adding random noise.

```

rng(0,'twister') % Reset the global random number generator
peaknoise = 4.5;
Objfcn = @(x) smoothFcn(x,peaknoise); % Handle to the objective function.
% Plot the objective function (non-smooth)
fig = figure('Color','w');
showSmoothFcn(Objfcn,range);
title('Stochastic objective function')
ax = gca;
ax.CameraPosition = [-31.0391 -85.2792 -281.4265];
ax.CameraTarget = [0 0 -50];
ax.CameraViewAngle = 6.7937;

```



Run `fmincon` on a Stochastic Objective Function

The perturbed objective function is stochastic and not smooth. `fmincon` is a general constrained optimization solver which finds a local minimum using derivatives of the objective function. If you do not provide the first derivatives of the objective function, `fmincon` uses finite differences to approximate the derivatives. In this example, the objective function is random, so finite difference estimates derivatives hence can be unreliable. `fmincon` can potentially stop at a point that is not a minimum. This may happen because the optimal conditions seems to be satisfied at the final point because of noise, or `fmincon` could not make further progress.

```

[Xop,Fop] = fmincon(Objfcn,X0,[],[],[],[],LB,UB,[],options)
figure(fig);
hold on;
ph = [];
ph(1) = plot3(X0(1),X0(2),Objfcn(X0)+30,'or','MarkerSize',10,'MarkerFaceColor','r');

```

```

ph(2) = plot3(Xop(1),Xop(2),Fop, 'dm', 'MarkerSize',10, 'MarkerFaceColor', 'm');
% Add legend to plot
legendLabels = {'Start point', '|fmincon| solution'};
lh = legend(ph, legendLabels, 'Location', 'SouthEast');
lp = lh.Position;
lh.Position = [1-lp(3)-0.005 0.005 lp(3) lp(4)];
hold off;

```

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of step
0	3	-1.925772e+01	0.000e+00	2.126e+08	
1	6	-7.107849e+01	0.000e+00	2.623e+08	8.873e+00
2	11	-8.055890e+01	0.000e+00	2.401e+08	6.715e-01
3	20	-8.325315e+01	0.000e+00	7.348e+07	3.047e-01
4	48	-8.366302e+01	0.000e+00	1.762e+08	1.593e-07
5	64	-8.591081e+01	0.000e+00	1.569e+08	3.111e-10

Local minimum possible. Constraints satisfied.

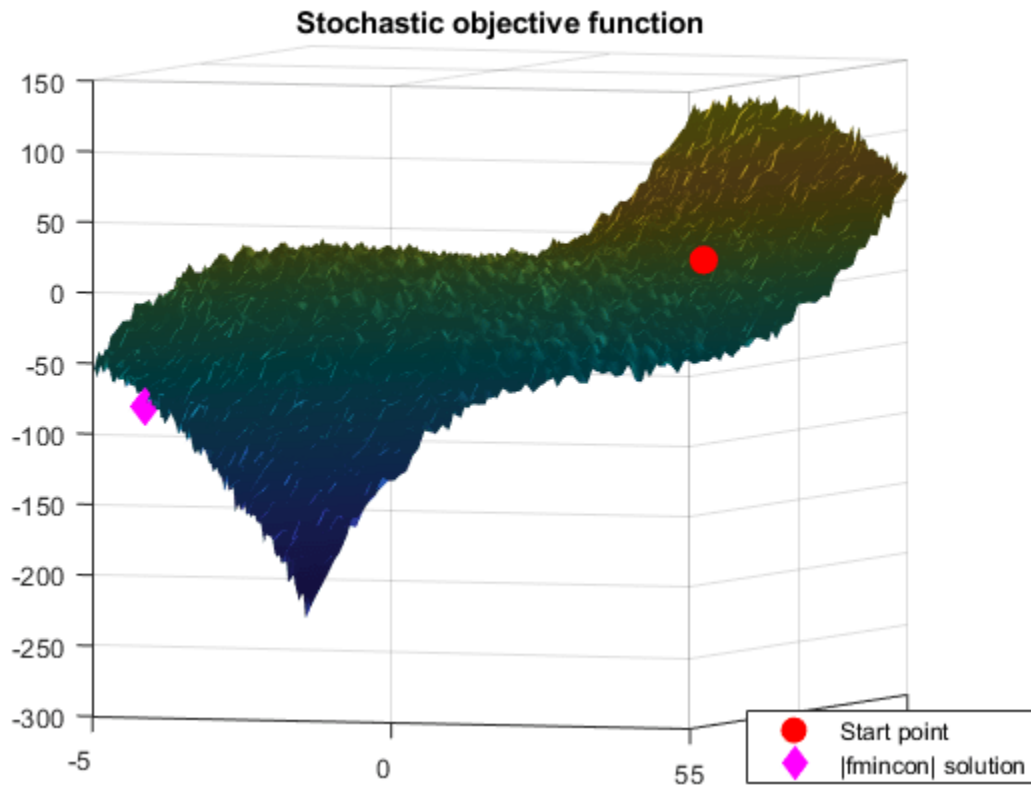
fmincon stopped because the size of the current step is less than the value of the step size tolerance and constraints are satisfied to within the value of the constraint tolerance.

Xop =

```
-4.9628    2.6673
```

Fop =

```
-85.9108
```



Run patternsearch

Now minimize the stochastic objective function using the Global Optimization Toolbox `patternsearch` solver. Pattern search optimization techniques are a class of direct search methods for optimization. A pattern search algorithm does not use derivatives of the objective function to find an optimal point.

```
PSoptions = optimoptions(@patternsearch,'Display','iter');
[Xps,Fps] = patternsearch(Objfcn,X0,[],[],[],[],LB,UB,PSoptions)
figure(fig);
hold on;
ph(3) = plot3(Xps(1),Xps(2),Fps,'dc','MarkerSize',10,'MarkerFaceColor','c');
% Add legend to plot
legendLabels = [legendLabels, 'Pattern Search solution'];
lh = legend(ph,legendLabels,'Location','SouthEast');
lp = lh.Position;
lh.Position = [1-lp(3)-0.005 0.005 lp(3) lp(4)];
hold off
```

Iter	Func-count	f(x)	MeshSize	Method
0	1	-7.20766	1	
1	3	-34.7227	2	Successful Poll
2	3	-34.7227	1	Refine Mesh
3	5	-34.7227	0.5	Refine Mesh
4	8	-96.0847	1	Successful Poll

5	10	-96.0847	0.5	Refine Mesh
6	13	-132.888	1	Successful Poll
7	15	-132.888	0.5	Refine Mesh
8	17	-132.888	0.25	Refine Mesh
9	20	-197.689	0.5	Successful Poll
10	22	-197.689	0.25	Refine Mesh
11	24	-197.689	0.125	Refine Mesh
12	27	-241.344	0.25	Successful Poll
13	29	-241.344	0.125	Refine Mesh
14	31	-254.624	0.25	Successful Poll
15	33	-254.624	0.125	Refine Mesh
16	35	-254.624	0.0625	Refine Mesh
17	37	-254.624	0.03125	Refine Mesh
18	39	-254.624	0.01562	Refine Mesh
19	41	-254.624	0.007812	Refine Mesh
20	42	-256.009	0.01562	Successful Poll
21	44	-256.009	0.007812	Refine Mesh
22	47	-256.009	0.003906	Refine Mesh
23	50	-256.009	0.001953	Refine Mesh
24	53	-256.009	0.0009766	Refine Mesh
25	56	-256.009	0.0004883	Refine Mesh
26	59	-256.009	0.0002441	Refine Mesh
27	62	-256.009	0.0001221	Refine Mesh
28	65	-256.009	6.104e-05	Refine Mesh
29	68	-256.009	3.052e-05	Refine Mesh
30	71	-256.009	1.526e-05	Refine Mesh
Iter	Func-count	f(x)	MeshSize	Method
31	74	-256.009	7.629e-06	Refine Mesh
32	77	-256.009	3.815e-06	Refine Mesh
33	80	-256.009	1.907e-06	Refine Mesh
34	83	-256.009	9.537e-07	Refine Mesh

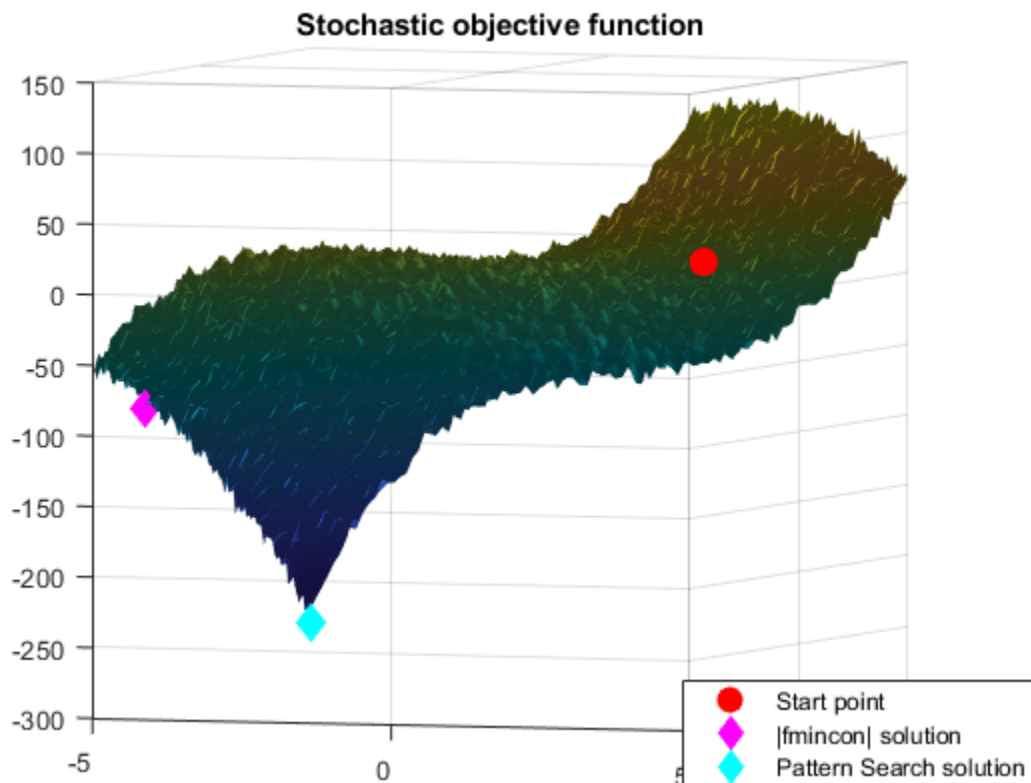
Optimization terminated: mesh size less than options.MeshTolerance.

Xps =

-4.9688 -5.0000

Fps =

-256.0095



Pattern search is not as strongly affected by random noise in the objective function. Pattern search requires only function values and not the derivatives, hence noise (of some uniform kind) may not affect it. However, pattern search requires more function evaluation to find the true minimum than derivative based algorithms, a cost for not using the derivatives.

See Also

More About

- "Global Optimization Toolbox Solver Characteristics" on page 1-30

Explore patternsearch Algorithms

Starting with R2022b, patternsearch has four algorithms:

- "classic"
- "nups" (Nonuniform Pattern Search)
- "nups-gps"
- "nups-mads"

This example shows how the choice of algorithm affects a bounded problem with a nonsmooth objective function.

Objective Function

The objective function for this example is based on the 2-D `ps_example` function that is available when you run the example.

```
type ps_example

function f = ps_example(x)
%PS_EXAMPLE objective function for patternsearch.

% Copyright 2003-2021 The MathWorks, Inc.

f = zeros(1,size(x,1));
for i = 1:size(x,1)
    if x(i,1) < -5
        f(i) = (x(i,1)+5)^2 + abs(x(i,2));
    elseif x(i,1) < -3
        f(i) = -2*sin(x(i,1)) + abs(x(i,2));
    elseif x(i,1) < 0
        f(i) = 0.5*x(i,1) + 2 + abs(x(i,2));
    elseif x(i,1) >= 0
        f(i) = .3*sqrt(x(i,1)) + 5/2 +abs(x(i,2));
    end
end
```

Extend the `ps_example` function to any even number of dimensions by creating pseudorandom offsets from the origin for every two dimensions, and adding the resulting objectives. See the code for the `testps` helper function at the [end of this example](#) on page 6-106.

Specify $N = 10$ dimensions, and set bounds of -20 to 20 for each component. Start patternsearch from the point $x_0 = [0 \ 0 \ \dots \ 0]$.

```
N = 10;
lb = -20*ones(1,N);
ub = lb;
x0 = zeros(size(lb));
```

Run Classic Algorithm

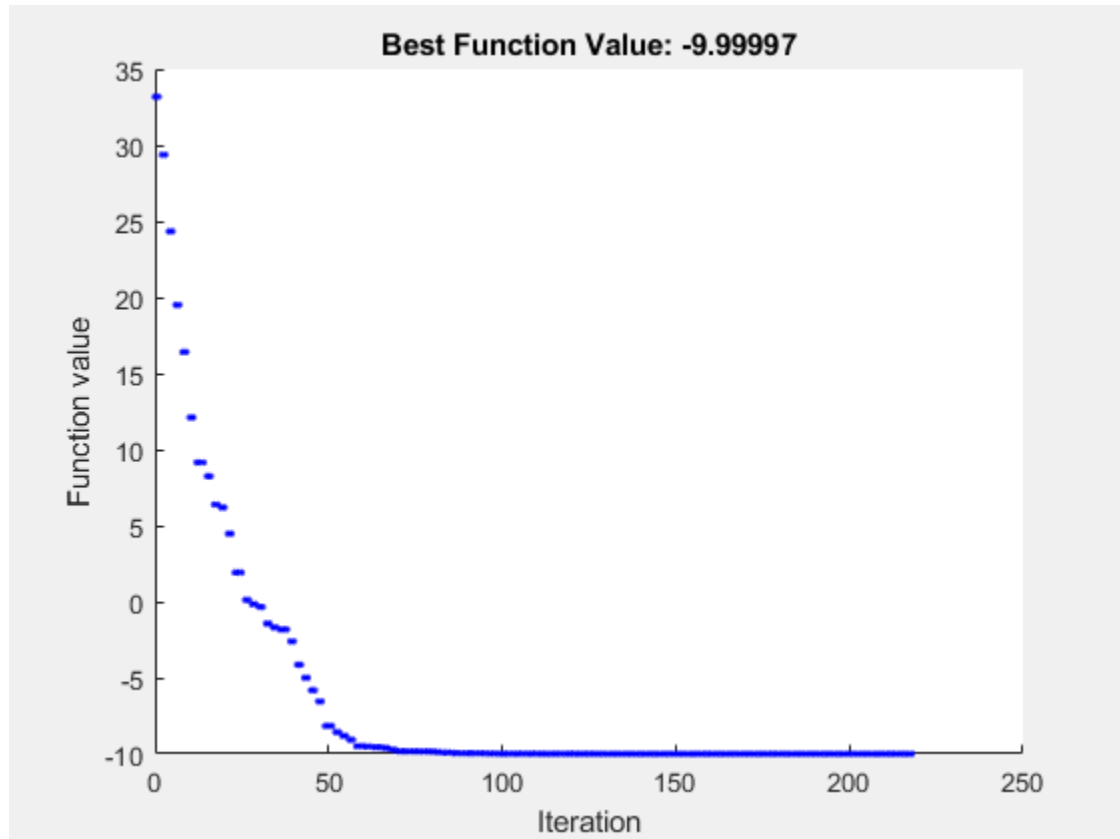
Set options to use the "classic" algorithm and to display the objective function values.

```
optscl = optimoptions("patternsearch",Algorithm="classic",PlotFcn="psplotbestf");
```

Run the optimization.

```
[xclassic,fvalclassic,eflagclassic,outputclassic] = ...
    patternsearch(@testps,x0,[],[],[],[],lb,ub,[],optscl)
```

Optimization terminated: mesh size less than options.MeshTolerance.



```
xclassic = 1×10
```

```
-7.9575    5.9058   -8.5047   -5.5481   -8.9402   -2.8633   -7.5058    0.8919   -5.6967    2.9
```

```
fvalclassic = -10.0000
```

```
eflagclassic = 1
```

```
outputclassic = struct with fields:
```

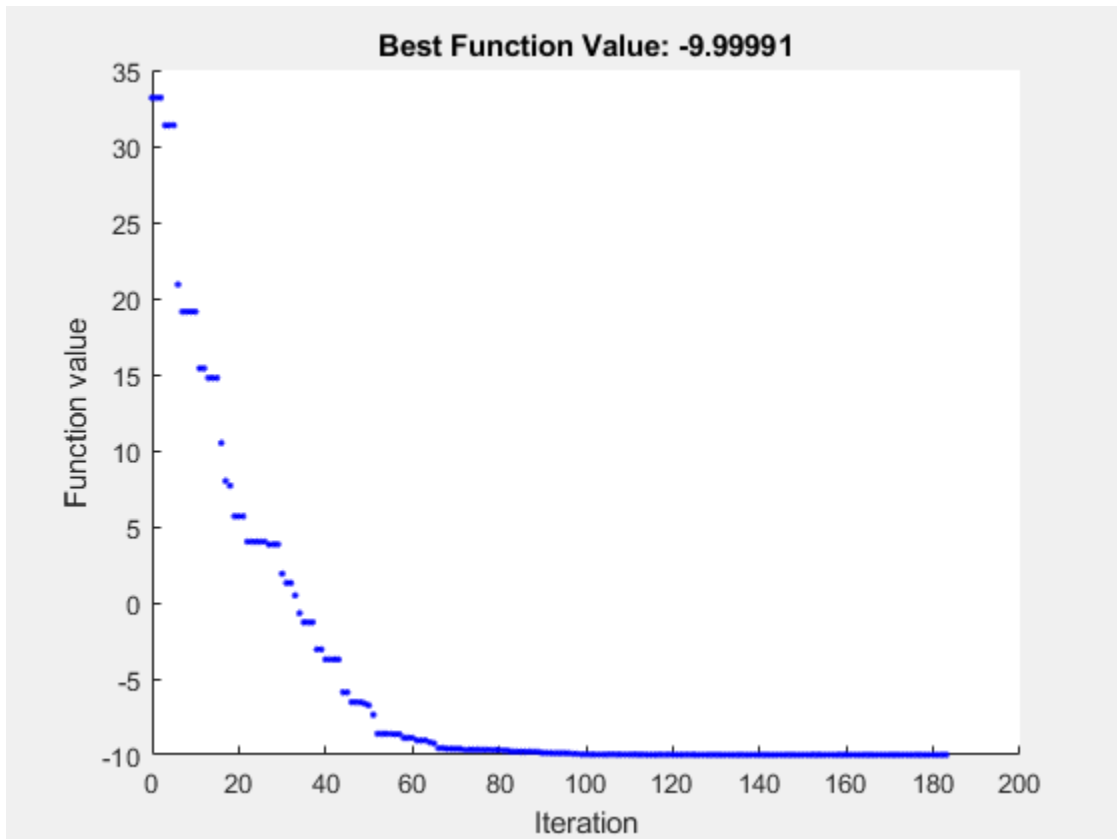
```
    function: @testps
    problemtype: 'boundconstraints'
    pollmethod: 'gpspositivebasis2n'
    maxconstraint: 0
    searchmethod: []
    iterations: 218
    funccount: 3487
    meshsize: 9.5367e-07
    rngstate: [1×1 struct]
    message: 'Optimization terminated: mesh size less than options.MeshTolerance.'
```

Run NUPS Algorithm

Set options to use the "nups" algorithm. To obtain reproducible results, set the random seed.

```
optsnups = optimoptions(optscl,Algorithm="nups");
rng default % For reproducibility
[xnups,fvalnups,eflagnups,outputnups] = ...
    patternsearch(@testps,x0,[],[],[],[],lb,ub,[],optsnups)
```

Optimization terminated: mesh size less than options.MeshTolerance.



```
xnups = 1×10
```

```
-7.9575    5.9058   -8.5047   -5.5480   -8.9401   -2.8633   -7.5058    0.8919   -5.6967    2.9
```

```
fvalnups = -9.9999
```

```
eflagnups = 1
```

```
outputnups = struct with fields:
    function: @testps
    problemtype: 'boundconstraints'
    pollmethod: 'nups'
    maxconstraint: 0
    searchmethod: []
    iterations: 183
    funccount: 1827
```

```

meshsize: 8.5928e-07
rngstate: [1x1 struct]
message: 'Optimization terminated: mesh size less than options.MeshTolerance.'
```

In this case, the "nups" algorithm reaches essentially the same solution as the "classic" algorithm while using fewer function evaluations.

Further Explorations

You can also try the "nups-gps" and "nups-mads" algorithms to see how they perform on this problem. Although you cannot always predict which algorithm works best on a problem, "nups" can be a good starting algorithm because it incorporates many adaptive features, making it likely to be the best algorithm.

Helper Function

This code creates the `testps` helper function.

```

function y = testps(x)
    N = numel(x); % Number of variables
    if mod(N,2) == 1
        disp("Number of variables must be even.")
        return
    end
    strm = RandStream("twister",Seed=1); % Set Seed for consistency
    % Use RandStream to avoid affecting the global stream
    dsp = 5*randn(strm,size(x));
    z = x - dsp; % Include random offsets
    y = 0;
    for i = 1:N/2
        y = y + ps_example(z([2*i-1,2*i]));
    end
end
```

See Also

`patternsearch`

Related Examples

- “Explore patternsearch Algorithms in Optimize Live Editor Task” on page 6-107
- “Nonuniform Pattern Search (NUPS) Algorithm” on page 6-35
- “How Pattern Search Polling Works” on page 6-27

Explore patternsearch Algorithms in Optimize Live Editor Task

Beginning in R2022b, patternsearch has four algorithm options:

- "classic"
- "nups" (Nonuniform Pattern Search)
- "nups-gps"
- "nups-mads"

This example shows how you can try the different patternsearch algorithms when solving a problem using the **Optimize** Live Editor task.

Specify Problem

Set up an optimization problem that has a quadratic plus linear objective function, bounds, and two linear constraints. Typically, `quadprog` is the best solver to use for this type of problem. However, this example uses `patternsearch` so you can try its different algorithms.

Set the number of variables for this problem to $N = 10$. Create a pseudorandom symmetric matrix Q of size N -by- N and a pseudorandom vector z of length N for the objective function $\text{fun}(x) = x*Q*x' - N*x*z$.

```
N = 10;
rng default
x0 = rand(1,N);
x0 = x0/(2*sum(x0));
Q = 6*eye(N) + randn(N);
Q = (Q + Q');
z = rand(N,1);
fun = @(x)x*Q*x' - N*x*z;
```

Set linear constraints on the problem: $\text{sum}(x) \leq 1$ and $\text{sum}(i*x) \leq N/3$, where i is the index of the vector x .

```
A = [ones(1,N);1:N]; % sum(x) <= 1, sum(i*x) <= N/3
b = [1;N/3];
```

Confirm that Q is positive definite, so that the problem is convex.

```
eig(Q)

ans = 10×1

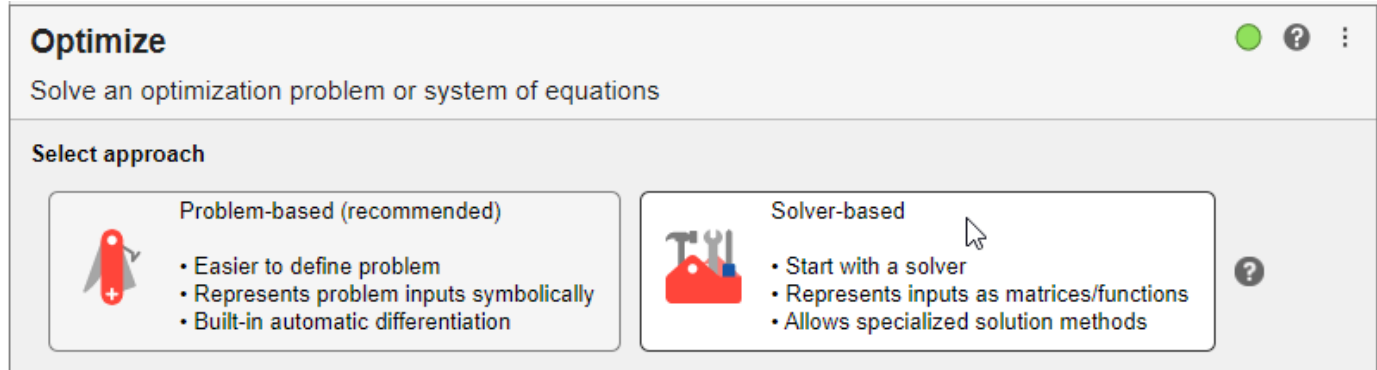
    1.5071
    4.0347
    7.3749
    9.0561
   11.6929
   11.9473
   13.0976
   15.2099
   16.1175
   18.8949
```

Create Problem in Optimize Live Editor Task

Open a new or existing live script. In the **Code** section of the **Live Editor** tab, click **Task** to open the gallery of Live Editor tasks. Under **Optimization**, click **Optimize**.

Select Approach

In the **Optimize** Live Editor task, select the solver-based approach.



Specify Problem Type

Select these options to specify the type of problem:

- Objective — Nonlinear
- Constraints — Lower bounds, upper bounds, and linear inequality
- Solver — patternsearch

Select Problem Data

Select these options for the problem data:

- Objective function — Click the **From file** arrow and select **Function handle**. Click the **select** arrow and select **fun**.
- Initial point — x_0
- Constraints — Lower bounds 0
- Constraints — Upper bounds 1
- Constraints — Linear inequality constraint arrays **A** and **b**

Specify Solver Options

Click the arrow to expand the **Specify solver options** section of the task. Then, click the **Add** button. The task specifies the **classic** algorithm for the algorithm settings.

Display Progress

Select two plots to display: **Best value** and **Evaluation count**.

Your **Optimize** Live Editor task should match the one in the figure below.

Optimize ● ? ⋮

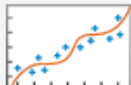
solution, objectiveValue = Minimize **fun** using **patternsearch** solver

Specify problem type

Objective


Linear


Quadratic


Least squares



Nonlinear



Nonsmooth

Examples: $f(x, y) = xy$, $f(x) = \cos(x)$, $f(x) = \log(x)$, $f(x) = e^x$, $f(x) = x^3$, Solve $F(x) = 0$, ...


Constraints

 Unconstrained


 Lower bounds

 Upper bounds

 Linear inequality

 Linear equality

 Second-order cone

 Nonlinear

 Integer

Examples: $x \geq 0$, $x \leq 2$, $x + y \leq 5$

Solver

patternsearch - Pattern search
▼
?

Select problem data

Objective function Function handle ▼ fun ▼ ?

Initial point (x0) x0 ▼

Constraints

Lower bounds All bounds the same ▼ 0 ≤ x

Upper bounds All bounds the same ▼ 1 ≥ x

Linear inequality A A ▼ * $x \leq$ b b ▼

Specify solver options

?

Algorithm settings ▼

Algorithm ▼

classic ▼

-
+

Display progress

Text display Final output ▼

Plot

Best value

Mesh size

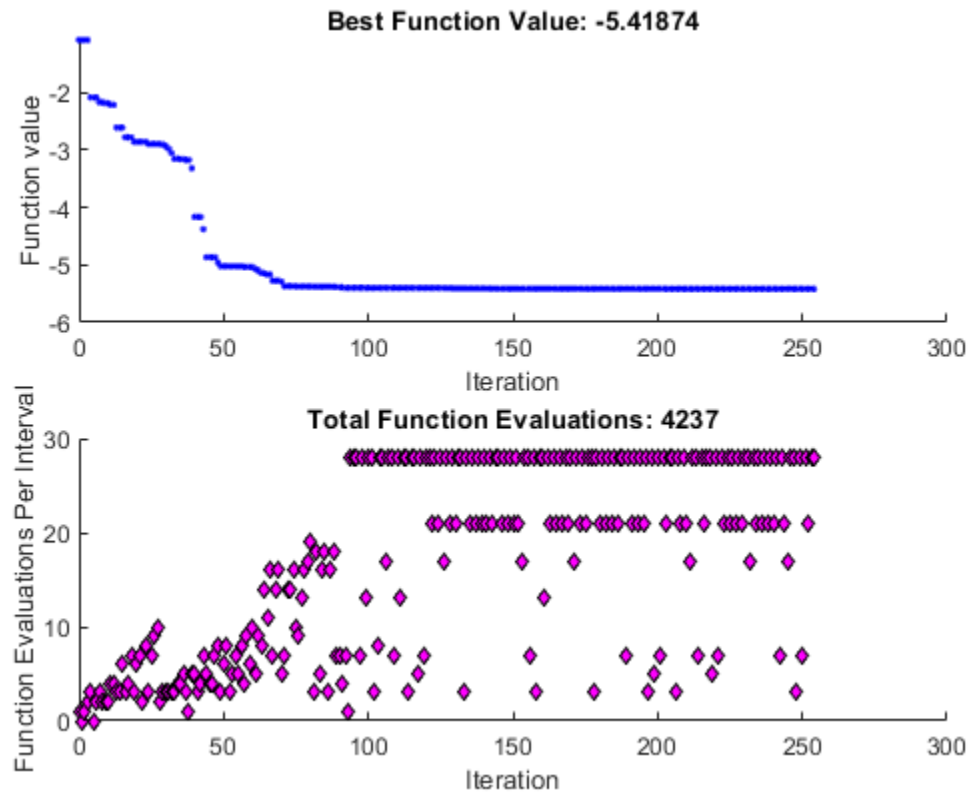
Evaluation count

Best point

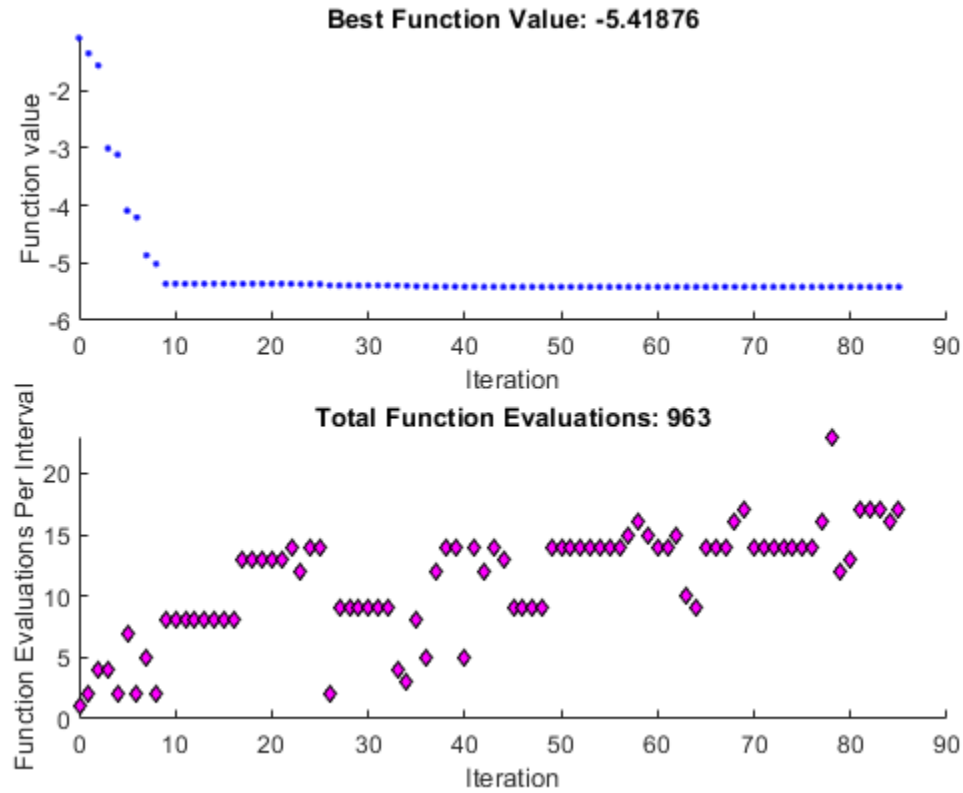
Max constraint violation

Find Solution Using Different Algorithms

After selecting the options for your problem, run the solver by clicking **Run** in the **Run** section of the Live Editor tab. The solver runs the "classic" patternsearch algorithm and displays the two specified plots.



Change the algorithm to **nups**. The solver runs the new algorithm and displays the two specified plots.




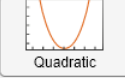
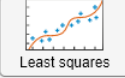
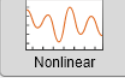
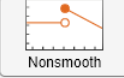
The solver finishes running the "nups" algorithm in about a quarter of the number of function evaluations as the "classic" algorithm, and reaches a slightly better (lower) objective function value.

Working in the **Optimize** Live Editor task, you can continue to explore the other patternsearch algorithms and plot functions, or other options and solvers.

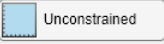
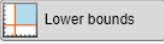
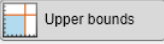
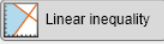
Optimize ● ? ⋮

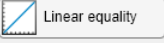
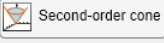
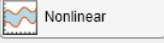
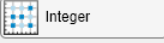
`solution`, `objectiveValue` = Minimize `fun` using `patternsearch` solver

▼ Specify problem type

Objective  Linear  Quadratic  Least squares  Nonlinear  Nonsmooth

Examples: $f(x, y) = x/y$, $f(x) = \cos(x)$, $f(x) = \log(x)$, $f(x) = e^x$, $f(x) = x^3$, Solve $F(x) = 0, \dots$

Constraints  Unconstrained  Lower bounds  Upper bounds  Linear inequality

 Linear equality  Second-order cone  Nonlinear  Integer

Examples: $x \geq 0, x \leq 2, x + y \leq 5$

Solver `patternsearch - Pattern search` ?

▼ Select problem data

Objective function `Function handle` `fun` ?

Initial point (x0) `x0`

Constraints Lower bounds `All bounds the same` `0` $\leq x$
 Upper bounds `All bounds the same` `1` $\geq x$
 Linear inequality A `A` $*x \leq b$ `b`

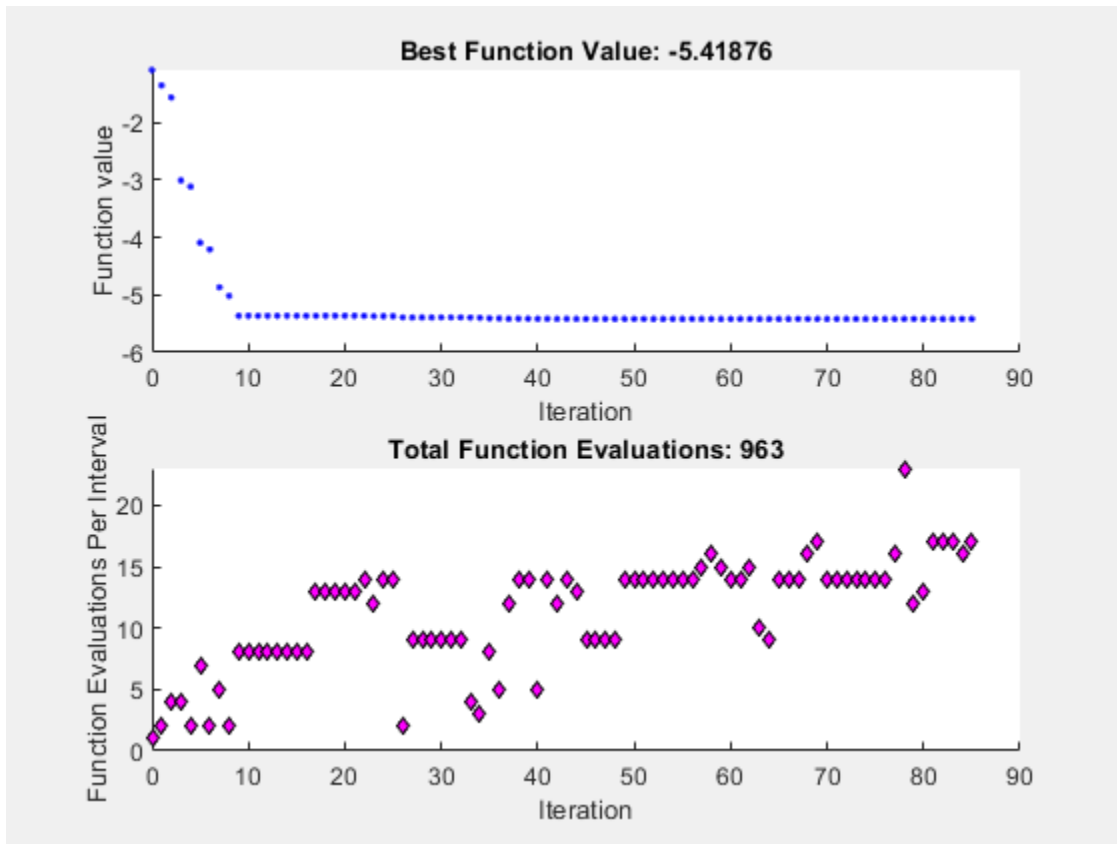
► Specify solver options

▼ Display progress

Text display `Final output`

Plot Best value Mesh size Evaluation count Best point
 Max constraint violation

Optimization terminated: mesh size less than options.MeshTolerance.



See Also

patternsearch

Related Examples

- "Explore patternsearch Algorithms" on page 6-103
- "Nonuniform Pattern Search (NUPS) Algorithm" on page 6-35
- "How Pattern Search Polling Works" on page 6-27

Problem-Based Direct Search

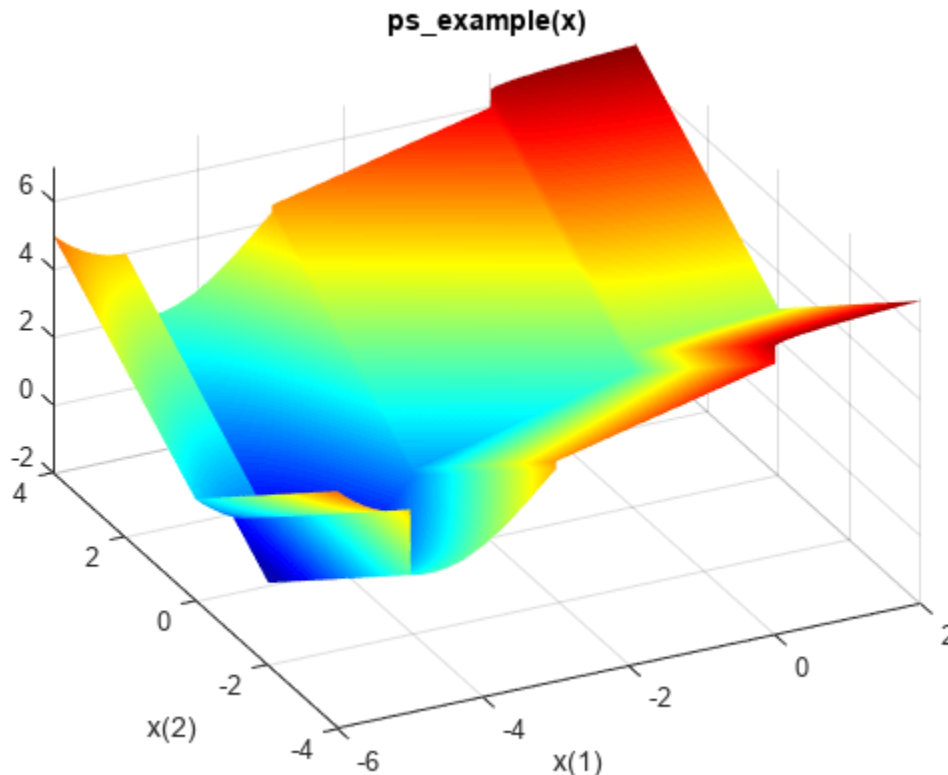
- “Optimize Nonsmooth Function Using patternsearch, Problem-Based” on page 7-2
- “Constrained Minimization Using Pattern Search, Problem-Based” on page 7-4
- “Effects of Pattern Search Options, Problem-Based” on page 7-10
- “Search and Poll, Problem-Based” on page 7-16

Optimize Nonsmooth Function Using patternsearch, Problem-Based

This example shows how to minimize a nonsmooth function using direct search in the problem-based approach. The function to minimize, `ps_example(x)`, is included when you run this example.

Plot the objective function.

```
fsurf(@(x,y)reshape(ps_example([x(:),y(:)]),size(x)),...
      [-6 2 -4 4],"LineStyle","none","MeshDensity",300)
colormap 'jet'
view(-26,43)
xlabel("x(1)")
ylabel("x(2)")
title("ps_example(x)")
```



Create a 2-D optimization variable `x`. The `ps_example` function expects the variable to be a row vector, so specify `x` as a 2-element row vector.

```
x = optimvar("x",1,2);
```

To use `ps_example` as the objective function, convert the function to an optimization expression using `fcn2optimexpr`.

```
fun = fcn2optimexpr(@ps_example,x);
```

Create an optimization problem with objective function `ps_example`.

```
prob = optimproblem("Objective", fun);
```

Specify the initial point `x0` as a structure with field `x` taking the value `[2.1 1.7]`.

```
x0.x = [2.1 1.7];
```

Solve the problem, specifying the `patternsearch` solver.

```
[sol, fval] = solve(prob, x0, "Solver", "patternsearch")
```

```
Solving problem using patternsearch.
```

```
Optimization terminated: mesh size less than options.MeshTolerance.
```

```
sol = struct with fields:  
  x: [-4.7124 -7.6294e-07]
```

```
fval = -2.0000
```

`patternsearch` finds a better solution (lower function value) than the default `fminunc` solver, which is not recommended for minimizing nonsmooth functions.

```
[solfminunc, fvalfminunc] = solve(prob, x0)
```

```
Solving problem using fminunc.
```

```
Local minimum possible.
```

```
fminunc stopped because it cannot decrease the objective function  
along the current search direction.
```

```
solfminunc = struct with fields:  
  x: [1.9240 8.8818e-16]
```

```
fvalfminunc = 2.9161
```

See Also

[patternsearch](#) | [fcn2optimexpr](#) | [solve](#)

Related Examples

- “Direct Search”

Constrained Minimization Using Pattern Search, Problem-Based

This example shows how to minimize an objective function, subject to nonlinear inequality constraints and bounds, using pattern search in the problem-based approach. For a solver-based version of this problem, see “Constrained Minimization Using Pattern Search, Solver-Based” on page 6-14.

Constrained Minimization Problem

For this problem, the objective function to minimize is a simple function of 2-D variables X and Y:

$$\text{camxy} = @(X,Y)(4 - 2.1.*X.^2 + X.^4./3).*X.^2 + X.*Y + (-4 + 4.*Y.^2).*Y.^2;$$

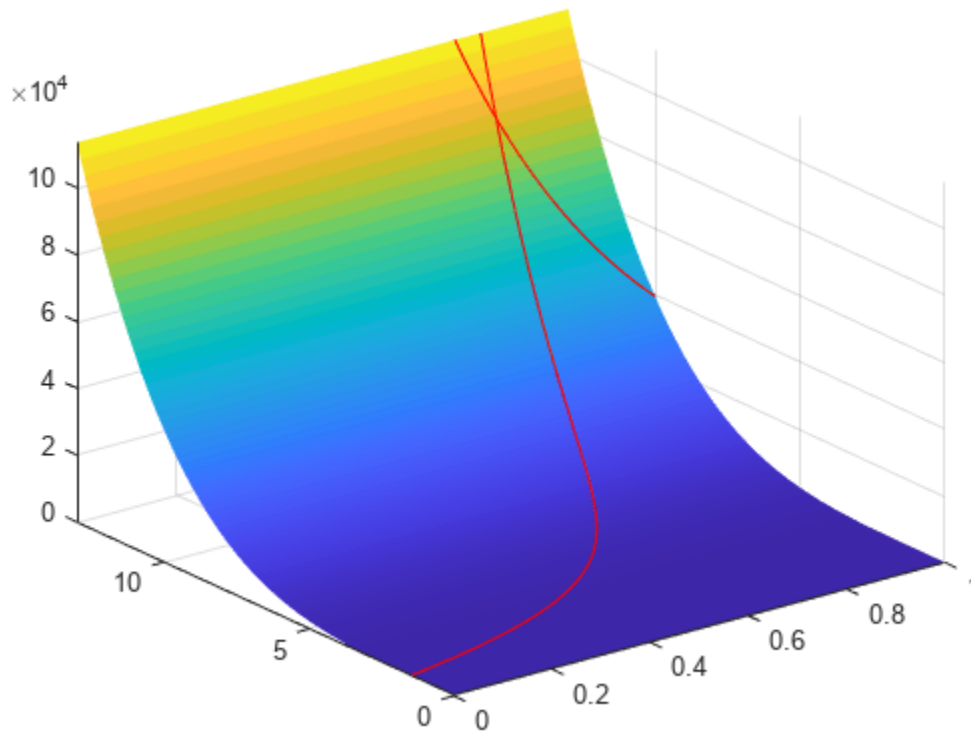
This function is known as "cam," as described in L.C.W. Dixon and G.P. Szego [1] on page 7-9.

Additionally, the problem has nonlinear constraints and bounds.

$$\begin{aligned} x(1)*x(2) + x(1) - x(2) + 1.5 &\leq 0 && \text{(nonlinear constraint)} \\ 10 - x(1)*x(2) &\leq 0 && \text{(nonlinear constraint)} \\ 0 &\leq x(1) \leq 1 && \text{(bound)} \\ 0 &\leq x(2) \leq 13 && \text{(bound)} \end{aligned}$$

Plot the nonlinear constraint region on a surface plot of the objective function. The constraints limit the solution to the small region above both red curves.

```
x1 = linspace(0,1);
y1 = (-x1 - 1.5)./(x1 - 1);
y2 = 10./x1;
[X,Y] = meshgrid(x1,linspace(0,13));
Z = camxy(X,Y);
surf(X,Y,Z,"LineStyle","none")
hold on
z1 = camxy(x1,y1);
z2 = camxy(x1,y2);
plot3(x1,y1,z1,'r-',x1,y2,z2,'r-')
xlim([0 1])
ylim([0 13])
zlim([0,max(Z,[],"all")])
hold off
```

Create Optimization Variables, Problem, and Constraints

To set up this problem, create optimization variables x and y . Set the bounds as you create the variables.

```
x = optimvar("x", "LowerBound", 0, "UpperBound", 1);
y = optimvar("y", "LowerBound", 0, "UpperBound", 13);
```

Create the objective as an optimization expression.

```
cam = camxy(x,y);
```

Create an optimization problem with this objective function.

```
prob = optimproblem("Objective", cam);
```

Create the two nonlinear inequality constraints, and include them in the problem.

```
prob.Constraints.cons1 = x*y + x - y + 1.5 <= 0;
prob.Constraints.cons2 = 10 - x*y <= 0;
```

Review the problem.

```
show(prob)
```

```
OptimizationProblem :
```

```
Solve for:
```

```

x, y

minimize :
    (((((4 - (2.1 .* x.^2)) + (x.^4 ./ 3)) .* x.^2) + (x .* y)) + (((-4) + (4 .* y.^2)) .* y)

subject to cons1:
    (((x .* y) + x) - y) + 1.5) <= 0

subject to cons2:
    (10 - (x .* y)) <= 0

variable bounds:
    0 <= x <= 1

    0 <= y <= 13

```

Set Initial Point and Solve

Set the initial point as a structure with field `x` equal to 0.5 and `y` equal to 0.5.

```

x0.x = 0.5;
x0.y = 0.5;

```

Solve the problem specifying the `patternsearch` solver.

```
[sol,fval] = solve(prob,x0,"Solver","patternsearch")
```

```

Solving problem using patternsearch.
Optimization finished: mesh size less than options.MeshTolerance
and constraint violation is less than options.ConstraintTolerance.

```

```

sol = struct with fields:
    x: 0.8122
    y: 12.3122

```

```
fval = 9.1324e+04
```

`patternsearch` finds the solution point `x = 0.8122`, `y = 12.3122` with objective function value `9.1324e4`.

Add Visualization

To observe the solver's progress, specify options that select two plot functions. The plot function `psplotbestf` plots the best objective function value at every iteration, and the plot function `psplotmaxconstr` plots the maximum constraint violation at every iteration. Set these two plot functions in a cell array. Also, display information about the solver's progress in the Command Window by setting the `Display` option to `'iter'`.

```

options = optimoptions(@patternsearch,...
    "PlotFcn",{@psplotbestf,@psplotmaxconstr},...
    "Display","iter");

```

Run the solver, including the `options` argument.

```

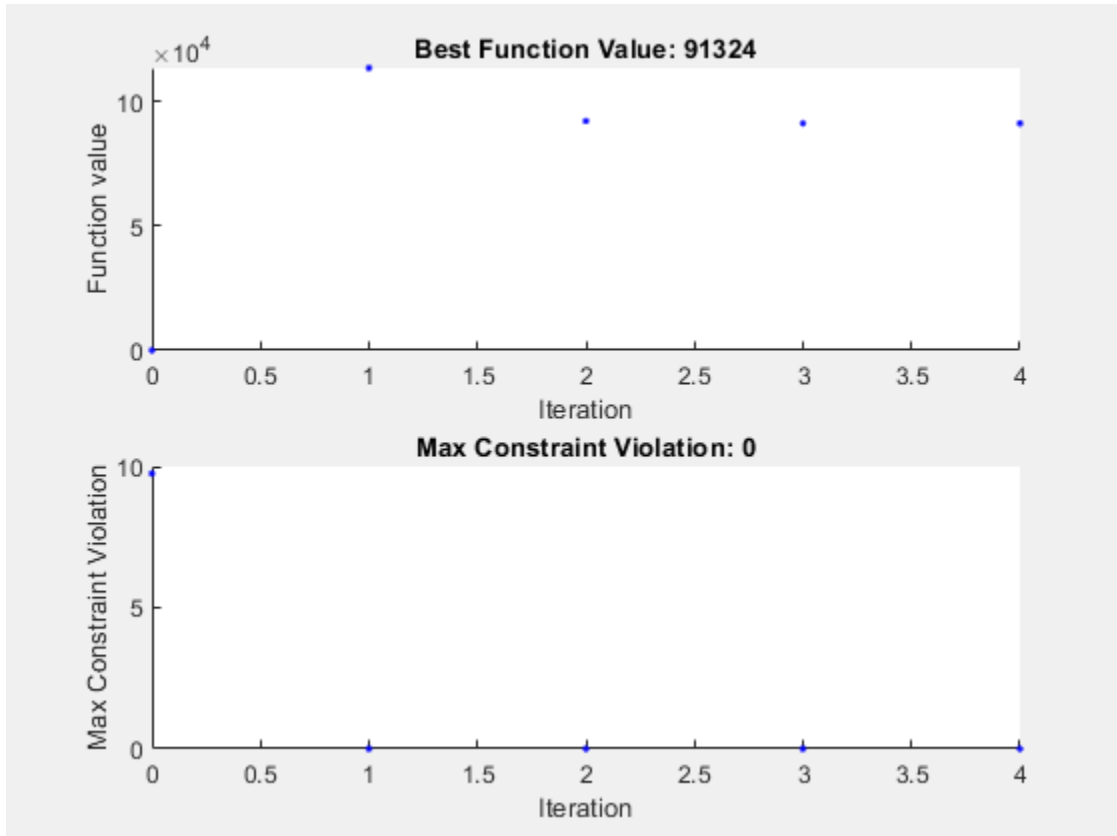
[sol,fval] = solve(prob,x0,"Solver","patternsearch","Options",options)

Solving problem using patternsearch.

```

Iter	Func-count	f(x)	Max Constraint	MeshSize	Method
0	1	0.373958	9.75	0.9086	
1	18	113581	1.617e-10	0.001	Increase penalty
2	147	92267	0	1e-05	Increase penalty
3	373	91333.2	0	1e-07	Increase penalty
4	638	91324	0	1e-09	Increase penalty

Optimization finished: mesh size less than options.MeshTolerance and constraint violation is less than options.ConstraintTolerance.



```
sol = struct with fields:
  x: 0.8122
  y: 12.3122
```

```
fval = 9.1324e+04
```

Nonlinear constraints cause `patternsearch` to solve many subproblems at each iteration. As shown in both the plots and the iterative display, the solution process has few iterations. However, the `Func-count` column in the iterative display shows many function evaluations per iteration. Both the plots and the iterative display show that the initial point is infeasible, and that the objective function is low at the initial point. During the solution process, the objective function value initially increases, then decreases to its final value.

Unsupported Functions

If your objective or nonlinear constraint functions are not “Supported Operations for Optimization Variables and Expressions”, use `fcn2optimexpr` to convert them to a form suitable for the problem-

based approach. For example, suppose that instead of the constraint $xy \geq 10$ you have the constraint $I_1(x) + I_1(y) \geq 10$, where $I_1(x)$ is the modified Bessel function `besseli(1,x)`. (The Bessel functions are not supported functions.) Create this constraint using `fcn2optimexpr` as follows. First create an optimization expression for $I_1(x) + I_1(y)$.

```
bfun = fcn2optimexpr(@(t,u)besseli(1,t) + besseli(1,u),x,y);
```

Next, replace the constraint `cons2` with the constraint `bfun >= 10`.

```
prob.Constraints.cons2 = bfun >= 10;
```

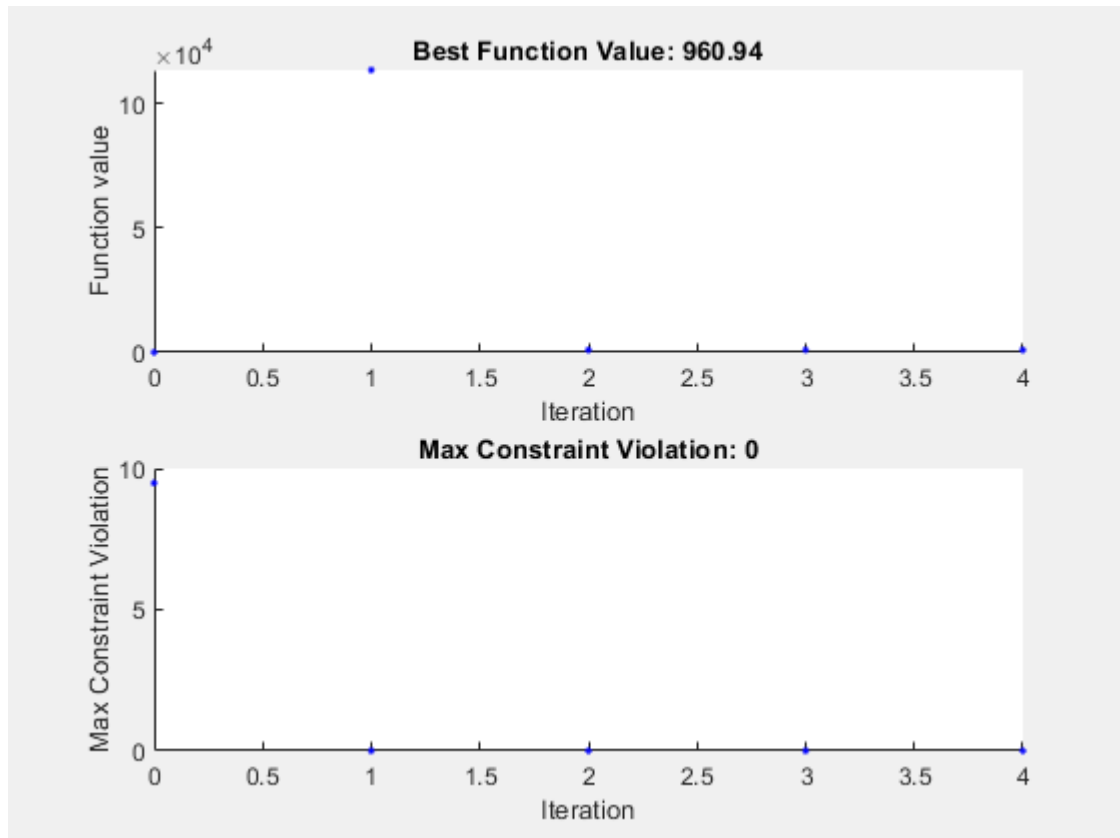
Solve the problem. The solution differs because the constraint region is different.

```
[sol2,fval2] = solve(prob,x0,"Solver","patternsearch","Options",options)
```

Solving problem using patternsearch.

Iter	Func-count	f(x)	Max Constraint	MeshSize	Method
0	1	0.373958	9.484	0.9307	
1	18	113581	0	0.001	Increase penalty
2	183	962.841	0	1e-05	Increase penalty
3	499	960.942	0	1e-07	Increase penalty
4	636	960.94	0	8.511e-15	Update multipliers

Optimization finished: mesh size less than options.MeshTolerance and constraint violation is less than options.ConstraintTolerance.



```
sol2 = struct with fields:  
  x: 0.4998  
  y: 3.9981
```

```
fval2 = 960.9401
```

References

[1] Dixon, L. C. W., and G .P. Szego (eds.). *Towards Global Optimisation 2*. North-Holland: Elsevier Science Ltd., Amsterdam, 1978.

See Also

patternsearch | solve

Related Examples

- “Direct Search”
- “Constrained Minimization Using ga, Problem-Based” on page 9-19
- “Constrained Minimization Using Pattern Search, Solver-Based” on page 6-14

Effects of Pattern Search Options, Problem-Based

This example shows the effects of some options for pattern search in the problem-based approach. The options include plotting, stopping criteria, and other algorithmic controls for speeding a solution. For a list of available options for patternsearch algorithms, see “Options Table for Pattern Search Algorithms” on page 17-21.

Set Up a Problem for Pattern Search

The problem to minimize is a quadratic function of six variables subject to linear equality and inequality constraints. The objective function, `lincontest7`, is available when you run this example.

type `lincontest7`

```
function y = lincontest7(x)
%LINCONTEST7 objective function.
% y = LINCONTEST7(X) evaluates y for the input X. Make sure that x is a column
% vector, whereas objective function gets a row vector.

% Copyright 2003-2017 The MathWorks, Inc.
x = x(:);
```

```
%Define a quadratic problem in terms of H and f
H = [36 17 19 12 8 15;
     17 33 18 11 7 14;
     19 18 43 13 8 16;
     12 11 13 18 6 11;
     8 7 8 6 9 8;
     15 14 16 11 8 29];
```

```
f = [ 20 15 21 18 29 24 ]';
```

```
y = 0.5*x'*H*x + f'*x;
```

Create a six-element optimization variable `x` as a row vector.

```
x = optimvar("x",1,6);
```

Create an optimization problem with the objective function `lincontest7(x)`.

```
prob = optimproblem("Objective",lincontest7(x));
```

Specify an initial point for the optimization.

```
x0.x = [2 1 0 9 1 0];
```

Create linear constraint matrices for the constraints $A_{\text{ineq}}x' \leq B_{\text{ineq}}$ and $A_{\text{eq}}x' = B_{\text{eq}}$. You need to use `x'` in these constraints because `x` is a row vector.

```
Aineq = [-8 7 3 -4 9 0 ];
Bineq = [7];
Aeq = [7 1 8 3 3 3; 5 0 5 1 5 8; 2 6 7 1 1 8; 1 0 0 0 0 0];
Beq = [84 62 65 1]';
prob.Constraints.Aineq = Aineq*x' <= Bineq;
prob.Constraints.Aeq = Aeq*x' == Beq;
```

Run the `patternsearch` solver, and note the number of iterations and function evaluations required to reach the solution.

```
[sol,Fval,eflag,output] = solve(prob,x0,"Solver","patternsearch");
```

```
Solving problem using patternsearch.
Optimization terminated: mesh size less than options.MeshTolerance.
```

```
fprintf('The number of iterations is: %d\n', output.iterations);
```

```
The number of iterations is: 148
```

```
fprintf('The number of function evaluations is: %d\n', output.funccount);
```

```
The number of function evaluations is: 825
```

```
fprintf('The best function value found is: %g\n', Fval);
```

```
The best function value found is: 2189.18
```

Add Visualization

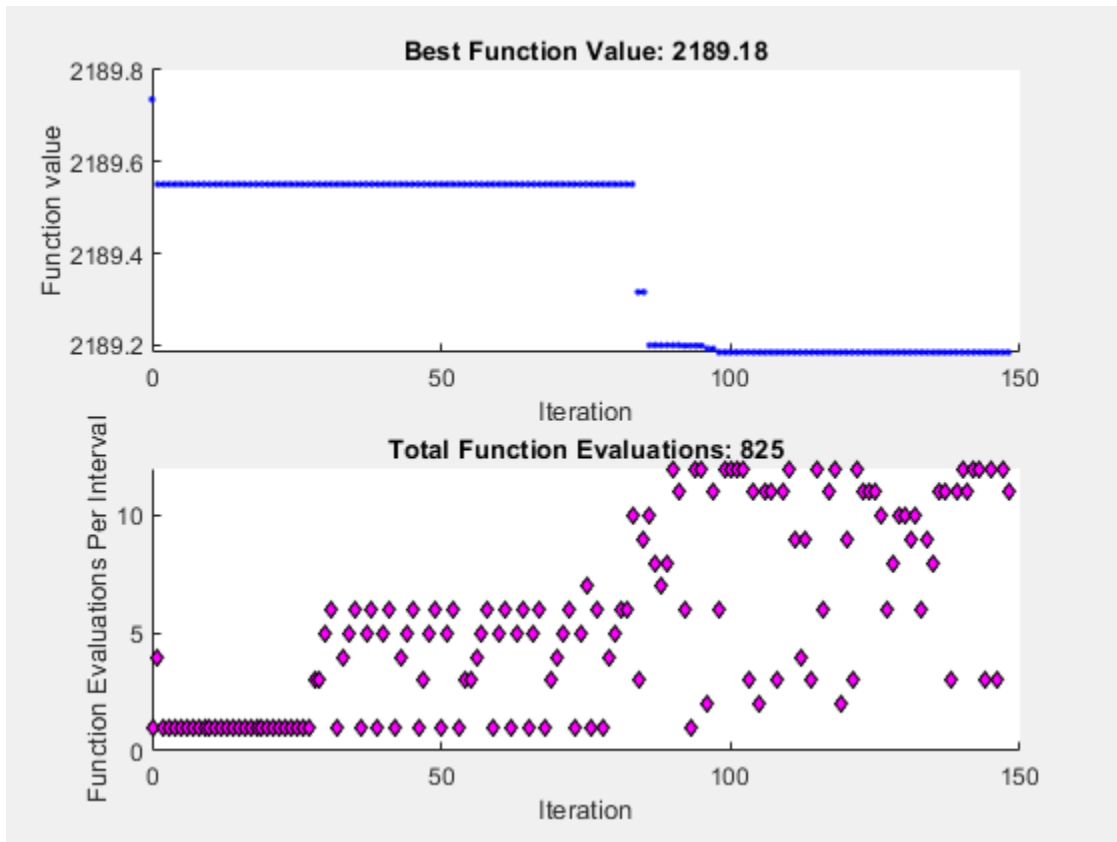
Monitor the optimization process by specifying options that select two plot functions. The plot function `psplotbestf` plots the best objective function value at every iteration, and the plot function `psplotfunccount` plots the number of times the objective function is evaluated at each iteration. Set these two plot functions in a cell array.

```
opts = optimoptions(@patternsearch,"PlotFcn",{@psplotbestf,@psplotfunccount});
```

Run the `patternsearch` solver, including the `opts` argument.

```
[sol2,Fval2,eflag2,output2] = solve(prob,x0,"Solver","patternsearch",...
    "Options",opts);
```

```
Solving problem using patternsearch.
Optimization terminated: mesh size less than options.MeshTolerance.
```



Mesh Options

Pattern search involves evaluating the objective function at points in a mesh. The size of the mesh can influence the speed of the solution. You can control the size of the mesh by setting options.

Initial Mesh Size

The mesh at each iteration is the span of a set of search directions that are added to the current point, scaled by the current mesh size. The solver starts with an initial mesh size of 1 by default. To start with the initial mesh size of 1/2, set the `InitialMeshSize` option. This can save an iteration and several function evaluations when the initial point is good relative to a mesh of size 1.

```
opts = optimoptions(opts, 'InitialMeshSize', 1/2);
```

Mesh Scaling

You can scale the mesh to improve the minimization of a poorly scaled optimization problem. Scaling rotates the pattern by some degree and scales along the search directions. The `ScaleMesh` option is on (`true`) by default, but you can turn it off if the problem is well scaled. In general, if the problem is poorly scaled, setting this option to `true` can reduce the number of function evaluations. For this problem, set `ScaleMesh` to `false`, because `lincontest7` is a well-scaled objective function.

```
opts = optimoptions(opts, 'ScaleMesh', false);
```

Mesh Accelerator

Direct search methods require many function evaluations compared to derivative-based optimization methods. The pattern search algorithm can quickly find the neighborhood of an optimum point, but

can be slow in detecting the minimum itself. The `patternsearch` solver can reduce the number of function evaluations by using an accelerator. When the accelerator is on (`opts.AccelerateMesh = true`), the solver contracts the mesh size rapidly after reaching a minimum mesh size. This option is recommended only for smooth problems; in other types of problems, you can lose some accuracy. The `AccelerateMesh` option is off (`false`) by default. For this problem, set `AccelerateMesh` to `true` because the objective function is smooth.

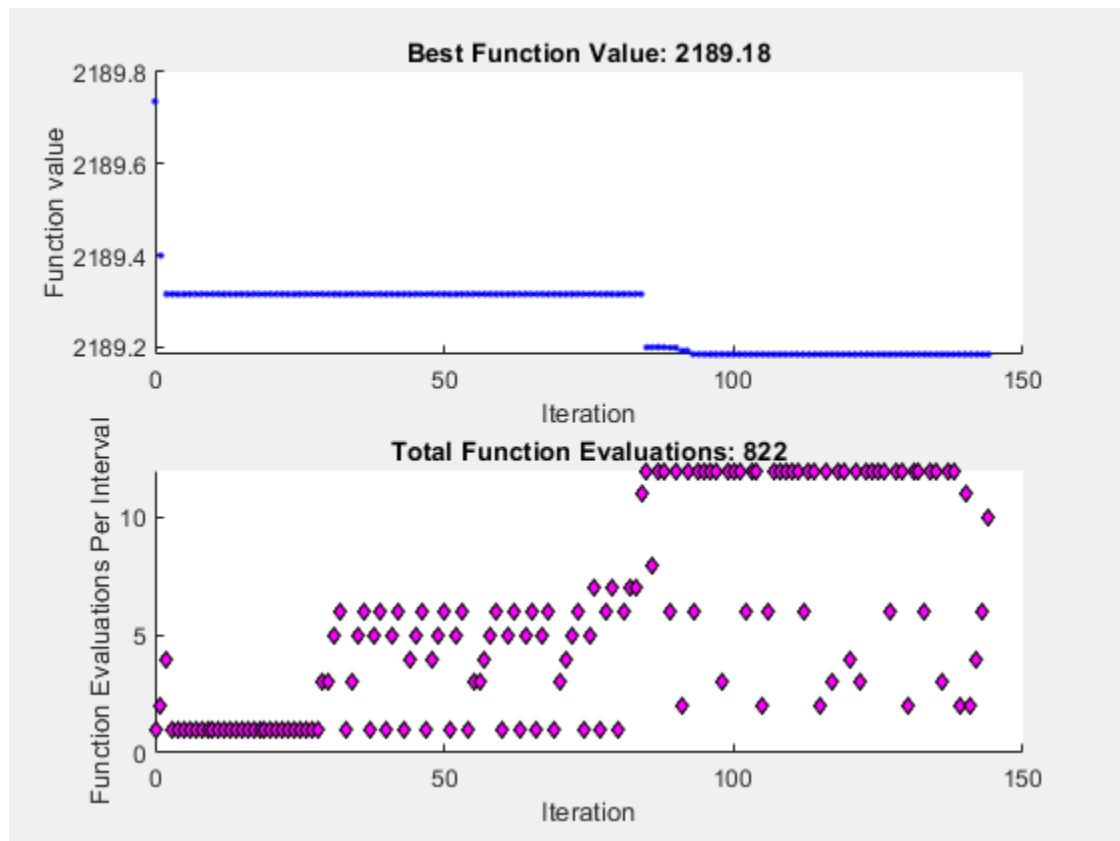
```
opts = optimoptions(opts,'AccelerateMesh',true);
```

Run the `patternsearch` solver.

```
[sol3,Fval3,eflag3,output3] = solve(prob,x0,"Solver","patternsearch",...
    "Options",opts);
```

Solving problem using `patternsearch`.

Optimization terminated: mesh size less than options.MeshTolerance.



```
fprintf('The number of iterations is: %d\n', output3.iterations);
```

The number of iterations is: 144

```
fprintf('The number of function evaluations is: %d\n', output3.funccount);
```

The number of function evaluations is: 822

```
fprintf('The best function value found is: %g\n', Fval3);
```

The best function value found is: 2189.18

The mesh option settings reduce the number of iterations and the number of function evaluations, and with no apparent loss of accuracy.

Stopping Criteria and Tolerances

`MeshTolerance` is a tolerance on the mesh size. If the mesh size is less than `MeshTolerance`, the solver stops. `StepTolerance` is the minimum tolerance on the change in the current point to the next point. `FunctionTolerance` is the minimum tolerance on the change in the function value from the current point to the next point.

Set the `MeshTolerance` to `1e-7`, which is 10 times smaller than the default value. This setting can increase the number of function evaluations and iterations, and can lead to a more accurate solution.

```
opts.MeshTolerance = 1e-7;
```

Search Methods in Pattern Search

The pattern search algorithm can use an additional search method at every iteration, based on the value of the `SearchFcn` option. When you specify a search method using `SearchFcn`, `patternsearch` performs the specified search first, before the mesh search. If the search method is successful, `patternsearch` skips the mesh search, commonly called the poll function, for that iteration. If the search method is unsuccessful in improving the current point, `patternsearch` performs the mesh search.

You can specify different search methods for `SearchFcn`, including `searchga` and `searchneldermead`, which are optimization algorithms. Use these two search methods only for the first iteration, which is the default setting. Using either of these methods at every iteration might not improve the results and can be computationally expensive. However, you can use the `searchlhs` method, which generates Latin hypercube points, at every iteration or possibly every 10 iterations.

Other choices for search methods include poll methods such as positive basis $N+1$ or positive basis $2N$. A recommended strategy is to use positive basis $N+1$ (which requires at most $N+1$ points to create a pattern) as a search method and positive basis $2N$ (which requires $2N$ points to create a pattern) as a poll method.

Update the options structure to use `positivebasisnp1` as the search method. Because positive basis $2N$ is the default for the `PollFcn` option, do not set that option.

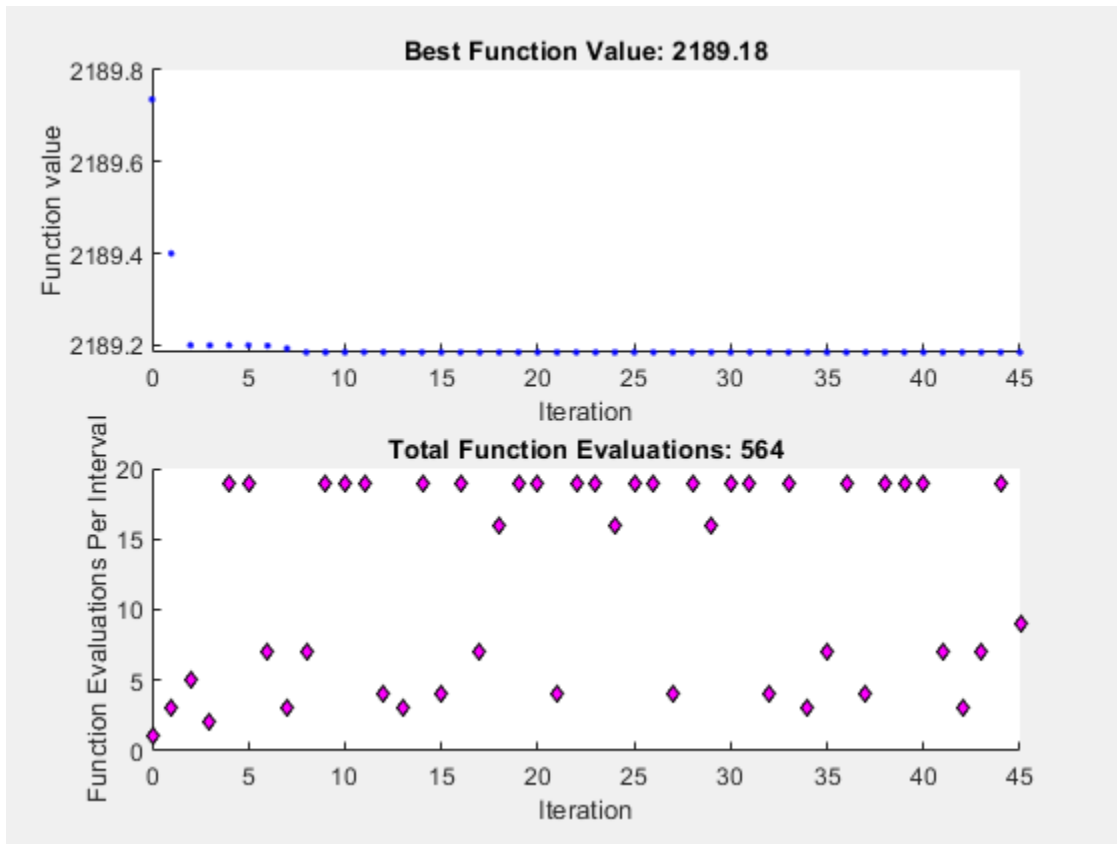
```
opts.SearchFcn = @positivebasisnp1;
```

Run the `patternsearch` solver.

```
[sol4,Fval4,eflag4,output4] = solve(prob,x0,"Solver","patternsearch",...  
    "Options",opts);
```

Solving problem using `patternsearch`.

Optimization terminated: change in X less than `options.StepTolerance`.



```
fprintf('The number of iterations is: %d\n', output4.iterations);
```

The number of iterations is: 45

```
fprintf('The number of function evaluations is: %d\n', output4.funccount);
```

The number of function evaluations is: 564

```
fprintf('The best function value found is: %g\n', Fval4);
```

The best function value found is: 2189.18

The total number of iterations and function evaluations decreases, even though the mesh tolerance is smaller than its previous value and is the stopping criterion that halts the solver.

See Also

`patternsearch` | `solve`

Related Examples

- “Effects of Pattern Search Options” on page 6-18
- “Search and Poll” on page 6-42
- “Pattern Search Options” on page 17-7

Search and Poll, Problem-Based

In addition to polling the mesh points, the pattern search algorithm can perform an optional step at every iteration, called search. At each iteration, the search step applies another optimization method to the current point. If this search does not improve the current point, the poll step is performed.

The objective function, `lincontest7`, is available when you run this example.

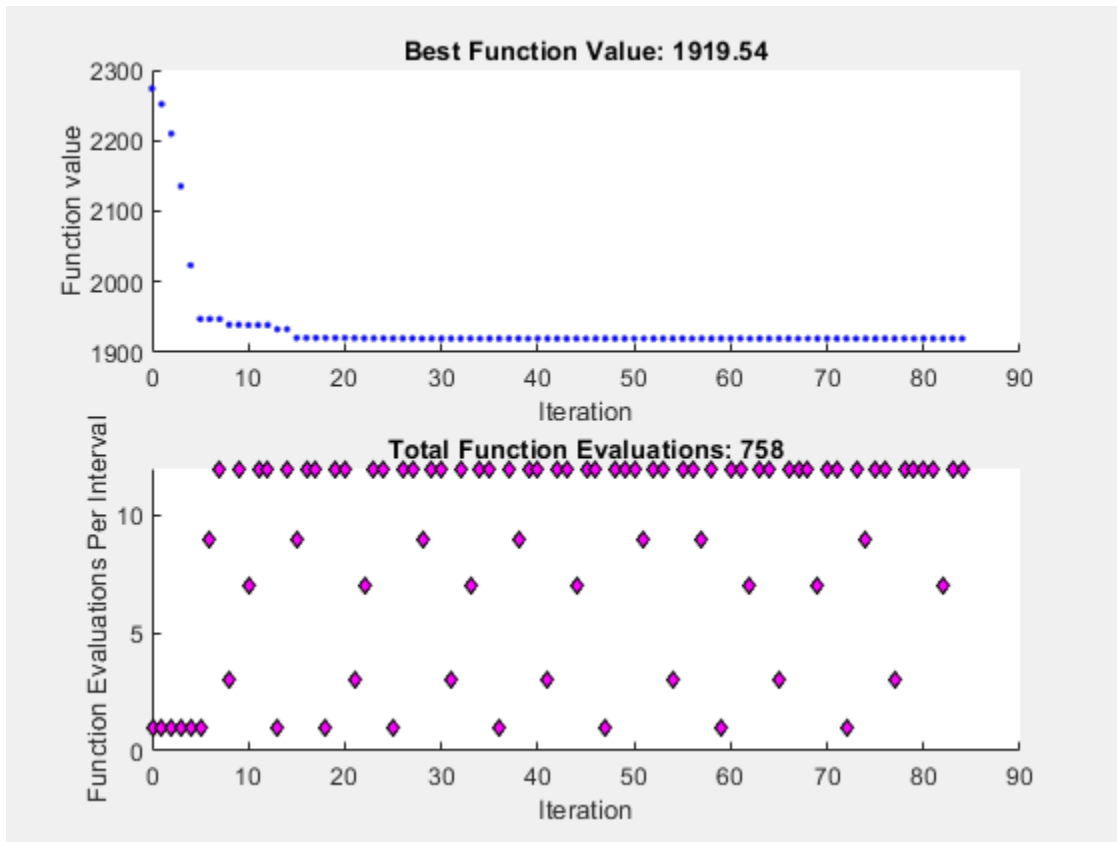
Search Using a Poll Method

The following example illustrates the use of a search method on the problem described in “Constrained Minimization Using `patternsearch` and Optimize Live Editor Task” on page 6-71. In this case, the search method is the GSS Positive Basis 2N poll. For comparison, first run the problem without a search method.

```
x = optimvar("x",1,6);
prob = optimproblem("Objective",lincontest7(x));
x0.x = [2 1 0 9 1 0];
Aineq = [-8 7 3 -4 9 0];
bineq = 7;
Aeq = [7 1 8 3 3 3; 5 0 -5 1 -5 8; -2 -6 7 1 1 9; 1 -1 2 -2 3 -3];
beq = [84 62 65 1];
prob.Constraints.Aineq = Aineq*x' <= bineq;
prob.Constraints.Aeq = Aeq*x' == beq';
options = optimoptions('patternsearch',...
    'PlotFcn',{@psplotbestf,@psplotfunccount});
[x,fval,exitflag,output] = solve(prob,x0,...
    "Options",options,"Solver","patternsearch");
```

Solving problem using `patternsearch`.

Optimization terminated: mesh size less than `options.MeshTolerance`.

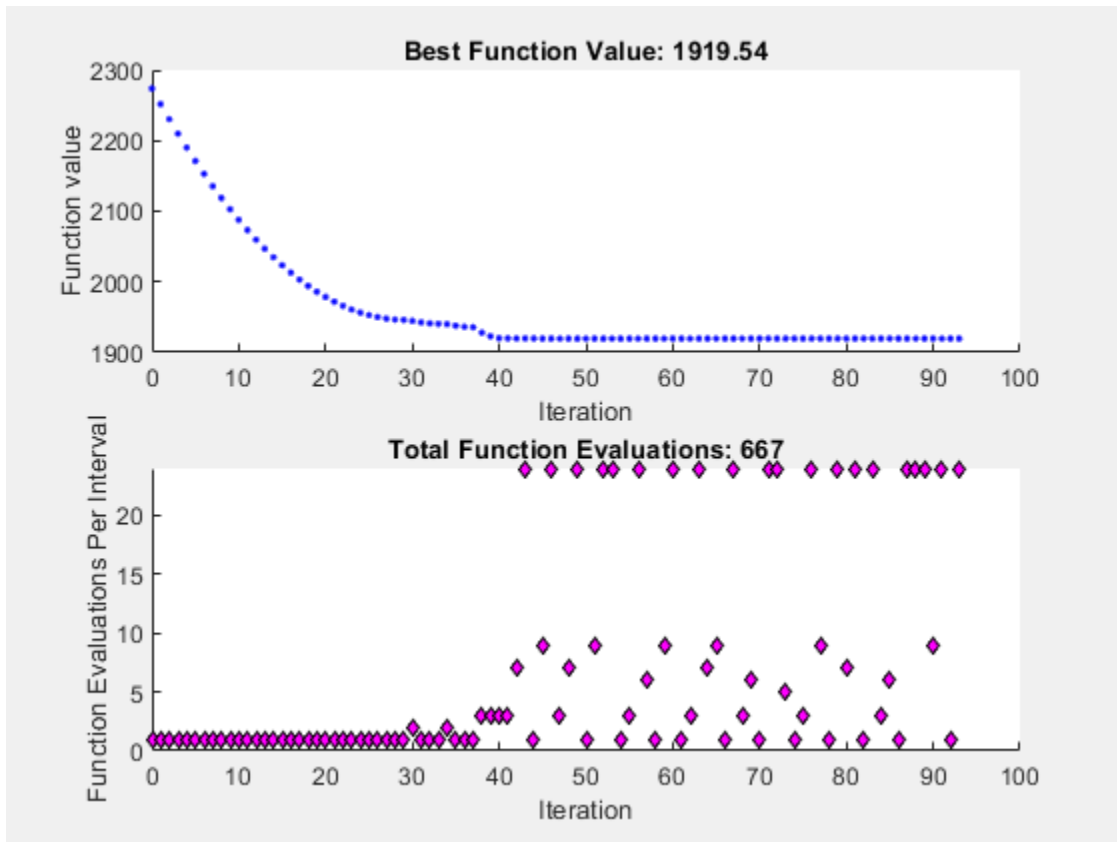


To use the GSS Positive Basis 2N poll as a search method, change the SearchFcn option.

```
rng default % For reproducibility
options.SearchFcn = @GSSPositiveBasis2N;
[x2,fval2,exitflag2,output2] = solve(prob,x0,...
    "Options",options,"Solver","patternsearch");
```

Solving problem using patternsearch.

Optimization terminated: mesh size less than options.MeshTolerance.



Both optimizations reached the same objective function value. Using the search method reduces the number of function evaluations, though not the number of iterations.

```
table([output.funccount;output2.funccount],[output.iterations;output2.iterations],...
      'VariableNames',["Function Evaluations" "Iterations"],...
      'RowNames',["Without Search" "With Search"])
```

ans=2×2 table

	Function Evaluations	Iterations
Without Search	758	84
With Search	667	93

Search Using a Different Solver

patternsearch takes a long time to minimize Rosenbrock's function. The function is

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2.$$

Rosenbrock's function is described and plotted in "Constrained Nonlinear Problem Using Optimize Live Editor Task or Solver". The minimum of Rosenbrock's function is 0, attained at the point [1, 1]. Because patternsearch is not efficient at minimizing this function, use a different search method to help.

Create the objective function.

```
dejong2fcn = @(x)100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
```

The default maximum number of iterations for `patternsearch` with two variables is 200, and the default maximum number of function evaluations is 4000. Increase these values to `MaxFunctionEvaluations = 5000` and `MaxIterations = 2000`.

```
opts = optimoptions("patternsearch","MaxFunctionEvaluations",5000,"MaxIterations",2000);
```

Run `patternsearch` starting from `[-1.9 2]`.

```
x = optimvar("x",1,2);
prob = optimproblem("Objective",dejong2fcn(x));
x0.x = [-1.9,2];
[sol,feval,eflag,output] = solve(prob,x0,...
    "Options",opts,"Solver","patternsearch");
```

Solving problem using `patternsearch`.

Maximum number of function evaluations exceeded: increase `options.MaxFunctionEvaluations`.

```
disp(feval)
```

```
0.8560
```

```
disp(output.funccount)
```

```
5000
```

The optimization did not complete even after 5000 function evaluations, and the result is not very close to the optimal value of 0.

Set the options to use `fminsearch` as the search method, using the default number of function evaluations and iterations.

```
opts = optimoptions("patternsearch","SearchFcn",@searchneldermead);
```

Rerun the optimization.

```
[sol2,feval2,eflag2,output2] = solve(prob,x0,...
    "Options",opts,"Solver","patternsearch");
```

Solving problem using `patternsearch`.

Optimization terminated: mesh size less than `options.MeshTolerance`.

```
disp(feval2)
```

```
4.0686e-10
```

```
disp(output2.funccount)
```

```
291
```

The objective function value at the solution is much better (lower) when using this search method, and the number of function evaluations is much lower. `fminsearch` is more efficient at getting close to the minimum of Rosenbrock's function.

See Also

`patternsearch` | `solve`

Related Examples

- “Direct Search”
- “Search and Poll” on page 6-42

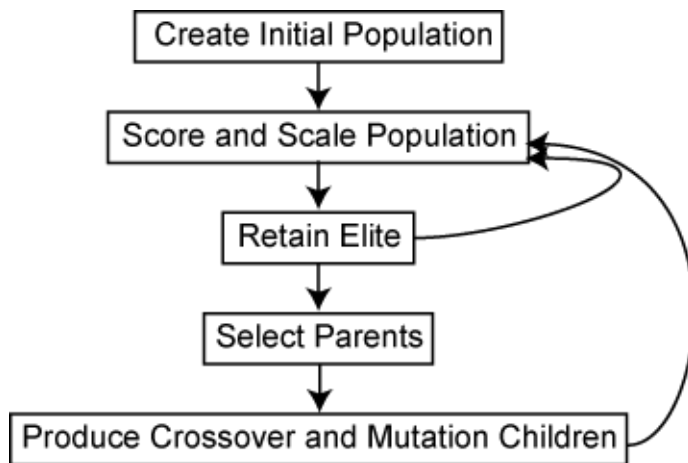
Using the Genetic Algorithm

- “What Is the Genetic Algorithm?” on page 8-2
- “Minimize Rastrigin's Function” on page 8-4
- “Genetic Algorithm Terminology” on page 8-13
- “How the Genetic Algorithm Works” on page 8-15
- “Coding and Minimizing a Fitness Function Using the Genetic Algorithm” on page 8-22
- “Constrained Minimization Using the Genetic Algorithm” on page 8-27
- “Effects of Genetic Algorithm Options” on page 8-32
- “Mixed Integer ga Optimization” on page 8-40
- “Solve a Mixed-Integer Engineering Design Problem Using the Genetic Algorithm” on page 8-47
- “Nonlinear Constraint Solver Algorithms for Genetic Algorithm” on page 8-56
- “Create Custom Plot Function” on page 8-59
- “Resume ga” on page 8-62
- “Options and Outputs” on page 8-64
- “Reproduce Results” on page 8-67
- “Run ga from a File” on page 8-69
- “Population Diversity” on page 8-72
- “Fitness Scaling” on page 8-80
- “Vary Mutation and Crossover” on page 8-83
- “Global vs. Local Optimization Using ga” on page 8-91
- “Hybrid Scheme in the Genetic Algorithm” on page 8-95
- “Set Maximum Number of Generations and Stall Generations” on page 8-101
- “Vectorize the Fitness Function” on page 8-103
- “Custom Output Function for Genetic Algorithm” on page 8-105
- “Custom Data Type Optimization Using the Genetic Algorithm” on page 8-109
- “When to Use a Hybrid Function” on page 8-116

What Is the Genetic Algorithm?

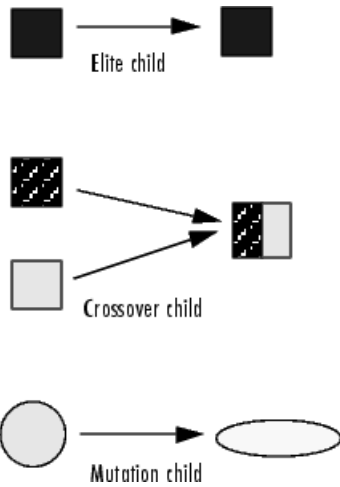
The genetic algorithm is a method for solving both constrained and unconstrained optimization problems that is based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions. At each step, the genetic algorithm selects individuals from the current population to be parents and uses them to produce the children for the next generation. Over successive generations, the population "evolves" toward an optimal solution. You can apply the genetic algorithm to solve a variety of optimization problems that are not well suited for standard optimization algorithms, including problems in which the objective function is discontinuous, nondifferentiable, stochastic, or highly nonlinear. The genetic algorithm can address problems of mixed integer programming, where some components are restricted to be integer-valued.

This flow chart outlines the main algorithmic steps. For details, see "How the Genetic Algorithm Works" on page 8-15.



The genetic algorithm uses three main types of rules at each step to create the next generation from the current population:

- *Selection rules* select the individuals, called *parents*, that contribute to the population at the next generation. The selection is generally stochastic, and can depend on the individuals' scores.
- *Crossover rules* combine two parents to form children for the next generation.
- *Mutation rules* apply random changes to individual parents to form children.



The genetic algorithm differs from a classical, derivative-based, optimization algorithm in two main ways, as summarized in the following table:

Classical Algorithm	Genetic Algorithm
Generates a single point at each iteration. The sequence of points approaches an optimal solution.	Generates a population of points at each iteration. The best point in the population approaches an optimal solution.
Selects the next point in the sequence by a deterministic computation.	Selects the next population by computation which uses random number generators.
Typically converges quickly to a local solution.	Typically takes many function evaluations to converge. May or may not converge to a local or global minimum.

See Also

More About

- “Genetic Algorithm Terminology” on page 8-13
- “How the Genetic Algorithm Works” on page 8-15
- “Nonlinear Constraint Solver Algorithms for Genetic Algorithm” on page 8-56
- “Local vs. Global Optima” on page 1-24

Minimize Rastrigin's Function

Rastrigin's Function

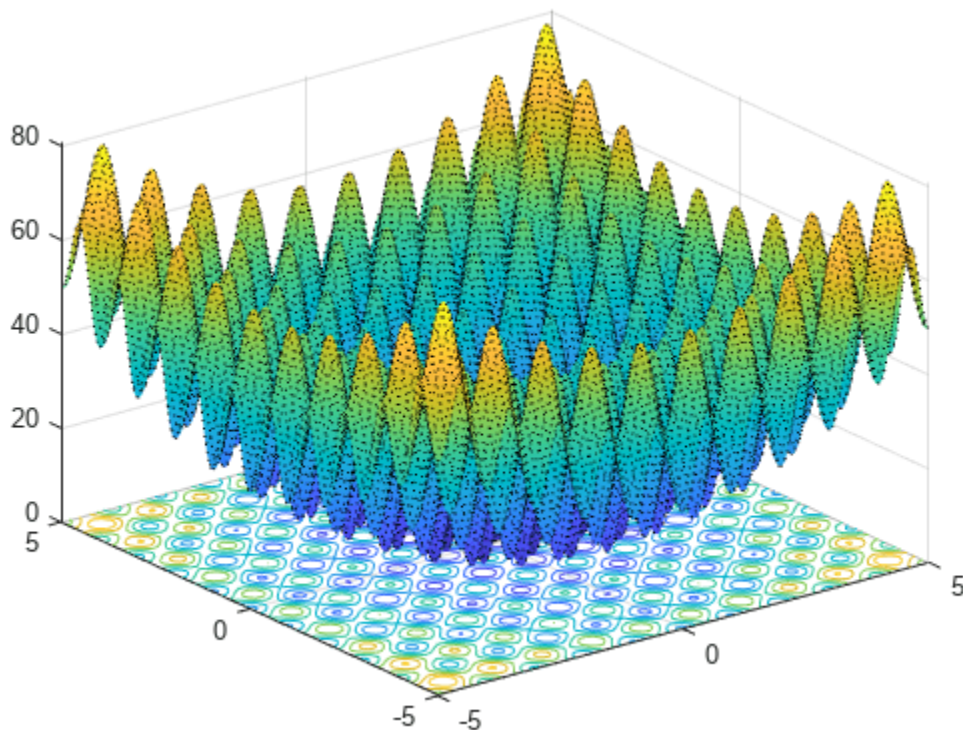
This example shows how to find the minimum of Rastrigin's function, a function that is often used to test the genetic algorithm. The example presents two approaches for minimizing: using the Optimize Live Editor task and working at the command line.

For two independent variables, Rastrigin's function is defined as

$$\text{Ras}(x) = 20 + x_1^2 + x_2^2 - 10(\cos 2\pi x_1 + \cos 2\pi x_2).$$

The `rastriginsfcn.m` file, which computes the values of Rastrigin's function, is available when you run this example. Create a surface plot of Rastrigin's function.

```
fsurf(@(x,y)reshape(rastriginsfcn([x(:),y(:)]),size(x)),...
    "MeshDensity",100,...
    "ShowContours","on",...
    "LineStyle",":")
```



As the plot shows, Rastrigin's function has many local minima—the “valleys” in the plot. However, the function has just one global minimum, which occurs at the point $[0\ 0]$ in the x - y plane, where the value of the function is 0. At any local minimum other than $[0\ 0]$, the value of Rastrigin's function is greater than 0. The farther the local minimum is from the origin, the larger the value of the function is at that point.

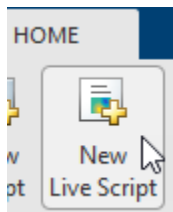
Rastrigin's function is often used to test the genetic algorithm, because its many local minima make it difficult for standard, gradient-based methods to find the global minimum.

Minimize Using the Optimize Live Editor Task

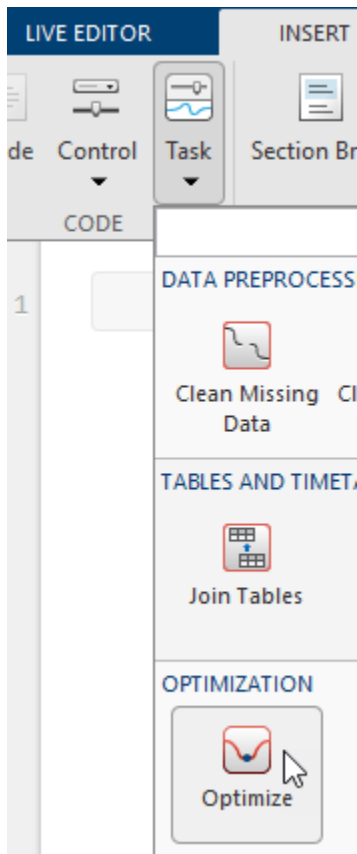
This section explains how to find the minimum of Rastrigin's function using the genetic algorithm.

Note: Because the genetic algorithm uses random number generators, the algorithm returns different results each time you run it.

- Create a new live script by clicking the **New Live Script** button in the **File** section on the **Home** tab.



- Insert an Optimize Live Editor task. Click the **Insert** tab and then, in the **Code** section, select **Task > Optimize**.




Optimize ● ? :

Solve an optimization problem or system of equations


Select approach

Problem-based (recommended)



- Easier to define problem
- Represents problem inputs symbolically
- Built-in automatic differentiation

Solver-based



- Start with a solver
- Represents inputs as matrices/functions
- Allows specialized solution methods

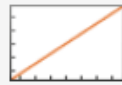
- Click the **Solver-based** button.

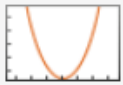
Optimize ○ ? :

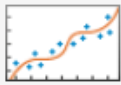
Minimize a function with or without constraints

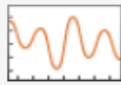
▼ Specify problem type


Objective


Linear


Quadratic



Least squares



Nonlinear



Nonsmooth


Select an objective type to see example functions


Constraints


 Unconstrained


 Lower bounds


 Upper bounds

 Linear inequality

 Linear equality

 Second-order cone

 Nonlinear

 Integer

Select constraint types to see example formulas

Solver ?

▼ Select problem data

Objective function ▼ ?

Initial point (x0) ▼

► Specify solver options

▼ Display progress

Text display ▼

Plot

Current point

Evaluation count

Objective value and feasibility

Objective value

Max constraint violation

Step size

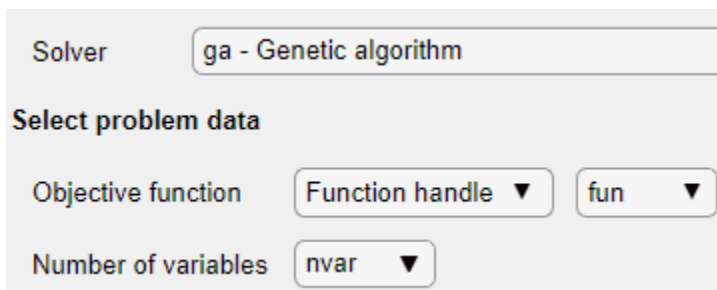
Optimality measure

- For use in entering problem data, insert a new section by clicking the **Section Break** button on the **Insert** tab. New sections appear above and below the task.

- In the new section above the task, enter the following code to define the number of variables and objective function.

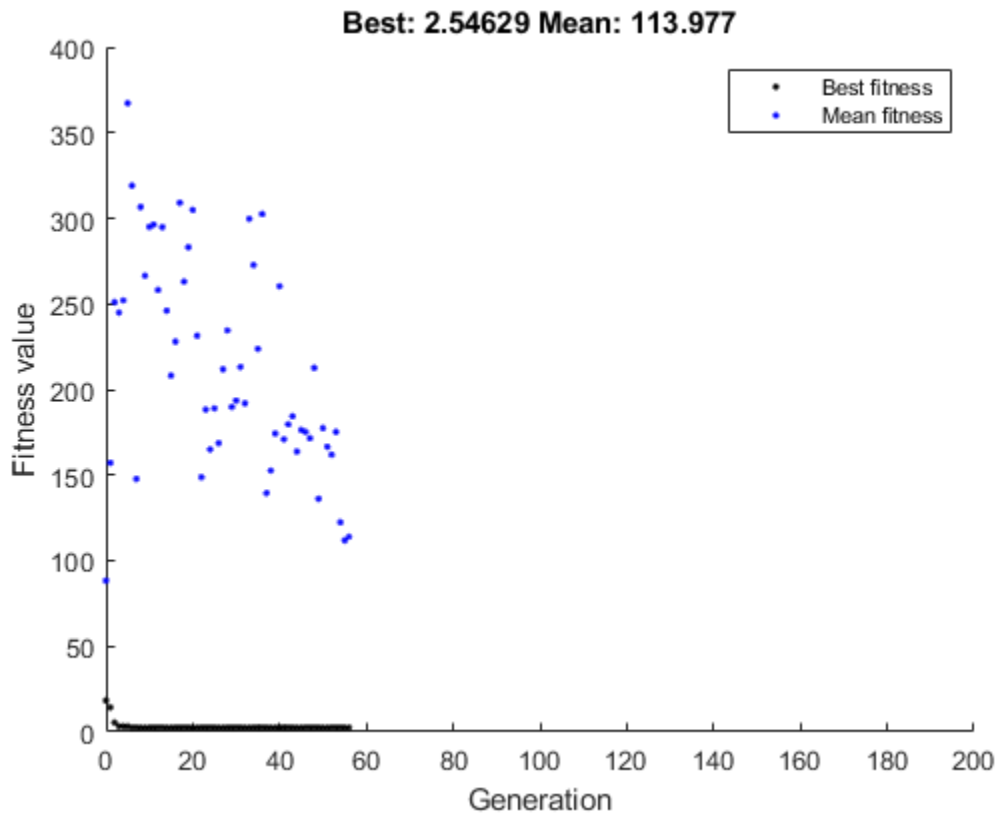
```
rng default % For reproducibility
nvar = 2;
fun = @rastriginsfcn;
```

- To place these variables into the workspace, run the section by pressing **Ctrl+Enter**.
- In the **Specify problem type** section of the task, click the **Objective > Nonlinear** button.
- Select **Solver > ga - Genetic algorithm**.
- In the **Select problem data** section of the task, select **Objective function > Function handle** and then choose `fun`.
- Select **Number of variables > nvar**.



The screenshot shows a configuration panel for a task. At the top, the 'Solver' is set to 'ga - Genetic algorithm'. Below this, the 'Select problem data' section contains three dropdown menus: 'Objective function' is set to 'Function handle', 'Number of variables' is set to 'nvar', and there is an additional dropdown menu set to 'fun'.

- In the **Display progress** section of the task, select the **Best fitness** plot.
- To run the solver, click the options button `:` at the top right of the task window, and select **Run Section**. The plot appears in a separate figure window and in the task output area. Note that your plot might be different from the one shown, because `ga` is a stochastic algorithm.



- The points at the bottom of the plot denote the best fitness values, while the points above them denote the averages of the fitness values in each generation. The top of the plot displays the best and mean values, numerically, in the current generation.
- To see the solution and fitness function value, look at the top of the task.

Optimize

```
solution, objectiveValue = Minimize fun using ga solver
```

- To view the values of these variables, enter the following code in the section below the task.

```
disp(solution)
disp(objectiveValue)
```

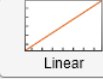
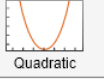
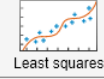
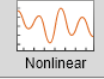
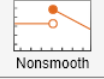
The value shown is not very close to the actual minimum value of Rastrigin's function, which is 0. The topics "Set Initial Range" on page 8-72, "Vary Mutation and Crossover" on page 8-83, and "Set Maximum Number of Generations and Stall Generations" on page 8-101 describe ways to achieve a result that is closer to the actual minimum. Or, you can simply rerun the solver to try to obtain a better result.

The final state of the Live Editor task appears here.

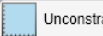



Optimize ○ ? ⋮

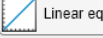
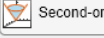


`solution`, `objectiveValue` = Minimize `fun` using `ga` solver

▼ Specify problem type

Objective  Linear  Quadratic  Least squares  Nonlinear  Nonsmooth

Examples: $f(x, y) = x/y$, $f(x) = \cos(x)$, $f(x) = \log(x)$, $f(x) = e^x$, $f(x) = x^3$, Solve $F(x) = 0, \dots$

Constraints  Unconstrained  Lower bounds  Upper bounds  Linear inequality

 Linear equality  Second-order cone  Nonlinear  Integer

Select constraint types to see example formulas

Solver ga - Genetic algorithm ?

▼ Select problem data

Objective function Function handle fun ?

Number of variables nvar

► Specify solver options

▼ Display progress

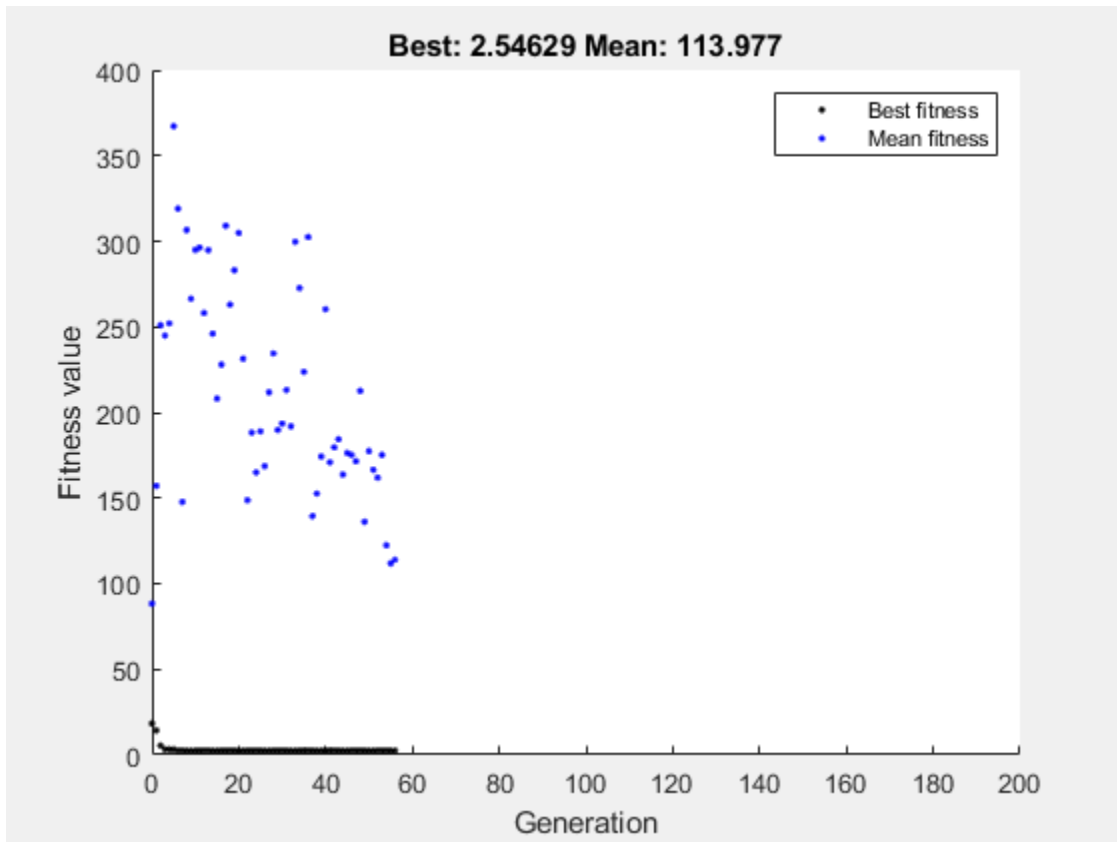
Text display Final output

Plot Distance Genealogy Selection Score diversity

Scores Stopping criteria Max constraint violation Best fitness

Best individual Expectation value Range

Optimization terminated: average change in the fitness value less than options.FunctionTolerance



```
disp(solution)
```

```
0.9785    0.9443
```

```
disp(objectiveValue)
```

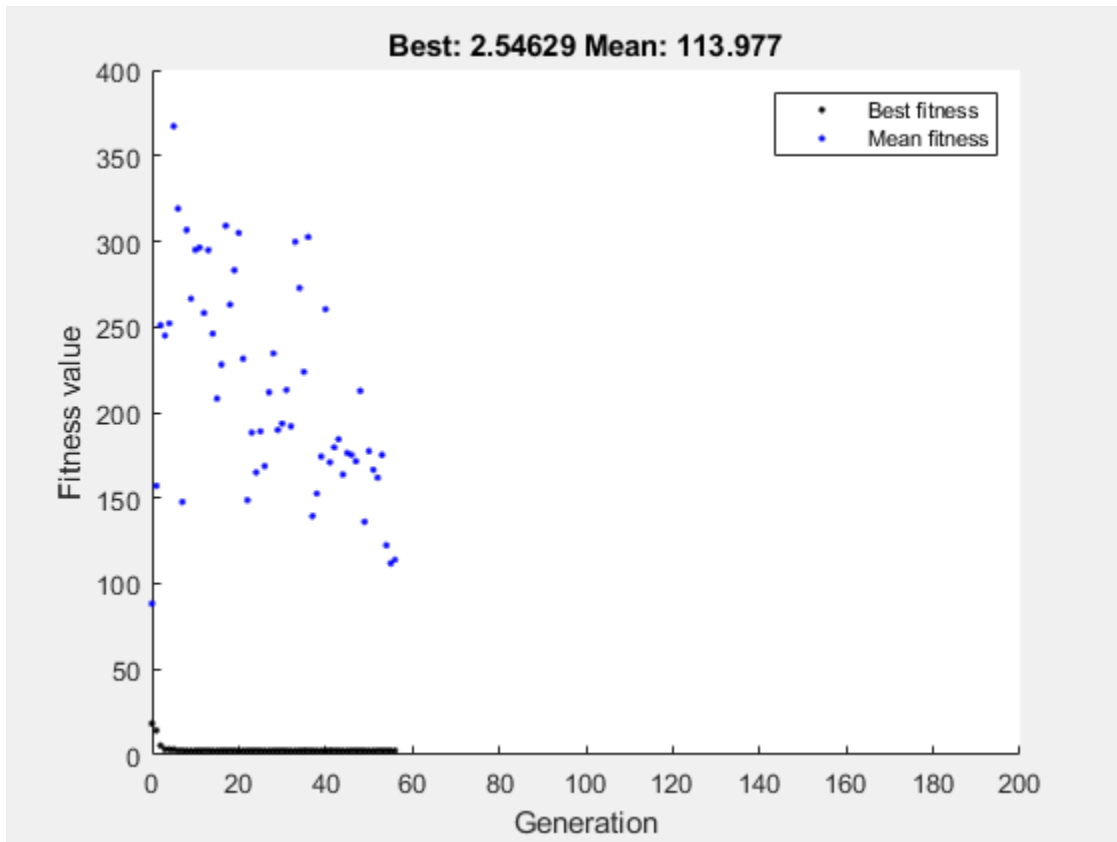
```
2.5463
```

Minimize at the Command Line

To find the minimum of Rastrigin's function at the command line, enter the following code.

```
rng default % For reproducibility
options = optimoptions('ga','PlotFcn','gaplotbestf');
[solution,objectiveValue] = ga(@rastriginsfcn,2,...
    [],[],[],[],[],[],[],[],options)
```

Optimization terminated: average change in the fitness value less than options.FunctionTolerance



```
solution = 1x2
```

```
    0.9785    0.9443
```

```
objectiveValue = 2.5463
```

The points at the bottom of the plot denote the best fitness values, while the points above them denote the averages of the fitness values in each generation. The top of the plot displays the best and mean values, numerically, in the current generation.

Both the `Optimize Live Editor` task and the command line allow you to formulate and solve problems, and they give identical results. The command line is more streamlined, but provides less help for choosing a solver, setting up the problem, and choosing options such as plot functions. You can also start a problem using `Optimize`, and then generate code for command line use, as in “Constrained Nonlinear Problem Using `Optimize Live Editor Task` or `Solver`”.

See Also

More About

- “Constrained Minimization Using the Genetic Algorithm” on page 8-27
- “Effects of Genetic Algorithm Options” on page 8-32
- “Constrained Minimization Using the Genetic Algorithm” on page 8-27

- “Add Interactive Tasks to a Live Script”

Genetic Algorithm Terminology

In this section...

“Fitness Functions” on page 8-13
 “Individuals” on page 8-13
 “Populations and Generations” on page 8-13
 “Diversity” on page 8-13
 “Fitness Values and Best Fitness Values” on page 8-14
 “Parents and Children” on page 8-14

Fitness Functions

The *fitness function* is the function you want to optimize. For standard optimization algorithms, this is known as the objective function. The toolbox software tries to find the minimum of the fitness function.

Write the fitness function as a file or anonymous function, and pass it as a function handle input argument to the main genetic algorithm function.

Individuals

An *individual* is any point to which you can apply the fitness function. The value of the fitness function for an individual is its score. For example, if the fitness function is

$$f(x_1, x_2, x_3) = (2x_1 + 1)^2 + (3x_2 + 4)^2 + (x_3 - 2)^2,$$

the vector (2, -3, 1), whose length is the number of variables in the problem, is an individual. The score of the individual (2, -3, 1) is $f(2, -3, 1) = 51$.

An individual is sometimes referred to as a *genome* and the vector entries of an individual as *genes*.

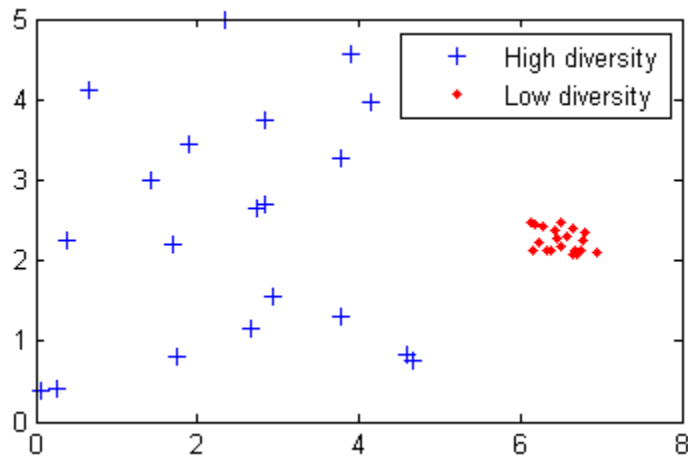
Populations and Generations

A *population* is an array of individuals. For example, if the size of the population is 100 and the number of variables in the fitness function is 3, you represent the population by a 100-by-3 matrix. The same individual can appear more than once in the population. For example, the individual (2, -3, 1) can appear in more than one row of the array.

At each iteration, the genetic algorithm performs a series of computations on the current population to produce a new population. Each successive population is called a new *generation*.

Diversity

Diversity refers to the average distance between individuals in a population. A population has high diversity if the average distance is large; otherwise it has low diversity. In the following figure, the population on the left has high diversity, while the population on the right has low diversity.



Diversity is essential to the genetic algorithm because it enables the algorithm to search a larger region of the space.

Fitness Values and Best Fitness Values

The *fitness value* of an individual is the value of the fitness function for that individual. Because the toolbox software finds the minimum of the fitness function, the *best* fitness value for a population is the smallest fitness value for any individual in the population.

Parents and Children

To create the next generation, the genetic algorithm selects certain individuals in the current population, called *parents*, and uses them to create individuals in the next generation, called *children*. Typically, the algorithm is more likely to select parents that have better fitness values.

See Also

More About

- “What Is the Genetic Algorithm?” on page 8-2
- “How the Genetic Algorithm Works” on page 8-15
- “Nonlinear Constraint Solver Algorithms for Genetic Algorithm” on page 8-56

How the Genetic Algorithm Works

In this section...

“Outline of the Algorithm” on page 8-15
 “Initial Population” on page 8-15
 “Creating the Next Generation” on page 8-16
 “Plots of Later Generations” on page 8-18
 “Stopping Conditions for the Algorithm” on page 8-18
 “Selection” on page 8-19
 “Reproduction Options” on page 8-19
 “Mutation and Crossover” on page 8-20
 “Integer and Linear Constraints” on page 8-20

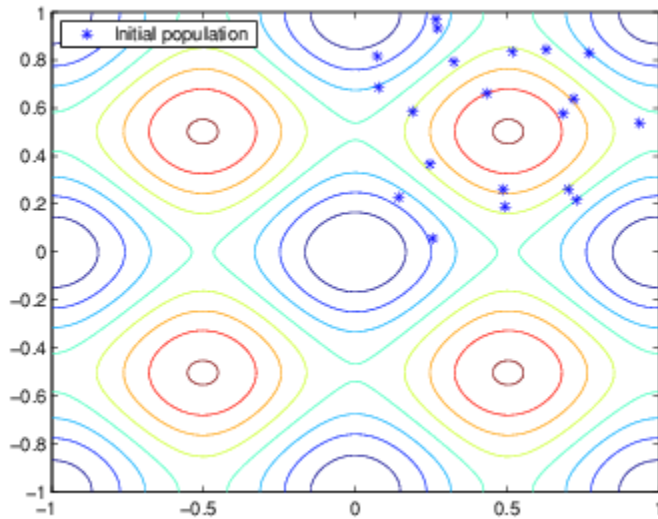
Outline of the Algorithm

The following outline summarizes how the genetic algorithm works:

- 1 The algorithm begins by creating a random initial population.
- 2 The algorithm then creates a sequence of new populations. At each step, the algorithm uses the individuals in the current generation to create the next population. To create the new population, the algorithm performs the following steps:
 - a Scores each member of the current population by computing its fitness value. These values are called the raw fitness scores.
 - b Scales the raw fitness scores to convert them into a more usable range of values. These scaled values are called expectation values.
 - c Selects members, called parents, based on their expectation.
 - d Some of the individuals in the current population that have lower fitness are chosen as *elite*. These elite individuals are passed to the next population.
 - e Produces children from the parents. Children are produced either by making random changes to a single parent—*mutation*—or by combining the vector entries of a pair of parents—*crossover*.
 - f Replaces the current population with the children to form the next generation.
- 3 The algorithm stops when one of the stopping criteria is met. See “Stopping Conditions for the Algorithm” on page 8-18.
- 4 The algorithm takes modified steps for linear and integer constraints. See “Integer and Linear Constraints” on page 8-20.
- 5 The algorithm is further modified for nonlinear constraints. See “Nonlinear Constraint Solver Algorithms for Genetic Algorithm” on page 8-56.

Initial Population

The algorithm begins by creating a random initial population, as shown in the following figure.



In this example, the initial population contains 20 individuals. Note that all the individuals in the initial population lie in the upper-right quadrant of the picture, that is, their coordinates lie between 0 and 1. For this example, the `InitialPopulationRange` option is `[0;1]`.

If you know approximately where the minimal point for a function lies, you should set `InitialPopulationRange` so that the point lies near the middle of that range. For example, if you believe that the minimal point for Rastrigin's function is near the point `[0 0]`, you could set `InitialPopulationRange` to be `[-1;1]`. However, as this example shows, the genetic algorithm can find the minimum even with a less than optimal choice for `InitialPopulationRange`.

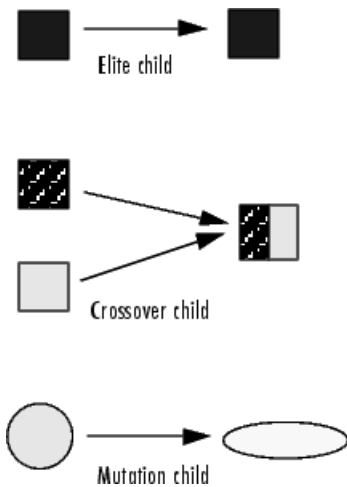
Creating the Next Generation

At each step, the genetic algorithm uses the current population to create the children that make up the next generation. The algorithm selects a group of individuals in the current population, called *parents*, who contribute their *genes*—the entries of their vectors—to their children. The algorithm usually selects individuals that have better fitness values as parents. You can specify the function that the algorithm uses to select the parents in the `SelectionFcn` option. See “Selection Options” on page 17-30.

The genetic algorithm creates three types of children for the next generation:

- *Elite children* are the individuals in the current generation with the best fitness values. These individuals automatically survive to the next generation.
- *Crossover children* are created by combining the vectors of a pair of parents.
- *Mutation children* are created by introducing random changes, or mutations, to a single parent.

The following schematic diagram illustrates the three types of children.



“Mutation and Crossover” on page 8-20 explains how to specify the number of children of each type that the algorithm generates and the functions it uses to perform crossover and mutation.

The following sections explain how the algorithm creates crossover and mutation children.

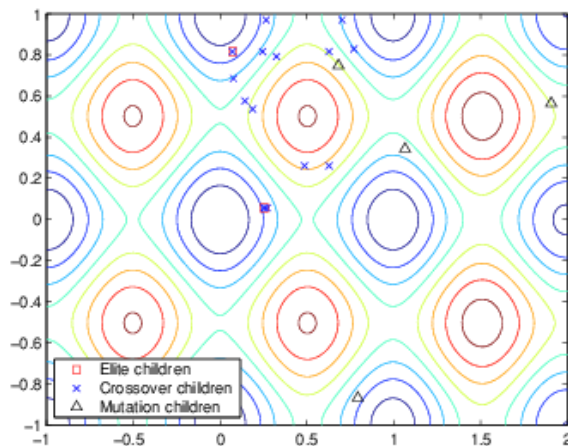
Crossover Children

The algorithm creates crossover children by combining pairs of parents in the current population. At each coordinate of the child vector, the default crossover function randomly selects an entry, or *gene*, at the same coordinate from one of the two parents and assigns it to the child. For problems with linear constraints, the default crossover function creates the child as a random weighted average of the parents.

Mutation Children

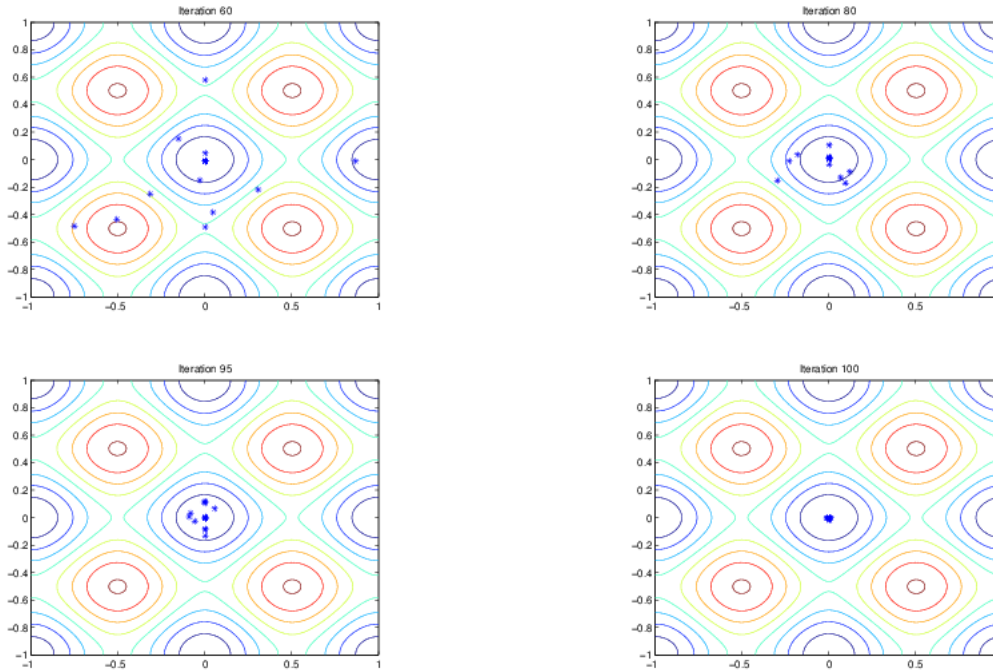
The algorithm creates mutation children by randomly changing the genes of individual parents. By default, for unconstrained problems the algorithm adds a random vector from a Gaussian distribution to the parent. For bounded or linearly constrained problems, the child remains feasible.

The following figure shows the children of the initial population, that is, the population at the second generation, and indicates whether they are elite, crossover, or mutation children.



Plots of Later Generations

The following figure shows the populations at iterations 60, 80, 95, and 100.



As the number of generations increases, the individuals in the population get closer together and approach the minimum point $[0 \ 0]$.

Stopping Conditions for the Algorithm

The genetic algorithm uses the following options to determine when to stop. See the default values for each option by running `opts = optimoptions('ga')`.

- **MaxGenerations** — The algorithm stops when the number of generations reaches `MaxGenerations`.
- **MaxTime** — The algorithm stops after running for an amount of time in seconds equal to `MaxTime`.
- **FitnessLimit** — The algorithm stops when the value of the fitness function for the best point in the current population is less than or equal to `FitnessLimit`.
- **MaxStallGenerations** — The algorithm stops when the average relative change in the fitness function value over `MaxStallGenerations` is less than `FunctionTolerance`.
- **MaxStallTime** — The algorithm stops if there is no improvement in the objective function during an interval of time in seconds equal to `MaxStallTime`.
- **FunctionTolerance** — The algorithm runs until the average relative change in the fitness function value over `MaxStallGenerations` is less than `FunctionTolerance`.
- **ConstraintTolerance** — The `ConstraintTolerance` is not used as stopping criterion. It is used to determine the feasibility with respect to nonlinear constraints. Also, `max(sqrt(eps), ConstraintTolerance)` determines feasibility with respect to linear constraints.

The algorithm stops as soon as any one of these conditions is met.

Selection

The selection function chooses parents for the next generation based on their scaled values from the fitness scaling function. The scaled fitness values are called the expectation values. An individual can be selected more than once as a parent, in which case it contributes its genes to more than one child. The default selection option, `@selectionstochunif`, lays out a line in which each parent corresponds to a section of the line of length proportional to its scaled value. The algorithm moves along the line in steps of equal size. At each step, the algorithm allocates a parent from the section it lands on.

A more deterministic selection option is `@selectionremainder`, which performs two steps:

- In the first step, the function selects parents deterministically according to the integer part of the scaled value for each individual. For example, if an individual's scaled value is 2.3, the function selects that individual twice as a parent.
- In the second step, the selection function selects additional parents using the fractional parts of the scaled values, as in stochastic uniform selection. The function lays out a line in sections, whose lengths are proportional to the fractional part of the scaled value of the individuals, and moves along the line in equal steps to select the parents.

Note that if the fractional parts of the scaled values all equal 0, as can occur using `Top` scaling, the selection is entirely deterministic.

For details and more selection options, see “Selection Options” on page 17-30.

Reproduction Options

Reproduction options control how the genetic algorithm creates the next generation. The options are

- `EliteCount` — The number of individuals with the best fitness values in the current generation that are guaranteed to survive to the next generation. These individuals are called *elite children*.

When `EliteCount` is at least 1, the best fitness value can only decrease from one generation to the next. This is what you want to happen, since the genetic algorithm minimizes the fitness function. Setting `EliteCount` to a high value causes the fittest individuals to dominate the population, which can make the search less effective.

- `CrossoverFraction` — The fraction of individuals in the next generation, other than elite children, that are created by crossover. The topic “Setting the Crossover Fraction” in “Vary Mutation and Crossover” on page 8-83 describes how the value of `CrossoverFraction` affects the performance of the genetic algorithm.

Because elite individuals have already been evaluated, `ga` does not reevaluate the fitness function of elite individuals during reproduction. This behavior assumes that the fitness function of an individual is not random, but is a deterministic function. To change this behavior, use an output function. See `EvalElites` in “The State Structure” on page 17-25.

Mutation and Crossover

The genetic algorithm uses the individuals in the current generation to create the children that make up the next generation. Besides elite children, which correspond to the individuals in the current generation with the best fitness values, the algorithm creates

- Crossover children by selecting vector entries, or genes, from a pair of individuals in the current generation and combines them to form a child
- Mutation children by applying random changes to a single individual in the current generation to create a child

Both processes are essential to the genetic algorithm. Crossover enables the algorithm to extract the best genes from different individuals and recombine them into potentially superior children. Mutation adds to the diversity of a population and thereby increases the likelihood that the algorithm will generate individuals with better fitness values.

See “Creating the Next Generation” on page 8-16 for an example of how the genetic algorithm applies mutation and crossover.

You can specify how many of each type of children the algorithm creates as follows:

- `EliteCount` specifies the number of elite children.
- `CrossoverFraction` specifies the fraction of the population, other than elite children, that are crossover children.

For example, if the `PopulationSize` is 20, the `EliteCount` is 2, and the `CrossoverFraction` is 0.8, the numbers of each type of children in the next generation are as follows:

- There are two elite children.
- There are 18 individuals other than elite children, so the algorithm rounds $0.8 \times 18 = 14.4$ to 14 to get the number of crossover children.
- The remaining four individuals, other than elite children, are mutation children.

Integer and Linear Constraints

When a problem has integer or linear constraints (including bounds), the algorithm modifies the evolution of the population.

- When the problem has both integer and linear constraints, the software modifies all generated individuals to be feasible with respect to those constraints. You can use any creation, mutation, or crossover function, and the entire population remains feasible with respect to integer and linear constraints.
- When the problem has only linear constraints, the software does not modify the individuals to be feasible with respect to those constraints. You must use creation, mutation, and crossover functions that maintain feasibility with respect to linear constraints. Otherwise, the population can become infeasible, and the result can be infeasible. The default operators maintain linear feasibility: `gacreationlinearfeasible` or `gacreationnonlinearfeasible` for creation, `mutationadaptfeasible` for mutation, and `crossoverintermediate` for crossover.

The internal algorithms for integer and linear feasibility are similar to those for `surrogateopt`. When a problem has integer and linear constraints, the algorithm first creates linearly feasible points. Then the algorithm tries to satisfy integer constraints by rounding linearly feasible points to integers

using a heuristic that attempts to keep the points linearly feasible. When this process is unsuccessful in obtaining enough feasible points for constructing a population, the algorithm calls `intlinprog` to try to find more points that are feasible with respect to bounds, linear constraints, and integer constraints.

Later, when mutation or crossover creates new population members, the algorithms ensure that the new members are integer and linear feasible by taking similar steps. Each new member is modified, if necessary, to be as close as possible to its original value, while also satisfying the integer and linear constraints and bounds.

See Also

More About

- “Genetic Algorithm Terminology” on page 8-13
- “Nonlinear Constraint Solver Algorithms for Genetic Algorithm” on page 8-56

Coding and Minimizing a Fitness Function Using the Genetic Algorithm

This example shows how to create and minimize a fitness function for the genetic algorithm solver `ga` using three techniques:

- Basic
- Including additional parameters
- Vectorized for speed

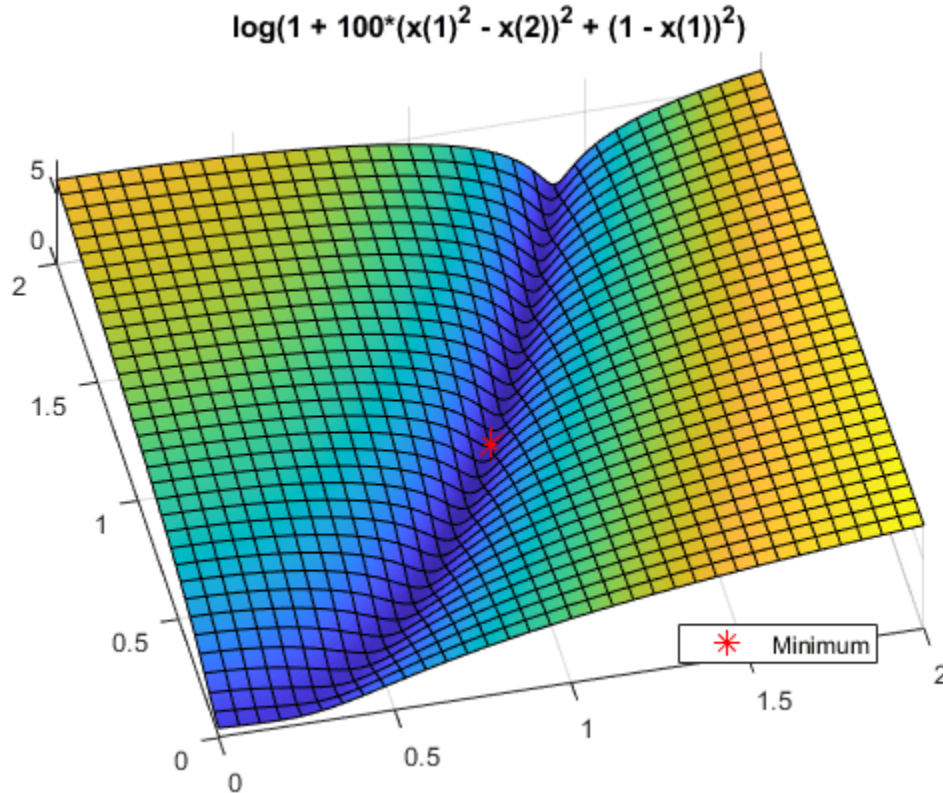
Basic Fitness Function

The basic fitness function is Rosenbrock's function, a common test function for optimizers. The function is a sum of squares:

$$f(x) = 100(x_1^2 - x_2)^2 + (1 - x_1)^2.$$

The function has a minimum value of zero at the point $[1, 1]$. Because the Rosenbrock function is quite steep, plot the logarithm of one plus the function.

```
fsurf(@(x,y)log(1 + 100*(x.^2 - y).^2 + (1 - x).^2),[0,2])
title('log(1 + 100*(x(1)^2 - x(2))^2 + (1 - x(1))^2)')
view(-13,78)
hold on
h1 = plot3(1,1,0.1,'r*','MarkerSize',12);
legend(h1,'Minimum','Location','best');
hold off
```



Fitness Function Code

The `simple_fitness` function file implements Rosenbrock's function.

```
type simple_fitness
function y = simple_fitness(x)
%SIMPLE_FITNESS fitness function for GA

% Copyright 2004 The MathWorks, Inc.

y = 100 * (x(1)^2 - x(2)) ^2 + (1 - x(1))^2;
```

A fitness function must take one input `x` where `x` is a row vector with as many elements as number of variables in the problem. The fitness function computes the value of the function and returns that scalar value in its one return argument `y`.

Minimize Using `ga`

To minimize the fitness function using `ga`, pass a function handle to the fitness function as well as the number of variables in the problem. To have `ga` examine the relevant region, include bounds $-3 \leq x(i) \leq 3$. Pass the bounds as the fifth and sixth arguments after `numberOfVariables`. For `ga` syntax details, see `ga`.

`ga` is a random algorithm. For reproducibility, set the random number stream.

```
rng default % For reproducibility
FitnessFunction = @simple_fitness;
```

```

numberOfVariables = 2;
lb = [-3,-3];
ub = [3,3];
[x,fval] = ga(FitnessFunction,numberOfVariables,[],[],[],[],lb,ub)

```

Optimization terminated: maximum number of generations exceeded.

```

x = 1x2
    1.5083    2.2781

```

```
fval = 0.2594
```

The `x` returned by the solver is the best point in the final population computed by `ga`. The `fval` is the value of the function `simple_fitness` evaluated at the point `x`. `ga` did not find an especially good solution. For ways to improve the solution, see “Effects of Genetic Algorithm Options” on page 8-32.

Fitness Function with Additional Parameters

Sometimes your fitness function has extra parameters that act as constants during the optimization. For example, a generalized Rosenbrock's function can have extra parameters representing the constants 100 and 1:

$$f(x, a, b) = a(x_1^2 - x_2)^2 + (b - x_1)^2.$$

`a` and `b` are parameters to the fitness function that act as constants during the optimization (they are not varied as part of the minimization). The `parameterized_fitness.m` file implements this parameterized fitness function.

```

type parameterized_fitness

function y = parameterized_fitness(x,p1,p2)
%PARAMETERIZED_FITNESS fitness function for GA

% Copyright 2004 The MathWorks, Inc.

y = p1 * (x(1)^2 - x(2)) ^2 + (p2 - x(1))^2;

```

Minimize Using Additional Parameters

Use an anonymous function to capture the values of the additional arguments, namely, the constants `a` and `b`. Create a function handle `FitnessFunction` to an anonymous function that takes one input `x`, and calls `parameterized_fitness` with `x`, `a`, and `b`. The anonymous function contains the values of `a` and `b` that exist when the function handle is created.

```

a = 100;
b = 1; % define constant values
FitnessFunction = @(x) parameterized_fitness(x,a,b);
[x,fval] = ga(FitnessFunction,numberOfVariables,[],[],[],[],lb,ub)

```

Optimization terminated: maximum number of generations exceeded.

```

x = 1x2
    1.3198    1.7434

```



```
fval = 0.1025
```

See “Passing Extra Parameters”.

Vectorized Fitness Function

To gain speed, *vectorize* your fitness function. A vectorized fitness function computes the fitness of a collection of points at once, which generally saves time over evaluating these points individually. To write a vectorized fitness function, have your function accept a matrix, where each matrix row represents one point, and have the fitness function return a column vector of fitness function values.

To change the `parameterized_fitness` function file to a vectorized form:

- Change each variable `x(i)` to `x(:,i)`, meaning the column vector of variables corresponding to `x(i)`.
- Change each vector multiplication `*` to `.*` and each exponentiation `^` to `.^` indicating that the operations are element-wise. There are no vector multiplications in this code, so simply change the exponents.

```
type vectorized_fitness
```

```
function y = vectorized_fitness(x,p1,p2)
%VECTORIZED_FITNESS fitness function for GA

% Copyright 2004-2010 The MathWorks, Inc.

y = p1 * (x(:,1).^2 - x(:,2)).^2 + (p2 - x(:,1)).^2;
```

This vectorized version of the fitness function takes a matrix `x` with an arbitrary number of points, meaning and arbitrary number of rows, and returns a column vector `y` with the same number of rows as `x`.

Tell the solver that the fitness function is vectorized in the 'UseVectorized' option.

```
options = optimoptions(@ga,'UseVectorized',true);
```

Include options as the last argument to `ga`.

```
VFitnessFunction = @(x) vectorized_fitness(x,100,1);
[x,fval] = ga(VFitnessFunction,numberOfVariables,[],[],[],[],lb,ub,[],options)
```

```
Optimization terminated: maximum number of generations exceeded.
```

```
x = 1x2
```

```
    1.6219    2.6334
```

```
fval = 0.3876
```

What is the difference in speed? Time the optimization both with and without vectorization.

```
tic
```

```
[x,fval] = ga(VFitnessFunction,numberOfVariables,[],[],[],[],lb,ub,[],options);
```

```
Optimization terminated: maximum number of generations exceeded.
```

```
v = toc;  
tic  
[x,fval] = ga(FitnessFunction,numberOfVariables,[],[],[],[],lb,ub);
```

Optimization terminated: maximum number of generations exceeded.

```
nv = toc;  
fprintf('Using vectorization took %f seconds. No vectorization took %f seconds.\n',v,nv)
```

Using vectorization took 0.153337 seconds. No vectorization took 0.212880 seconds.

In this case, the improvement by vectorization was not great, because computing the fitness function takes very little time. However, for more time-consuming fitness functions, vectorization can be helpful. See “Vectorize the Fitness Function” on page 8-103.

See Also

More About

- “Passing Extra Parameters”
- “Vectorize the Fitness Function” on page 8-103

Constrained Minimization Using the Genetic Algorithm

This example shows how to minimize an objective function subject to nonlinear inequality constraints and bounds using the Genetic Algorithm.

Constrained Minimization Problem

For this problem, the objective function to minimize is a simple function of a 2-D variable x .

$$\text{simple_objective}(x) = (4 - 2.1*x(1)^2 + x(1)^4/3)*x(1)^2 + x(1)*x(2) + (-4 + 4*x(2)^2)*x(2)^2;$$

This function is known as "cam," as described in L.C.W. Dixon and G.P. Szego [1] on page 8-31.

Additionally, the problem has nonlinear constraints and bounds.

$$\begin{array}{ll} x(1)*x(2) + x(1) - x(2) + 1.5 \leq 0 & \text{(nonlinear constraint)} \\ 10 - x(1)*x(2) \leq 0 & \text{(nonlinear constraint)} \\ 0 \leq x(1) \leq 1 & \text{(bound)} \\ 0 \leq x(2) \leq 13 & \text{(bound)} \end{array}$$

Code the Fitness Function

Create a MATLAB file named `simple_objective.m` containing the following code:

```
type simple_objective

function y = simple_objective(x)
%SIMPLE_OBJECTIVE Objective function for PATTERNSEARCH solver

% Copyright 2004 The MathWorks, Inc.

x1 = x(1);
x2 = x(2);
y = (4-2.1.*x1.^2+x1.^4./3).*x1.^2+x1.*x2+(-4+4.*x2.^2).*x2.^2;
```

Solvers such as `ga` accept a single input x , where x has as many elements as the number of variables in the problem. The objective function computes the scalar value of the objective function and returns it in its single output argument y .

Code the Constraint Function

Create a MATLAB file named `simple_constraint.m` containing the following code:

```
type simple_constraint

function [c, ceq] = simple_constraint(x)
%SIMPLE_CONSTRAINT Nonlinear inequality constraints.

% Copyright 2005-2007 The MathWorks, Inc.

c = [1.5 + x(1)*x(2) + x(1) - x(2);
     -x(1)*x(2) + 10];

% No nonlinear equality constraints:
ceq = [];
```

The constraint function computes the values of all the inequality and equality constraints and returns the vectors `c` and `ceq`, respectively. The value of `c` represents nonlinear inequality constraints that the solver attempts to make less than or equal to zero. The value of `ceq` represents nonlinear equality constraints that the solver attempts to make equal to zero. This example has no nonlinear equality constraints, so `ceq = []`. For details, see “Nonlinear Constraints”.

Minimizing Using `ga`

Specify the objective function as a function handle.

```
ObjectiveFunction = @simple_objective;
```

Specify the problem bounds.

```
lb = [0 0]; % Lower bounds
ub = [1 13]; % Upper bounds
```

Specify the nonlinear constraint function as a function handle.

```
ConstraintFunction = @simple_constraint;
```

Specify the number of problem variables.

```
nvars = 2;
```

Call the solver, requesting the optimal point `x` and the function value at the optimal point `fval`.

```
rng default % For reproducibility
[x,fval] = ga(ObjectiveFunction,nvars,[],[],[],[],lb,ub,ConstraintFunction)
```

Optimization finished: average change in the fitness value less than options.FunctionTolerance and

```
x = 1×2
```

```
    0.8122    12.3103
```

```
fval = 9.1268e+04
```

Add Visualization

To observe the solver's progress, specify options that select two plot functions. The plot function `gaplotbestf` plots the best objective function value at every iteration, and the plot function `gaplotmaxconstr` plots the maximum constraint violation at every iteration. Set these two plot functions in a cell array. Also, display information about the solver's progress in the Command Window by setting the `Display` option to `'iter'`.

```
options = optimoptions("ga","PlotFcn",{@gaplotbestf,@gaplotmaxconstr}, ...
    'Display','iter');
```

Run the solver, including the `options` argument.

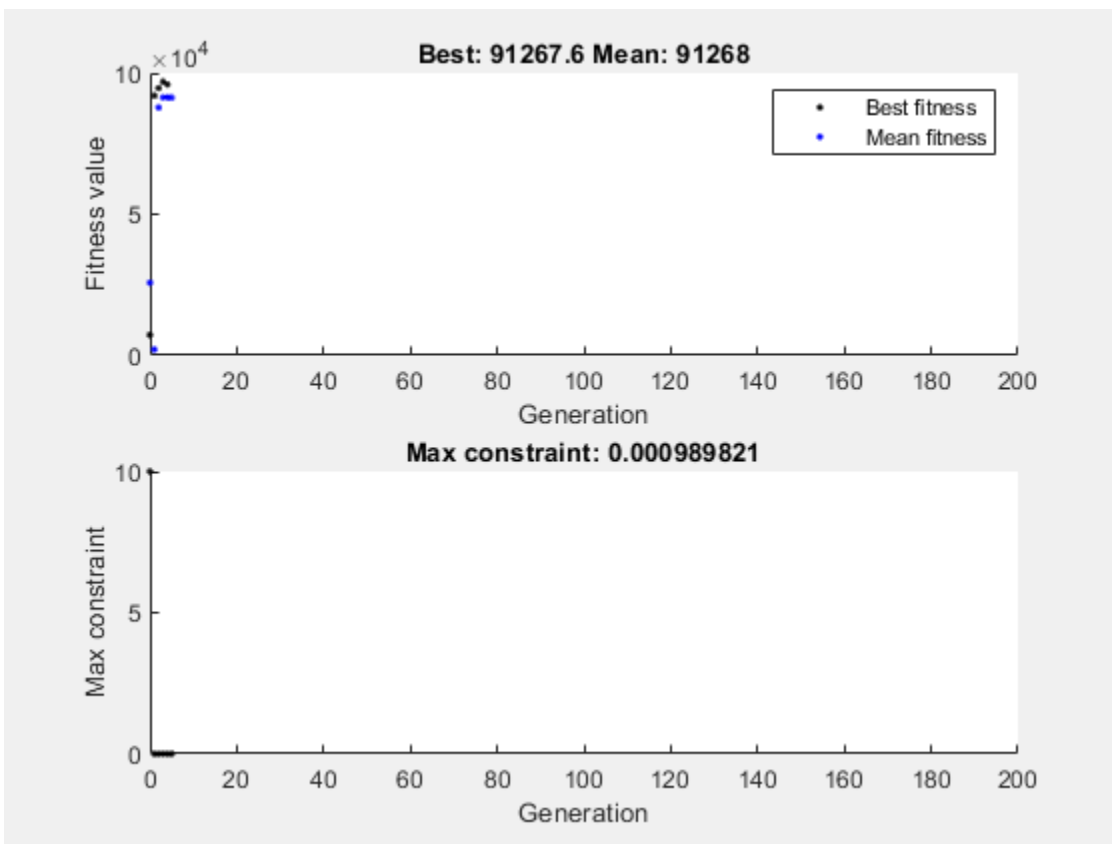
```
[x,fval] = ga(ObjectiveFunction,nvars,[],[],[],[],lb,ub, ...
    ConstraintFunction,options)
```

```
Single objective optimization:
2 Variable(s)
2 Nonlinear inequality constraint(s)
```

```
Options:
CreationFcn: @gacreationuniform
CrossoverFcn: @crossoverscattered
SelectionFcn: @selectionstochunif
MutationFcn: @mutationadaptfeasible
```

Generation	Func-count	Best f(x)	Max Constraint	Stall Generations
1	2524	91986.8	7.786e-09	0
2	4986	94677.4	0	0
3	10362	96929.2	0	0
4	16067	96006.3	0	0
5	23405	91267.6	0.0009898	0

Optimization finished: average change in the fitness value less than options.FunctionTolerance and



x = 1x2

0.8122 12.3103

fval = 9.1268e+04

With iterative display, that ga provides details about the problem type and the creation, crossover, mutation, and selection operators.

Nonlinear constraints cause ga to solve many subproblems at each iteration. As shown in both the plots and the iterative display, the solution process has few iterations. However, the Func-count column in the iterative display shows many function evaluations per iteration.

The `ga` solver handles linear constraints and bounds differently from nonlinear constraints. All the linear constraints and bounds are satisfied throughout the optimization. However, `ga` may not satisfy all the nonlinear constraints at every generation. If `ga` converges to a solution, the nonlinear constraints will be satisfied at that solution.

`ga` uses the mutation and crossover functions to produce new individuals at every generation. The way the `ga` satisfies the linear and bound constraints is to use mutation and crossover functions that only generate feasible points. For example, in the previous call to `ga`, the default mutation function (for unconstrained problems) `mutationgaussian` does not satisfy the linear constraints and so `ga` uses the `mutationadaptfeasible` function instead by default. If you provide a custom mutation function, this custom function must only generate points that are feasible with respect to the linear and bound constraints. All the crossover functions in the toolbox generate points that satisfy the linear constraints and bounds.

However, when your problem contains integer constraints, `ga` enforces that all iterations satisfy bounds and linear constraints. This feasibility occurs for all mutation, crossover, and creation operators, to within a small tolerance.

Provide a Start Point

To speed the solver, you can provide an initial population in the `InitialPopulationMatrix` option. `ga` uses the initial population to start its optimization. Specify a row vector or a matrix where each row represents one start point.

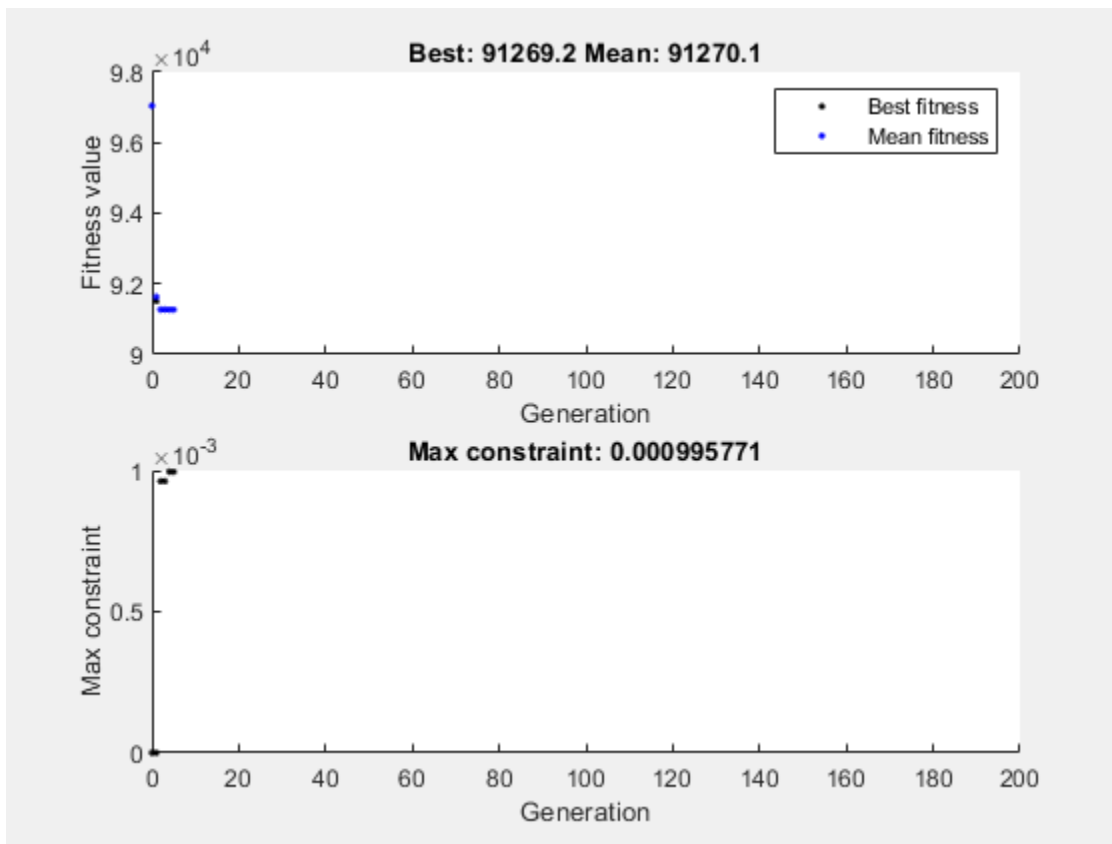
```
X0 = [0.8 12.5]; % Start point (row vector)
options.InitialPopulationMatrix = X0;
[x,fval] = ga(ObjectiveFunction,nvars,[],[],[],[],lb,ub, ...
    ConstraintFunction,options)
```

```
Single objective optimization:
2 Variable(s)
2 Nonlinear inequality constraint(s)
```

```
Options:
CreationFcn: @gacreationuniform
CrossoverFcn: @crossoverscattered
SelectionFcn: @selectionstochunif
MutationFcn: @mutationadaptfeasible
```

Generation	Func-count	Best f(x)	Max Constraint	Stall Generations
1	2500	91507.4	0	0
2	4950	91270.4	0.0009621	0
3	7400	91270.4	0.0009621	1
4	9850	91269.2	0.0009958	0
5	12300	91269.2	0.0009958	1

Optimization finished: average change in the fitness value less than options.FunctionTolerance and



$x = 1 \times 2$

0.8122 12.3104

fval = 9.1269e+04

In this case, providing a start point does not substantially change the solver progress.

References

[1] Dixon, L. C. W., and G .P. Szego (eds.). *Towards Global Optimisation 2*. North-Holland: Elsevier Science Ltd., Amsterdam, 1978.

See Also

More About

- “Write Constraints” on page 2-6

Effects of Genetic Algorithm Options

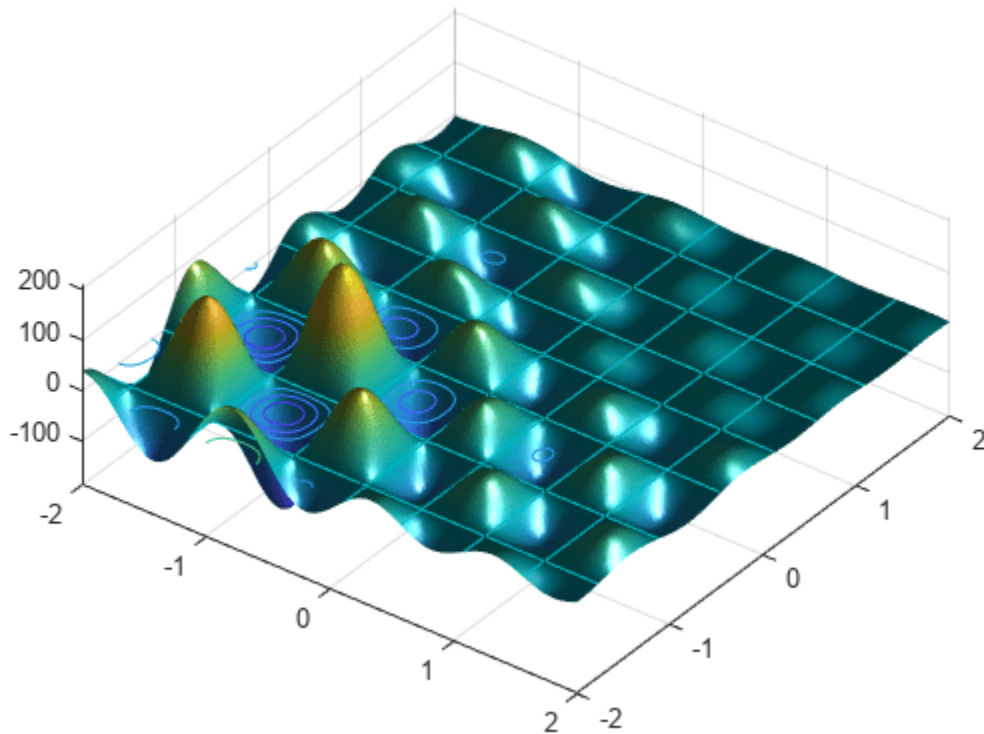
This example shows the effects of some options for the genetic algorithm function `ga`. You create and change options by using the `optimoptions` function.

Set Up a Problem for `ga`

`ga` searches for a minimum of a function using the genetic algorithm. For this example, use `ga` to minimize the fitness function `shufcn`, a real-valued function of two variables.

Plot `shufcn` over the range = `[-2 2; -2 2]` by calling `plotobjective`, which is included when you run this example.

```
plotobjective(@shufcn,[-2 2; -2 2]);
```



To use the `ga` solver, provide at least two input arguments: a fitness function and the number of variables in the problem. The first two output arguments returned by `ga` are `x`, the best point found, and `Fval`, the function value at the best point. A third output argument, `exitFlag`, indicates why `ga` stopped. `ga` can also return a fourth argument, `Output`, which contains information about the performance of the solver.

```
FitnessFunction = @shufcn;  
numberOfVariables = 2;
```

Run the `ga` solver.


```
rng default % For reproducibility
[x,Fval,exitFlag,Output] = ga(FitnessFunction,numberOfVariables);
```

Optimization terminated: average change in the fitness value less than options.FunctionTolerance

```
fprintf('The number of generations is: %d\n', Output.generations);
```

The number of generations is: 124

```
fprintf('The number of function evaluations is: %d\n', Output.funccount);
```

The number of function evaluations is: 5881

```
fprintf('The best function value found is: %g\n', Fval);
```

The best function value found is: -186.199

If you run this example without the `rng default` command, your results can differ, because `ga` is a stochastic algorithm.

How the Genetic Algorithm Works

The genetic algorithm works on a population using a set of operators that are applied to the population. A population is a set of points in the design space. The initial population is generated randomly by default. The algorithm computes the next generation of the population using the fitness of the individuals in the current generation. For details, see “How the Genetic Algorithm Works” on page 8-15.

Add Visualization

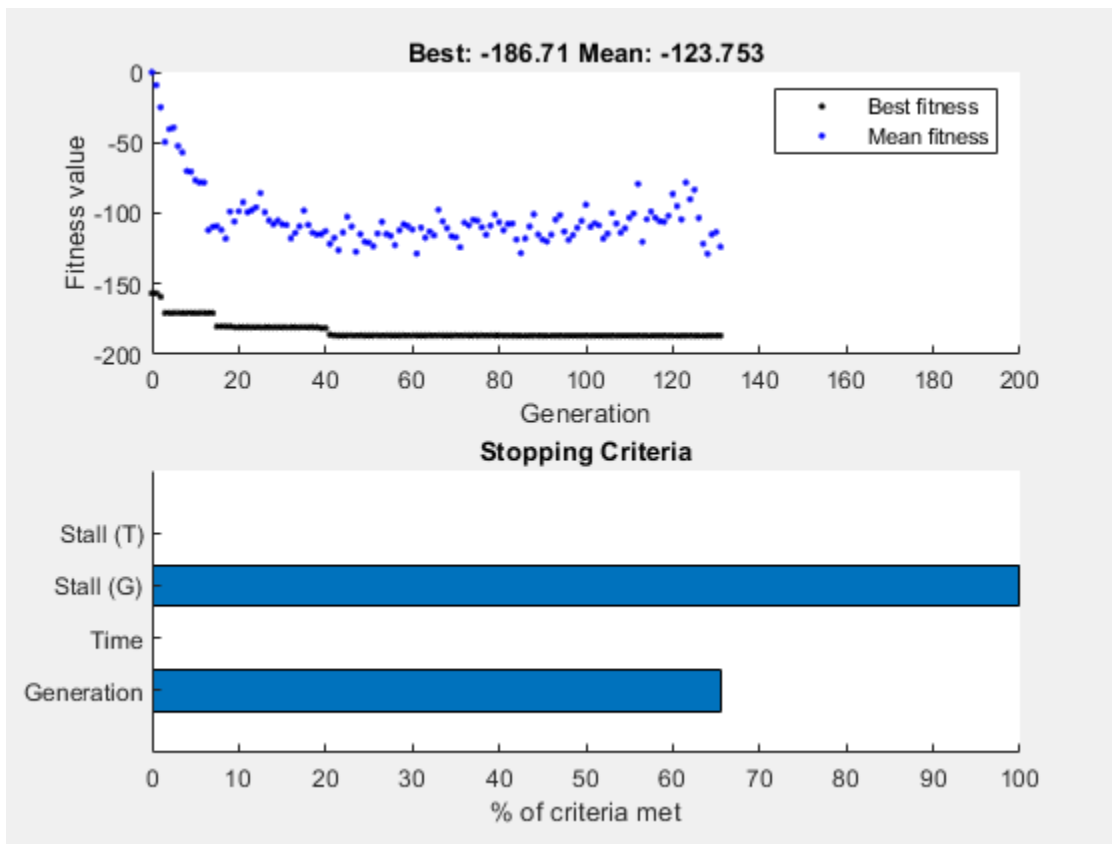
To visualize the solver performance while it is running, set a 'PlotFcn' option using `optimoptions`. In this case, select two plot functions in a cell array. Set `gaplotbestf`, which plots the best and mean score of the population at every generation. Also set `gaplotstopping`, which plots the percentage of stopping criteria satisfied.

```
opts = optimoptions(@ga, 'PlotFcn',{@gaplotbestf,@gaplotstopping});
```

Run the `ga` solver, including the `opts` argument.

```
[x,Fval,exitFlag,Output] = ...
    ga(FitnessFunction,numberOfVariables,[],[],[],[],[],[],[],[],opts);
```

Optimization terminated: average change in the fitness value less than options.FunctionTolerance



Specify Population Options

Population options can have a large effect on solver performance. The speed of each iteration depends on the population size: a larger population leads to slower iterations. Conversely, a larger population leads to `ga` exploring more thoroughly, so can lead to a better solution. Similarly, a wider initial range can lead to more thorough exploration, but can require a larger population to explore the wider range with a similar thoroughness.

Specify Population Size

`ga` creates a default initial population by using a uniform random number generator. The default population size used by `ga` is 50 when the number of decision variables is less than 5, and 200 otherwise. The default size might not work well for some problems; for example, a smaller population size can be sufficient for smaller problems. Since the current problem has only two variables, specify a population size of 10. Set the value of the option `PopulationSize` to 10 in the existing options, `opts`.

```
opts.PopulationSize = 10;
```

Specify Initial Population Range

The default method for generating an initial population uses a uniform random number generator. For problems without integer constraints, `ga` creates an initial population where all the points are in the range -10 to 10. For example, you can generate a population of size three in the default range using this command:

```
Population = [-10, -10] + 20*rand(3,2);
```

You can set the initial range by changing the `InitialPopulationRange` option. The range must be a matrix with two rows. If the range has only one column, that is, it is 2-by-1, then the range of every variable is the given range. For example, if you set the range to `[-1; 1]`, then the initial range for both variables is -1 to 1. To specify a different initial range for each variable, you must specify the range as a matrix with two rows and `numberOfVariables` columns. For example, if you set the range to `[-1 0; 1 2]`, then the first variable has the range -1 to 1, and the second variable has the range 0 to 2 (each column corresponds to a variable).

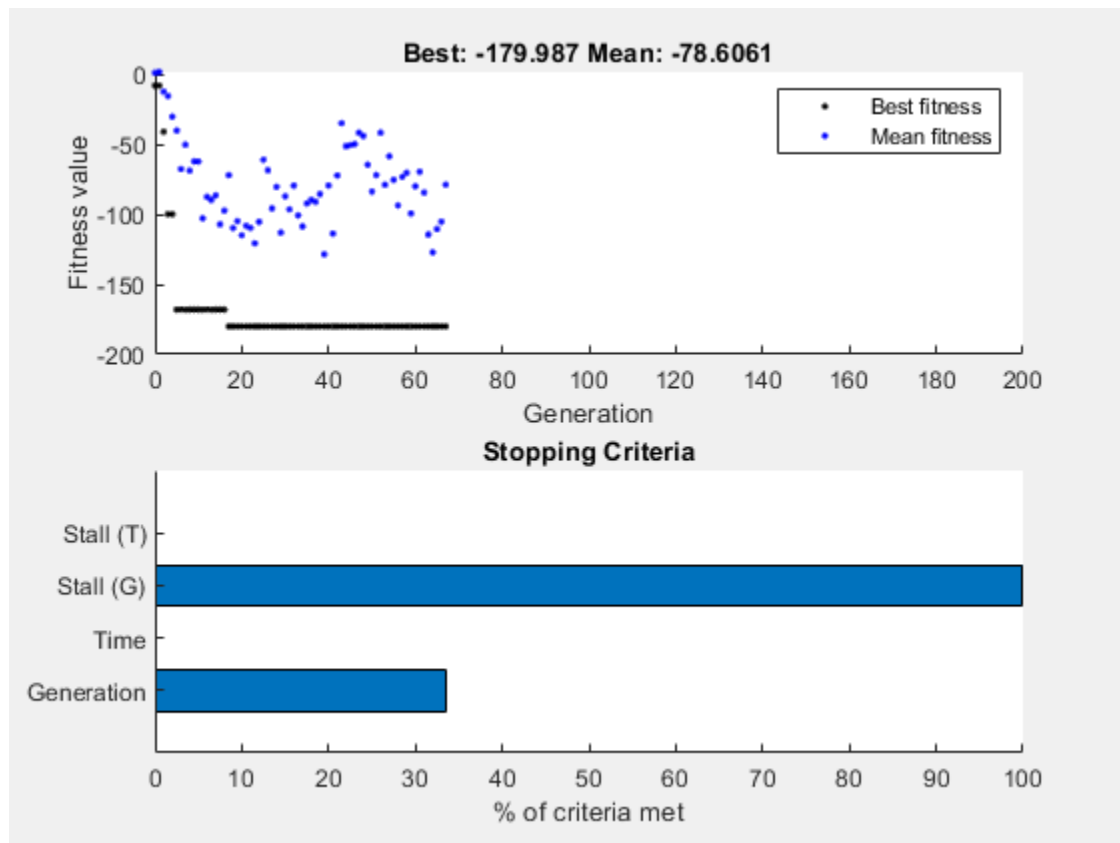
Modify the value of the option `InitialPopulationRange` in the existing options, `opts`.

```
opts.InitialPopulationRange = [-1 0; 1 2];
```

Run the ga solver.

```
[x,Fval,exitFlag,Output] = ga(FitnessFunction,numberOfVariables,[],[],[], ...
    [],[],[],[],opts);
```

Optimization terminated: average change in the fitness value less than `options.FunctionTolerance`



```
fprintf('The number of generations is: %d\n', Output.generations);
```

```
The number of generations is: 67
```

```
fprintf('The number of function evaluations is: %d\n', Output.funccount);
```

```
The number of function evaluations is: 614
```

```
fprintf('The best function value found is: %g\n', Fval);
```

The best function value found is: -179.987

Reproduce Results

By default, `ga` starts with a random initial population created using MATLAB® random number generators. The solver produces the next generation using `ga` operators that also use these same random number generators. Every time a random number is generated, the state of the random number generators changes. So, even if you do not change any options, you can get different results when you run the solver again.

Run the solver twice to show this phenomenon.

Run the `ga` solver.

```
[x,Fval,exitFlag,Output] = ga(FitnessFunction,numberOfVariables);
```

```
Optimization terminated: average change in the fitness value less than options.FunctionTolerance
```

```
fprintf('The best function value found is: %g\n', Fval);
```

```
The best function value found is: -186.484
```

Run `ga` again.

```
[x,Fval,exitFlag,Output] = ga(FitnessFunction,numberOfVariables);
```

```
Optimization terminated: average change in the fitness value less than options.FunctionTolerance
```

```
fprintf('The best function value found is: %g\n', Fval);
```

```
The best function value found is: -185.867
```

`ga` gives different results in the two runs because the state of the random number generator changes from one run to another.

If you want to reproduce your results before you run `ga`, you can save the state of the random number stream.

```
thestate = rng;
```

Run `ga`.

```
[x,Fval,exitFlag,Output] = ga(FitnessFunction,numberOfVariables);
```

```
Optimization terminated: average change in the fitness value less than options.FunctionTolerance
```

```
fprintf('The best function value found is: %g\n', Fval);
```

```
The best function value found is: -186.467
```

Reset the stream and rerun `ga`. The results are identical to the previous run.

```
rng(thestate);
```

```
[x,Fval,exitFlag,Output] = ga(FitnessFunction,numberOfVariables);
```

```
Optimization terminated: average change in the fitness value less than options.FunctionTolerance
```

```
fprintf('The best function value found is: %g\n', Fval);
```

```
The best function value found is: -186.467
```

If you run `ga` before specifying to reproduce the results, you can reset the random number generator as long as you have the output structure.

```
strm = RandStream.getGlobalStream;
strm.State = Output.rngstate.State;
```

Rerun `ga`. Again, the results are identical.

```
[x,Fval,exitFlag,Output] = ga(FitnessFunction,numberOfVariables);
```

```
Optimization terminated: average change in the fitness value less than options.FunctionTolerance
```

```
fprintf('The best function value found is: %g\n', Fval);
```

```
The best function value found is: -186.467
```

Modify Stopping Criteria

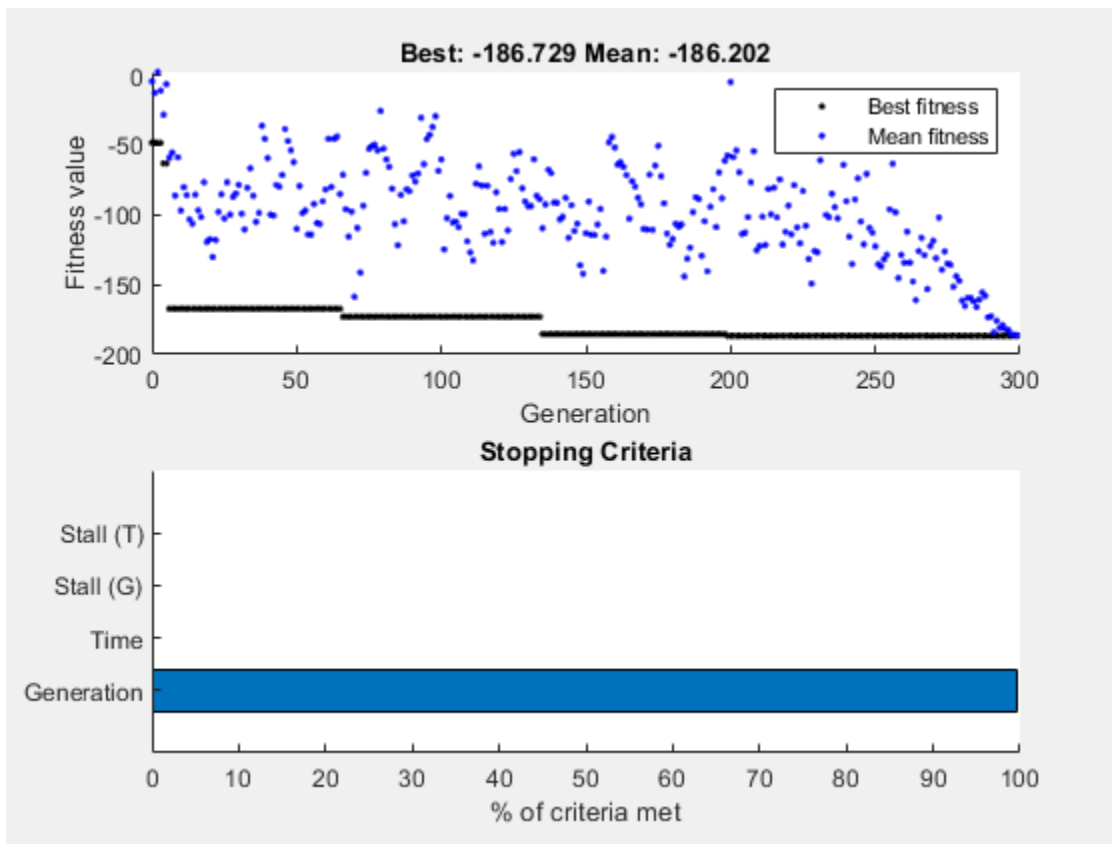
`ga` uses four different criteria to determine when to stop the solver. `ga` stops when it reaches the maximum number of generations; by default, this number is 100 times the number of variables. `ga` also detects if the best fitness value does not change for some time given in seconds (stall time limit), or for some number of generations (maximum stall generations). Another criteria is the maximum time limit in seconds. Modify the stopping criteria to increase the maximum number of generations to 300 and the maximum stall generations to 100.

```
opts = optimoptions(opts, 'MaxGenerations', 300, 'MaxStallGenerations', 100);
```

Rerun the `ga` solver.

```
[x,Fval,exitFlag,Output] = ga(FitnessFunction,numberOfVariables,[],[],[], ...
    [],[],[],[],opts);
```

```
Optimization terminated: average change in the fitness value less than options.FunctionTolerance
```



```
fprintf('The number of generations is: %d\n', Output.generations);
```

```
The number of generations is: 299
```

```
fprintf('The number of function evaluations is: %d\n', Output.funccount);
```

```
The number of function evaluations is: 2702
```

```
fprintf('The best function value found is: %g\n', Fval);
```

```
The best function value found is: -186.729
```

Specify ga Operators

ga starts with a random set of points in the population and uses operators to produce the next generation of the population. The different operators are scaling, selection, crossover, and mutation. The toolbox provides several functions to specify for each operator. Specify `fitscalingprop` for `FitnessScalingFcn` and `selectiontournament` for `SelectionFcn`.

```
opts = optimoptions(@ga, 'SelectionFcn', @selectiontournament, ...
    'FitnessScalingFcn', @fitscalingprop);
```

Rerun ga.

```
[x, Fval, exitFlag, Output] = ga(FitnessFunction, numberOfVariables, [], [], [], ...
    [], [], [], [], opts);
```

```
Optimization terminated: average change in the fitness value less than options.FunctionTolerance
```

```
fprintf('The number of generations is: %d\n', Output.generations);
```

```
The number of generations is: 52
```

```
fprintf('The number of function evaluations is: %d\n', Output.funccount);
```

```
The number of function evaluations is: 2497
```

```
fprintf('The best function value found is: %g\n', Fval);
```

```
The best function value found is: -186.417
```

The best function value can improve or get worse based on the specified operators. Experimenting with different operators is often the best way to determine which set of operators works best for your problem.

See Also

optimoptions

More About

- “Set and Change Options” on page 2-9
- “How the Genetic Algorithm Works” on page 8-15

Mixed Integer ga Optimization

In this section...

“Solving Mixed Integer Optimization Problems” on page 8-40

“Characteristics of the Integer ga Solver” on page 8-41

“Effective Integer ga” on page 8-45

“Integer ga Algorithm” on page 8-45

Solving Mixed Integer Optimization Problems

ga can solve problems when certain variables are integer-valued. Give `intcon`, a vector of the x components that are integers:

```
[x,fval,exitflag] = ga(fitnessfcn,nvars,A,b,[],[],...
    lb,ub,nonlcon,intcon,options)
```

`intcon` is a vector of positive integers that contains the x components that are integer-valued. For example, if you want to restrict $x(2)$ and $x(10)$ to be integers, set `intcon` to `[2,10]`.

The `surrogateopt` solver also accepts integer constraints.

Note Restrictions exist on the types of problems that `ga` can solve with integer variables. In particular, `ga` does not accept nonlinear equality constraints when there are integer variables. For details, see “Characteristics of the Integer ga Solver” on page 8-41.

Tip `ga` solves integer problems best when you provide lower and upper bounds for every x component.

Mixed Integer Optimization of Rastrigin's Function

This example shows how to find the minimum of Rastrigin's function restricted so the first component of x is an integer. The components of x are further restricted to be in the region $5\pi \leq x(1) \leq 20\pi$, $-20\pi \leq x(2) \leq -4\pi$.

Set up the bounds for your problem

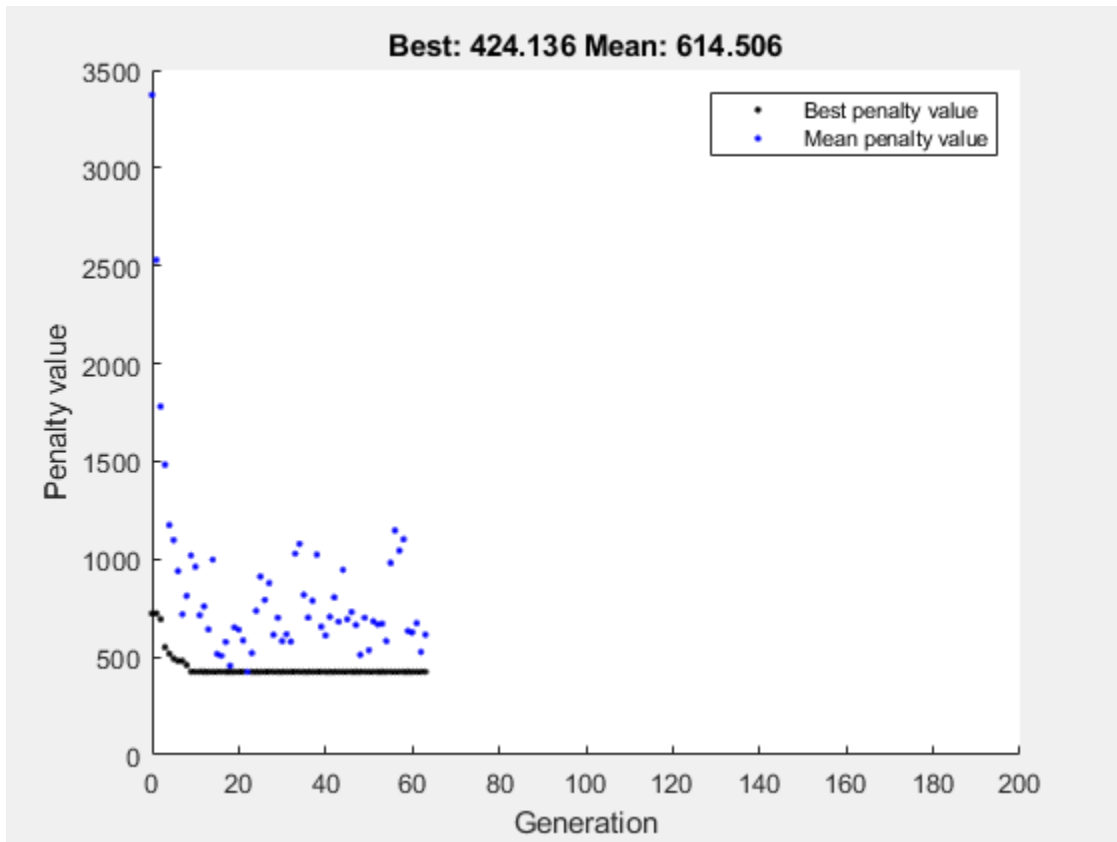
```
lb = [5*pi,-20*pi];
ub = [20*pi,-4*pi];
```

Set a plot function so you can view the progress of ga

```
opts = optimoptions('ga','PlotFcn',@gaplotbestf);
```

Call the ga solver where $x(1)$ has integer values

```
rng(1,'twister') % for reproducibility
intcon = 1;
[x,fval,exitflag] = ga(@rastriginsfcn,2,[],[],[],[],...
    lb,ub,[],intcon,opts)
```

Optimization terminated: average change in the penalty fitness value less than options.FunctionTolerance and constraint violation is less than options.ConstraintTolerance.

`x = 1x2`

`16.0000 -12.9325`

`fval = 424.1355`

`exitflag = 1`

`ga` converges quickly to the solution.

Characteristics of the Integer `ga` Solver

There are some restrictions on the types of problems that `ga` can solve when you include integer constraints:

- No nonlinear equality constraints. Any nonlinear constraint function must return `[]` for the nonlinear equality constraint. For a possible workaround, see “Integer Programming with a Nonlinear Equality Constraint” on page 8-42.
- Only `doubleVector` population type.
- No hybrid function. `ga` overrides any setting of the `HybridFcn` option.
- `ga` ignores the `ParetoFraction`, `DistanceMeasureFcn`, `InitialPenalty`, and `PenaltyFactor` options.

The listed restrictions are mainly natural, not arbitrary. For example, no hybrid functions support integer constraints. So `ga` does not use hybrid functions when there are integer constraints.

Integer Programming with a Nonlinear Equality Constraint

This example attempts to locate the minimum of the Ackley function (included when you run this example) in five dimensions with these constraints:

- $x(1)$, $x(3)$, and $x(5)$ are integers.
- $\text{norm}(x) = 4$.

The `ga` solver does not support nonlinear equality constraints, only nonlinear inequality constraints. This example shows a workaround that applies for some problems, but is not guaranteed to work.

The Ackley function is difficult to minimize. Adding integer and equality constraints increases the difficulty.

To include the nonlinear equality constraint, give a small tolerance `tol` that allows the norm of x to be within `tol` of 4. Without a tolerance, the nonlinear equality constraint is never satisfied, and the solver does not reach a feasible solution.

Write the expression $\text{norm}(x) = 4$ as two “less than zero” inequalities.

$$\|x - 4\| \leq 0$$

$$-\|x - 4\| \leq 0.$$

Allow a small tolerance in the inequalities.

$$\|x - 4\| - \text{tol} \leq 0$$

$$-\|x - 4\| - \text{tol} \leq 0.$$

Write a nonlinear inequality constraint function `eqCon` that implements these inequalities.

```
type eqCon
function [c, ceq] = eqCon(x)

ceq = [];
rad = 4;
tol = 1e-3;
confcnval = norm(x) - rad;
c = [confcnval - tol; -confcnval - tol];
```

Set these options:

- `MaxStallGenerations` = 50 — Allow the solver to try for a while.
- `FunctionTolerance` = 1e-10 — Specify a stricter stopping criterion than usual.
- `MaxGenerations` = 500 — Allow more generations than default.
- `PlotFcn` = `@gaplotbestfun` — Observe the optimization.

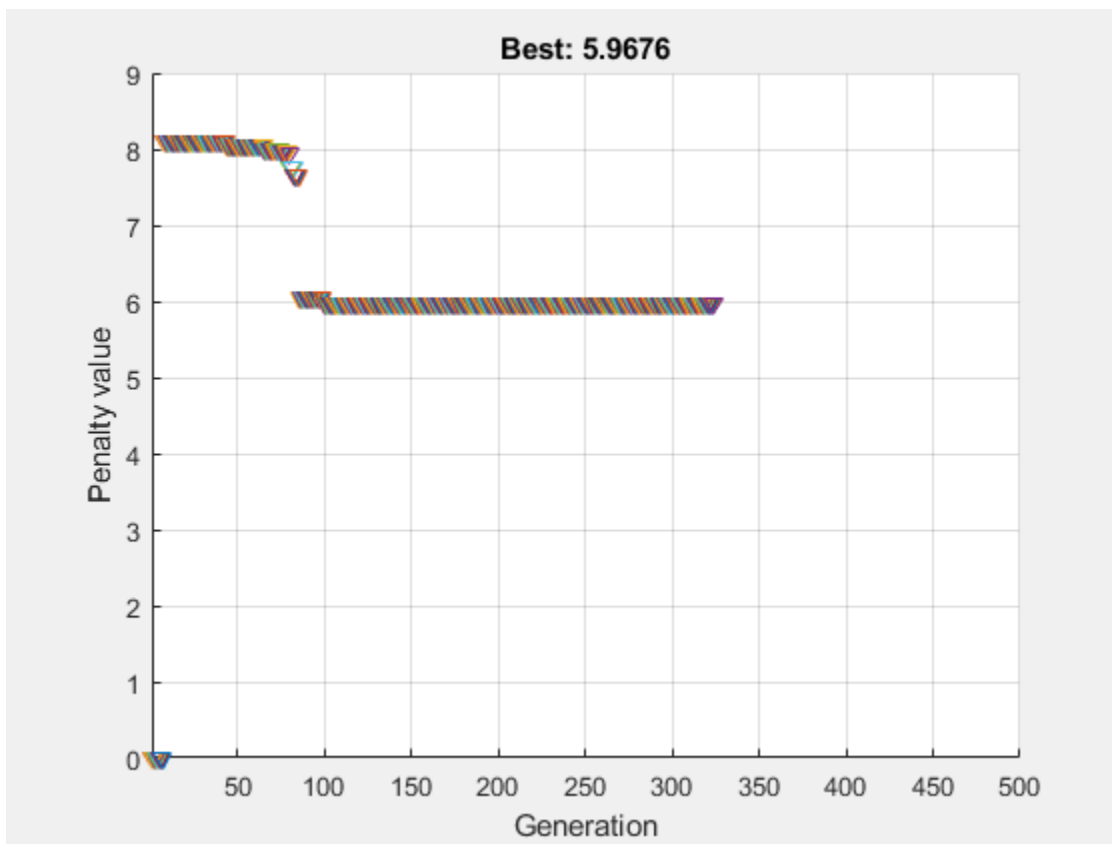
```
opts = optimoptions('ga', 'MaxStallGenerations', 50, 'FunctionTolerance', 1e-10, ...
    'MaxGenerations', 500, 'PlotFcn', @gaplotbestfun);
```

Set lower and upper bounds to help the solver.

```
nVar = 5;
lb = -5*ones(1,nVar);
ub = 5*ones(1,nVar);
```

Solve the problem.

```
rng(0,'twister') % for reproducibility
[x,fval,exitflag] = ga(@ackleyfcn,nVar,[],[],[],[], ...
    lb,ub,@eqCon,[1 3 5],opts);
```



Optimization terminated: average change in the penalty fitness value less than options.FunctionTolerance and constraint violation is less than options.ConstraintTolerance.

Examine the solution.

```
x,fval,exitflag,norm(x)
```

```
x = 1x5
```

```
0    0.9706    1.0000    3.6158   -1.0000
```

```
fval = 5.9676
```

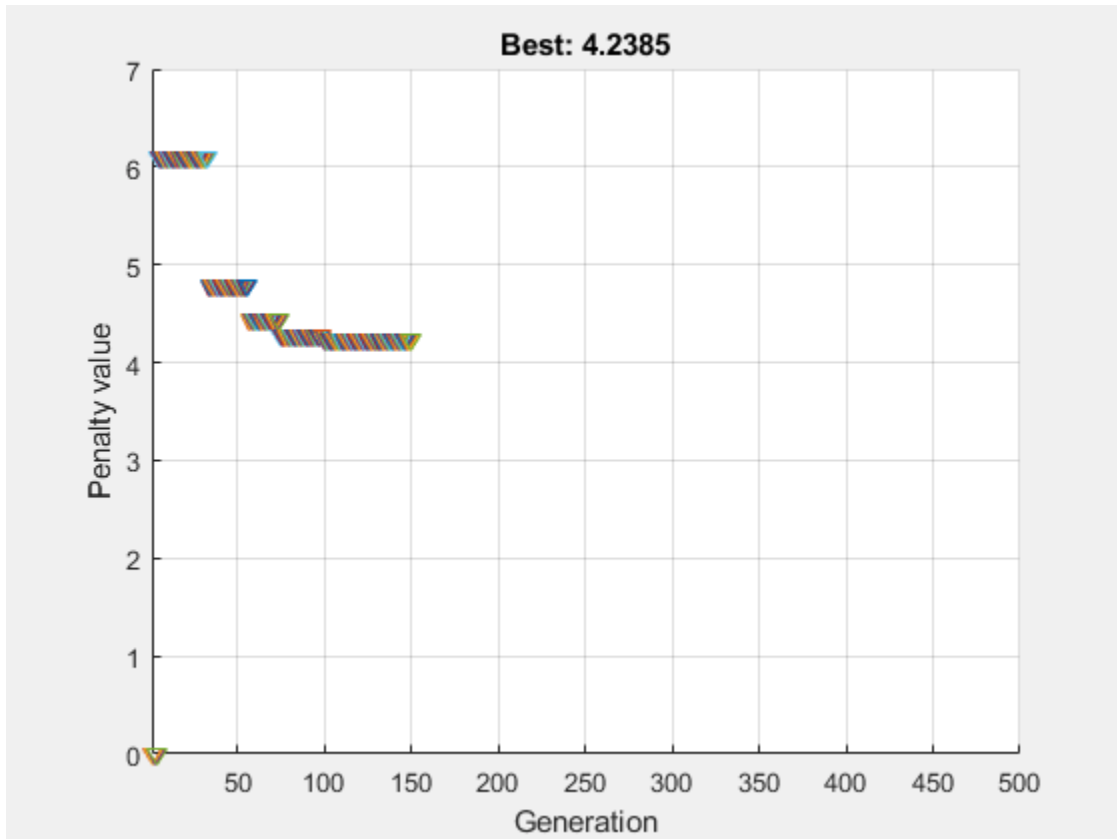
```
exitflag = 1
```

```
ans = 4.0020
```

The odd x components are integers, as specified. The norm of x is 4, to within the given relative tolerance of $1e-3$.

Despite the positive exit flag, the solution is not the global optimum. Run the problem again with a larger population and examine the solution.

```
opts = optimoptions(opts, 'Display', 'off', 'PopulationSize', 400);
[x2, fval2, exitflag2] = ga(@ackleyfcn, nVar, [], [], [], [], ...
    lb, ub, @eqCon, [1 3 5], opts);
```



Examine the second solution.

```
x2, fval2, exitflag2, norm(x2)
```

```
x2 = 1×5
```

```
-1.0000    2.0082   -1.0000   -2.9954    1.0000
```

```
fval2 = 4.2385
```

```
exitflag2 = 1
```

```
ans = 4.0006
```

The second run gives a better solution (lower fitness function value). Again, the odd x components are integers, and the norm of x_2 is 4, to within the given relative tolerance of $1e-3$.

Be aware that this procedure can fail; `ga` has difficulty with simultaneous integer and nonlinear equality constraints.

Effective Integer ga

To use `ga` most effectively on integer problems, follow these guidelines.

- Bound each component as tightly as you can. This practice gives `ga` the smallest search space, enabling `ga` to search most effectively.
- If you cannot bound a component, then specify an appropriate initial range. By default, `ga` creates an initial population with range $[-1e4, 1e4]$ for each component. A smaller or larger initial range can give better results when the default value is inappropriate. To change the initial range, use the `InitialPopulationRange` option.
- If you have more than 10 variables, set a population size that is larger than default by using the `PopulationSize` option. The default value is 200 for six or more variables. For a large population size:
 - `ga` can take a long time to converge. If you reach the maximum number of generations (exit flag 0), increase the value of the `MaxGenerations` option.
 - Decrease the mutation rate. To do so, increase the value of the `CrossoverFraction` option from its default of 0.8 to 0.9 or higher.
 - Increase the value of the `EliteCount` option from its default of $0.05 * \text{PopulationSize}$ to $0.1 * \text{PopulationSize}$ or higher.

For information on options, see the `ga` options input argument.

Integer ga Algorithm

Integer programming with `ga` involves several modifications of the basic algorithm (see “How the Genetic Algorithm Works” on page 8-15). For integer programming:

- By default, special creation, crossover, and mutation functions enforce variables to be integers. For details, see Deep et al. [2].
- If you use nondefault creation, crossover, or mutation functions, `ga` enforces linear feasibility and feasibility with respect to integer constraints at each iteration.
- The genetic algorithm attempts to minimize a penalty function, not the fitness function. The penalty function includes a term for infeasibility. This penalty function is combined with binary tournament selection by default to select individuals for subsequent generations. The penalty function value of a member of a population is:
 - If the member is feasible, the penalty function is the fitness function.
 - If the member is infeasible, the penalty function is the maximum fitness function among feasible members of the population, plus a sum of the constraint violations of the (infeasible) point.

For details of the penalty function, see Deb [1].

References

- [1] Deb, Kalyanmoy. *An efficient constraint handling method for genetic algorithms*. Computer Methods in Applied Mechanics and Engineering, 186(2-4), pp. 311-338, 2000.
- [2] Deep, Kusum, Krishna Pratap Singh, M.L. Kansal, and C. Mohan. *A real coded genetic algorithm for solving integer and mixed integer optimization problems*. Applied Mathematics and Computation, 212(2), pp. 505-518, 2009.

See Also

Related Examples

- “Solve a Mixed-Integer Engineering Design Problem Using the Genetic Algorithm” on page 8-47
- “Mixed-Integer Surrogate Optimization” on page 11-62
- “Solve Nonlinear Problem with Integer and Nonlinear Constraints” on page 11-82

Solve a Mixed-Integer Engineering Design Problem Using the Genetic Algorithm

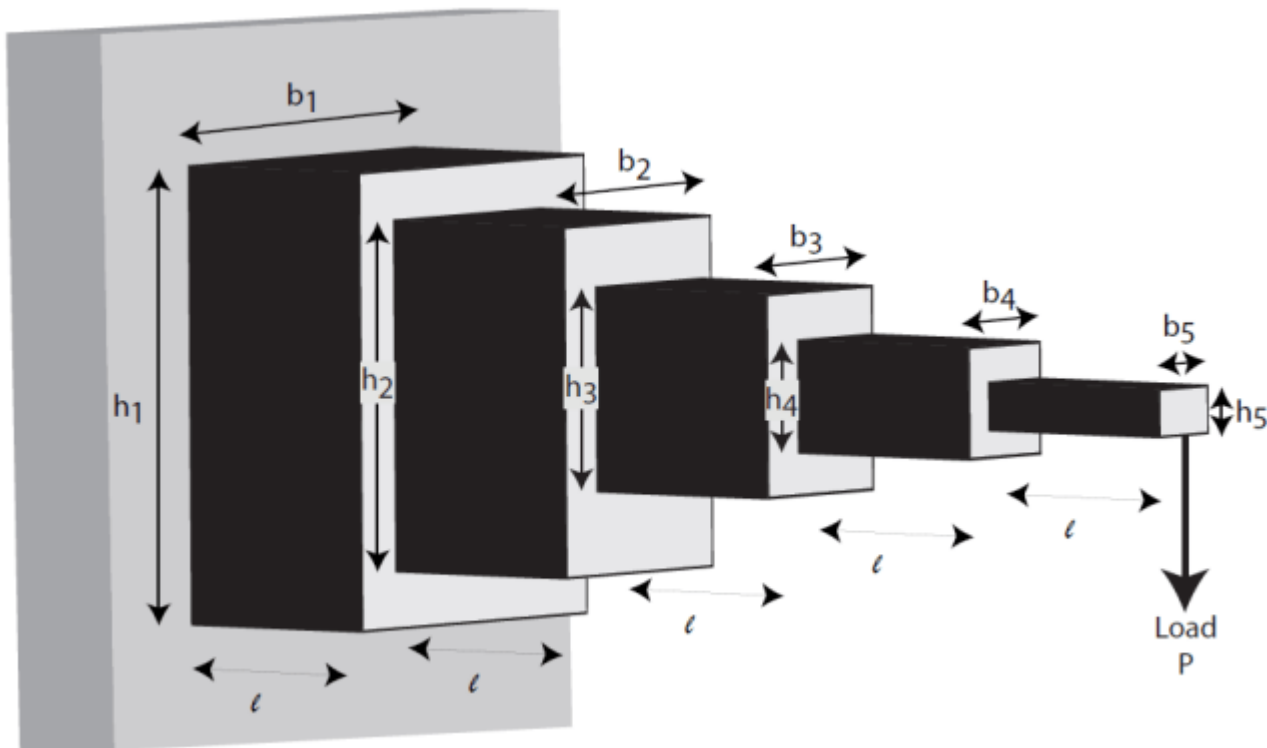
This example shows how to solve a mixed integer engineering design problem using the Genetic Algorithm (ga) solver in Global Optimization Toolbox.

The problem illustrated in this example involves the design of a stepped cantilever beam. In particular, the beam must be able to carry a prescribed end load. We will solve a problem to minimize the beam volume subject to various engineering design constraints.

In this example we will solve two bounded versions of the problem published in [1].

Stepped Cantilever Beam Design Problem

A stepped cantilever beam is supported at one end and a load is applied at the free end, as shown in the figure below. The beam must be able to support the given load, P , at a fixed distance L from the support. Designers of the beam can vary the width (b_i) and height (h_i) of each section. We will assume that each section of the cantilever has the same length, l .



Volume of the beam

The volume of the beam, V , is the sum of the volume of the individual sections

$$V = l(b_1h_1 + b_2h_2 + b_3h_3 + b_4h_4 + b_5h_5)$$

Constraints on the Design : 1 - Bending Stress

Consider a single cantilever beam, with the center of coordinates at the center of its cross section at the free end of the beam. The bending stress at a point (x, y, z) in the beam is given by the following equation

$$\sigma_b = M(x)y/I$$

where $M(x)$ is the bending moment at x , x is the distance from the end load and I is the area moment of inertia of the beam.

Now, in the stepped cantilever beam shown in the figure, the maximum moment of each section of the beam is PD_i , where D_i is the maximum distance from the end load, P , for each section of the beam. Therefore, the maximum stress for the i -th section of the beam, σ_i , is given by

$$\sigma_i = PD_i(h_i/2)/I_i$$

where the maximum stress occurs at the edge of the beam, $y = h_i/2$. The area moment of inertia of the i -th section of the beam is given by

$$I_i = b_i h_i^3 / 12$$

Substituting this into the equation for σ_i gives

$$\sigma_i = 6PD_i/b_i h_i^2$$

The bending stress in each part of the cantilever should not exceed the maximum allowable stress, σ_{max} . Consequently, we can finally state the five bending stress constraints (one for each step of the cantilever)

$$\frac{6Pl}{b_5 h_5^2} \leq \sigma_{max}$$

$$\frac{6P(2l)}{b_4 h_4^2} \leq \sigma_{max}$$

$$\frac{6P(3l)}{b_3 h_3^2} \leq \sigma_{max}$$

$$\frac{6P(4l)}{b_2 h_2^2} \leq \sigma_{max}$$

$$\frac{6P(5l)}{b_1 h_1^2} \leq \sigma_{max}$$

Constraints on the Design : 2 - End deflection

The end deflection of the cantilever can be calculated using Castigliano's second theorem, which states that

$$\delta = \frac{\partial U}{\partial P}$$

where δ is the deflection of the beam, U is the energy stored in the beam due to the applied force, P . The energy stored in a cantilever beam is given by

$$U = \int_0^L M^2/2EI \, dx$$

where M is the moment of the applied force at x .

Given that $M = Px$ for a cantilever beam, we can write the above equation as

$$U = P^2/2E \int_0^l [(x+4l)^2/I_1 + (x+3l)^2/I_2 + (x+2l)^2/I_3 + (x+l)^2/I_4 + x^2/I_5] \, dx$$

where I_n is the area moment of inertia of the n -th part of the cantilever. Evaluating the integral gives the following expression for U .

$$U = (P^2/2)(l^3/3E)(61/I_1 + 37/I_2 + 19/I_3 + 7/I_4 + 1/I_5)$$

Applying Castigliano's theorem, the end deflection of the beam is given by

$$\delta = Pl^3/3E(61/I_1 + 37/I_2 + 19/I_3 + 7/I_4 + 1/I_5)$$

Now, the end deflection of the cantilever, δ , should be less than the maximum allowable deflection, δ_{max} , which gives us the following constraint.

$$Pl^3/3E(61/I_1 + 37/I_2 + 19/I_3 + 7/I_4 + 1/I_5) \leq \delta_{max}$$

Constraints on the Design : 3 - Aspect ratio

For each step of the cantilever, the aspect ratio must not exceed a maximum allowable aspect ratio, a_{max} . That is,

$$h_i/b_i \leq a_{max} \text{ for } i = 1, \dots, 5$$

State the Optimization Problem

We are now able to state the problem to find the optimal parameters for the stepped cantilever beam given the stated constraints.

Let $x_1 = b_1$, $x_2 = h_1$, $x_3 = b_2$, $x_4 = h_2$, $x_5 = b_3$, $x_6 = h_3$, $x_7 = b_4$, $x_8 = h_4$, $x_9 = b_5$ and $x_{10} = h_5$

Minimize:

$$V = l(x_1x_2 + x_3x_4 + x_5x_6 + x_7x_8 + x_9x_{10})$$

Subject to:

$$\frac{6Pl}{x_9x_{10}^2} \leq \sigma_{max}$$

$$\frac{6P(2l)}{x_7x_8^2} \leq \sigma_{max}$$

$$\frac{6P(3l)}{x_5x_6^2} \leq \sigma_{max}$$

$$\frac{6P(4l)}{x_3x_4^2} \leq \sigma_{max}$$

$$\frac{6P(5l)}{x_1x_2^2} \leq \sigma_{max}$$

$$\frac{Pl^3}{E} \left(\frac{244}{x_1x_2^3} + \frac{148}{x_3x_4^3} + \frac{76}{x_5x_6^3} + \frac{28}{x_7x_8^3} + \frac{4}{x_9x_{10}^3} \right) \leq \delta_{max}$$

$$x_2/x_1 \leq 20, x_4/x_3 \leq 20, x_6/x_5 \leq 20, x_8/x_7 \leq 20 \text{ and } x_{10}/x_9 \leq 20$$

The first step of the beam can only be machined to the nearest centimeter. That is, x_1 and x_2 must be integer. The remaining variables are continuous. The bounds on the variables are given below:-

$$1 \leq x_1 \leq 5$$

$$30 \leq x_2 \leq 65$$

$$2.4 \leq x_3, x_5 \leq 3.1$$

$$45 \leq x_4, x_6 \leq 60$$

$$1 \leq x_7, x_9 \leq 5$$

$$30 \leq x_8, x_{10} \leq 65$$

Design Parameters for This Problem

For the problem we will solve in this example, the end load that the beam must support is $P = 50000N$.

The beam lengths and maximum end deflection are:

- Total beam length, $L = 500cm$
- Individual section of beam, $l = 100cm$
- Maximum beam end deflection, $\delta_{max} = 2.7cm$

The maximum allowed stress in each step of the beam, $\sigma_{max} = 14000N/cm^2$

Young's modulus of each step of the beam, $E = 2 \times 10^7 N/cm^2$

Solve the Mixed Integer Optimization Problem

We now solve the problem described in *State the Optimization Problem*.

Define the Fitness and Constraint Functions

Examine the MATLAB® files `cantileverVolume.m` and `cantileverConstraints.m` to see how the fitness and constraint functions are implemented.

A note on the linear constraints: When linear constraints are specified to `ga`, you normally specify them via the `A`, `b`, `Aeq` and `beq` inputs. In this case we have specified them via the nonlinear constraint function. This is because later in this example, some of the variables will become discrete. When there are discrete variables in the problem it is far easier to specify linear constraints in the nonlinear constraint function. The alternative is to modify the linear constraint matrices to work in the transformed variable space, which is not trivial and maybe not possible. Also, in the mixed integer `ga` solver, the linear constraints are not treated any differently to the nonlinear constraints regardless of how they are specified.

Set the Bounds

Create vectors containing the lower bound (`lb`) and upper bound constraints (`ub`).

```
lb = [1 30 2.4 45 2.4 45 1 30 1 30];
ub = [5 65 3.1 60 3.1 60 5 65 5 65];
```

Set the Options

To obtain a more accurate solution, we increase the `PopulationSize`, and `MaxGenerations` options from their default values, and decrease the `EliteCount` and `FunctionTolerance` options. These settings cause `ga` to use a larger population (increased `PopulationSize`), to increase the search of the design space (reduced `EliteCount`), and to keep going until its best member changes by very little (small `FunctionTolerance`). We also specify a plot function to monitor the penalty function value as `ga` progresses.

Note that there are a restricted set of `ga` options available when solving mixed integer problems - see Global Optimization Toolbox User's Guide for more details.

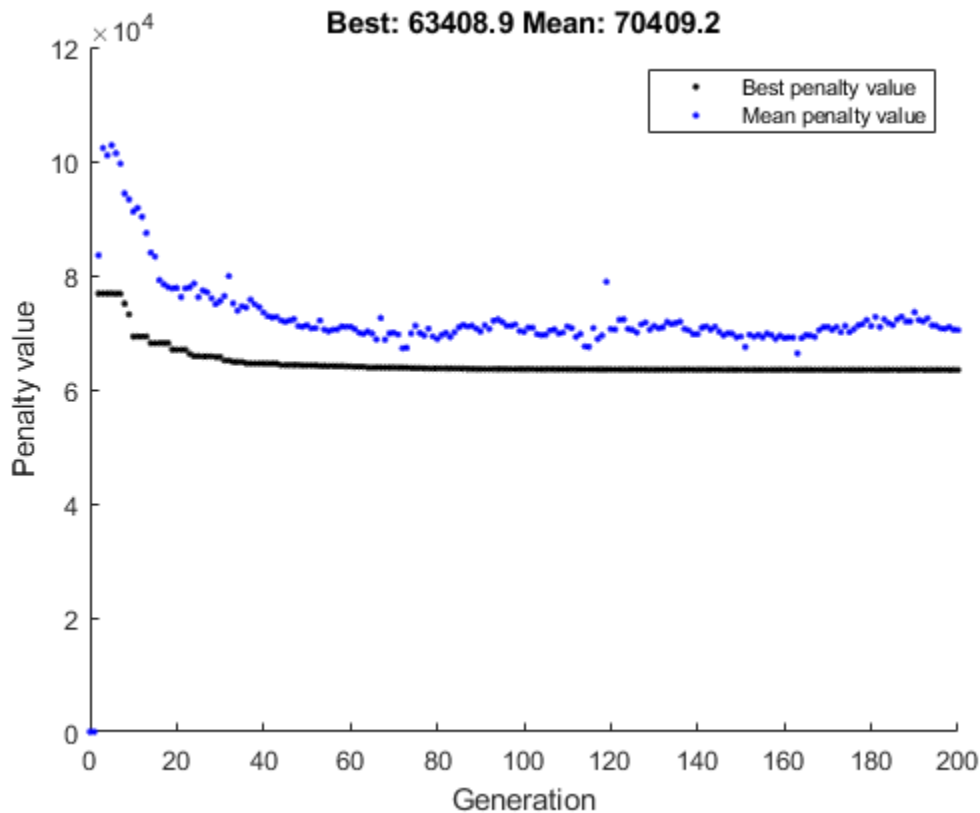
```
opts = optimoptions(@ga, ...
    'PopulationSize', 150, ...
    'MaxGenerations', 200, ...
    'EliteCount', 10, ...
    'FunctionTolerance', 1e-8, ...
    'PlotFcn', @gaplotbestf);
```

Call ga to Solve the Problem

We can now call `ga` to solve the problem. In the problem statement x_1 and x_2 are integer variables. We specify this by passing the index vector `[1 2]` to `ga` after the nonlinear constraint input and before the options input. We also seed and set the random number generator here for reproducibility.

```
rng(0, 'twister');
[xbest, fbest, exitflag] = ga(@cantileverVolume, 10, [], [], [], [], ...
    lb, ub, @cantileverConstraints, [1 2], opts);
```

Optimization terminated: maximum number of generations exceeded.



Analyze the Results

If a problem has integer constraints, `ga` reformulates it internally. In particular, the fitness function in the problem is replaced by a penalty function which handles the constraints. For feasible population members, the penalty function is the same as the fitness function.

The solution returned from `ga` is displayed below. Note that the section nearest the support is constrained to have a width (x_1) and height (x_2) which is an integer value and this constraint has been honored by GA.

```
display(xbest);
```

```
xbest =
```

```
Columns 1 through 7
```

```
3.0000 60.0000 2.8504 57.0057 2.6114 50.6243 2.2132
```

```
Columns 8 through 10
```

```
44.2349 1.7543 35.0595
```

We can also ask `ga` to return the optimal volume of the beam.

```
fprintf('\nCost function returned by ga = %g\n', fbest);
```

Cost function returned by ga = 63408.9

Add Discrete Non-Integer Variable Constraints

The engineers are now informed that the second and third steps of the cantilever can only have widths and heights that are chosen from a standard set. In this section, we show how to add this constraint to the optimization problem. Note that with the addition of this constraint, this problem is identical to that solved in [1].

First, we state the extra constraints that will be added to the above optimization

- The width of the second and third steps of the beam must be chosen from the following set:- [2.4, 2.6, 2.8, 3.1] cm
- The height of the second and third steps of the beam must be chosen from the following set:- [45, 50, 55, 60] cm

To solve this problem, we need to be able to specify the variables x_3 , x_4 , x_5 and x_6 as discrete variables. To specify a component x_j as taking discrete values from the set $S = v_1, \dots, v_k$, optimize with x_j an integer variable taking values from 1 to k , and use $S(x_j)$ as the discrete value. To specify the range (1 to k), set 1 as the lower bound and k as the upper bound.

So, first we transform the bounds on the discrete variables. Each set has 4 members and we will map the discrete variables to an integer in the range [1, 4]. So, to map these variables to be integer, we set the lower bound to 1 and the upper bound to 4 for each of the variables.

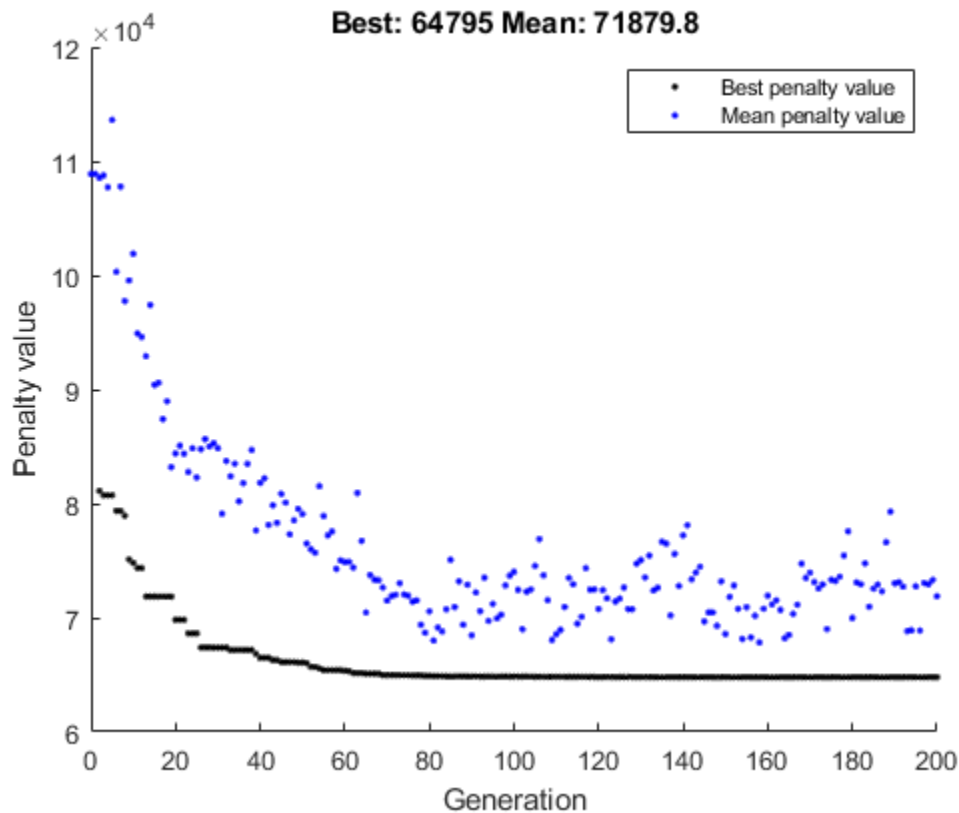
```
lb = [1 30 1 1 1 1 1 30 1 30];
ub = [5 65 4 4 4 4 5 65 5 65];
```

Transformed (integer) versions of x_3 , x_4 , x_5 and x_6 will now be passed to the fitness and constraint functions when the ga solver is called. To evaluate these functions correctly, x_3 , x_4 , x_5 and x_6 need to be transformed to a member of the given discrete set in these functions. To see how this is done, examine the MATLAB files `cantileverVolumeWithDisc.m`, `cantileverConstraintsWithDisc.m` and `cantileverMapVariables.m`.

Now we can call ga to solve the problem with discrete variables. In this case x_1, \dots, x_6 are integers. This means that we pass the index vector 1:6 to ga to define the integer variables.

```
rng(0, 'twister');
[xbestDisc, fbestDisc, exitflagDisc] = ga(@cantileverVolumeWithDisc, ...
    10, [], [], [], [], lb, ub, @cantileverConstraintsWithDisc, 1:6, opts);
```

Optimization terminated: maximum number of generations exceeded.



Analyze the Results

`xbestDisc(3:6)` are returned from `ga` as integers (i.e. in their transformed state). We need to reverse the transform to retrieve the value in their engineering units.

```
xbestDisc = cantileverMapVariables(xbestDisc);
display(xbestDisc);
```

```
xbestDisc =
```

```
Columns 1 through 7
```

```
3.0000 60.0000 3.1000 55.0000 2.6000 50.0000 2.2430
```

```
Columns 8 through 10
```

```
44.8603 1.8279 36.5593
```

As before, the solution returned from `ga` honors the constraint that x_1 and x_2 are integers. We can also see that x_3 , x_5 are chosen from the set [2.4, 2.6, 2.8, 3.1] cm and x_4 , x_6 are chosen from the set [45, 50, 55, 60] cm.

Recall that we have added additional constraints on the variables $x(3)$, $x(4)$, $x(5)$ and $x(6)$. As expected, when there are additional discrete constraints on these variables, the optimal solution has

a higher minimum volume. Note further that the solution reported in [1] has a minimum volume of 64558cm^3 and that we find a solution which is approximately the same as that reported in [1].

```
fprintf('\nCost function returned by ga = %g\n', fbestDisc);
```

```
Cost function returned by ga = 64795
```

Summary

This example illustrates how to use the genetic algorithm solver, `ga`, to solve a constrained nonlinear optimization problem which has integer constraints. The example also shows how to handle problems that have discrete variables in the problem formulation.

References

[1] Thanedar, P. B., and G. N. Vanderplaats. "Survey of Discrete Variable Optimization for Structural Design." *Journal of Structural Engineering* 121 (3), 1995, pp. 301-306.

See Also

More About

- "Mixed Integer `ga` Optimization" on page 8-40

Nonlinear Constraint Solver Algorithms for Genetic Algorithm

In this section...

“Augmented Lagrangian Genetic Algorithm” on page 8-56

“Penalty Algorithm” on page 8-57

Augmented Lagrangian Genetic Algorithm

By default, the genetic algorithm uses the Augmented Lagrangian Genetic Algorithm (ALGA) to solve nonlinear constraint problems without integer constraints. The optimization problem solved by the ALGA algorithm is

$$\min_x f(x)$$

such that

$$\begin{aligned} c_i(x) &\leq 0, \quad i = 1 \dots m \\ ceq_i(x) &= 0, \quad i = m + 1 \dots mt \\ A \cdot x &\leq b \\ Aeq \cdot x &= beq \\ lb &\leq x \leq ub, \end{aligned}$$

where $c(x)$ represents the nonlinear inequality constraints, $ceq(x)$ represents the equality constraints, m is the number of nonlinear inequality constraints, and mt is the total number of nonlinear constraints.

The Augmented Lagrangian Genetic Algorithm (ALGA) attempts to solve a nonlinear optimization problem with nonlinear constraints, linear constraints, and bounds. In this approach, bounds and linear constraints are handled separately from nonlinear constraints. A subproblem is formulated by combining the fitness function and nonlinear constraint function using the Lagrangian and the penalty parameters. A sequence of such optimization problems are approximately minimized using the genetic algorithm such that the linear constraints and bounds are satisfied.

A subproblem formulation is defined as

$$\Theta(x, \lambda, s, \rho) = f(x) - \sum_{i=1}^m \lambda_i s_i \log(s_i - c_i(x)) + \sum_{i=m+1}^{mt} \lambda_i ceq_i(x) + \frac{\rho}{2} \sum_{i=m+1}^{mt} ceq_i(x)^2,$$

where

- The components λ_i of the vector λ are nonnegative and are known as Lagrange multiplier estimates
- The elements s_i of the vector s are nonnegative shifts
- ρ is the positive penalty parameter.

The algorithm begins by using an initial value for the penalty parameter (`InitialPenalty`).

The genetic algorithm minimizes a sequence of subproblems, each of which is an approximation of the original problem. Each subproblem has a fixed value of λ , s , and ρ . When the subproblem is

minimized to a required accuracy and satisfies feasibility conditions, the Lagrangian estimates are updated. Otherwise, the penalty parameter is increased by a penalty factor (`PenaltyFactor`). This results in a new subproblem formulation and minimization problem. These steps are repeated until the stopping criteria are met.

Each subproblem solution represents one generation. The number of function evaluations per generation is therefore much higher when using nonlinear constraints than otherwise.

Choose the Augmented Lagrangian algorithm by setting the `NonlinearConstraintAlgorithm` option to 'auglag' using `optimoptions`.

For a complete description of the algorithm, see the following references:

References

- [1] Conn, A. R., N. I. M. Gould, and Ph. L. Toint. "A Globally Convergent Augmented Lagrangian Algorithm for Optimization with General Constraints and Simple Bounds," *SIAM Journal on Numerical Analysis*, Volume 28, Number 2, pages 545-572, 1991.
- [2] Conn, A. R., N. I. M. Gould, and Ph. L. Toint. "A Globally Convergent Augmented Lagrangian Barrier Algorithm for Optimization with General Inequality Constraints and Simple Bounds," *Mathematics of Computation*, Volume 66, Number 217, pages 261-288, 1997.

Penalty Algorithm

The penalty algorithm is similar to the "Integer ga Algorithm" on page 8-45. In its evaluation of the fitness of an individual, `ga` computes a penalty value as follows:

- If the individual is feasible, the penalty function is the fitness function.
- If the individual is infeasible, the penalty function is the maximum fitness function among feasible members of the population, plus a sum of the constraint violations of the (infeasible) individual.

For details of the penalty function, see Deb [1].

Choose the penalty algorithm by setting the `NonlinearConstraintAlgorithm` option to 'penalty' using `optimoptions`. When you make this choice, `ga` solves the constrained optimization problem as follows.

- 1 `ga` defaults to the `@gacreationnonlinearfeasible` creation function. This function attempts to create a feasible population with respect to all constraints. `ga` creates enough individuals to match the `PopulationSize` option. For details, see "Penalty Algorithm" on page 17-38.
- 2 `ga` overrides your choice of selection function, and uses `@selectiontournament` with two individuals per tournament.
- 3 `ga` proceeds according to "How the Genetic Algorithm Works" on page 8-15, using the penalty function as the fitness measure.

References

- [1] Deb, Kalyanmoy. *An efficient constraint handling method for genetic algorithms*. Computer Methods in Applied Mechanics and Engineering, 186(2-4), pp. 311-338, 2000.

See Also

More About

- “Genetic Algorithm Terminology” on page 8-13
- “How the Genetic Algorithm Works” on page 8-15

Create Custom Plot Function

In this section...

“About Custom Plot Functions” on page 8-59
 “Creating the Custom Plot Function” on page 8-59
 “Using the Custom Plot Function” on page 8-59
 “How the Plot Function Works” on page 8-60

About Custom Plot Functions

If none of the plot functions that come with the software is suitable for the output you want to plot, you can write your own custom plot function, which the genetic algorithm calls at each generation to create the plot. This example shows how to create a plot function that displays the change in the best fitness value from the previous generation to the current generation.

Creating the Custom Plot Function

To create the plot function for this example, copy and paste the following code into a new file in the MATLAB Editor.

```
function state = gaplotchange(options, state, flag)
% GAPLOTCHANGE Plots the logarithmic change in the best score from the
% previous generation.
%
% persistent last_best % Best score in the previous generation

if(strcmp(flag,'init')) % Set up the plot
    xlim([1,options.MaxGenerations]);
    axx = gca;
    axx.YScale = 'log';
    hold on;
    xlabel Generation
    title('Log Absolute Change in Best Fitness Value')
end

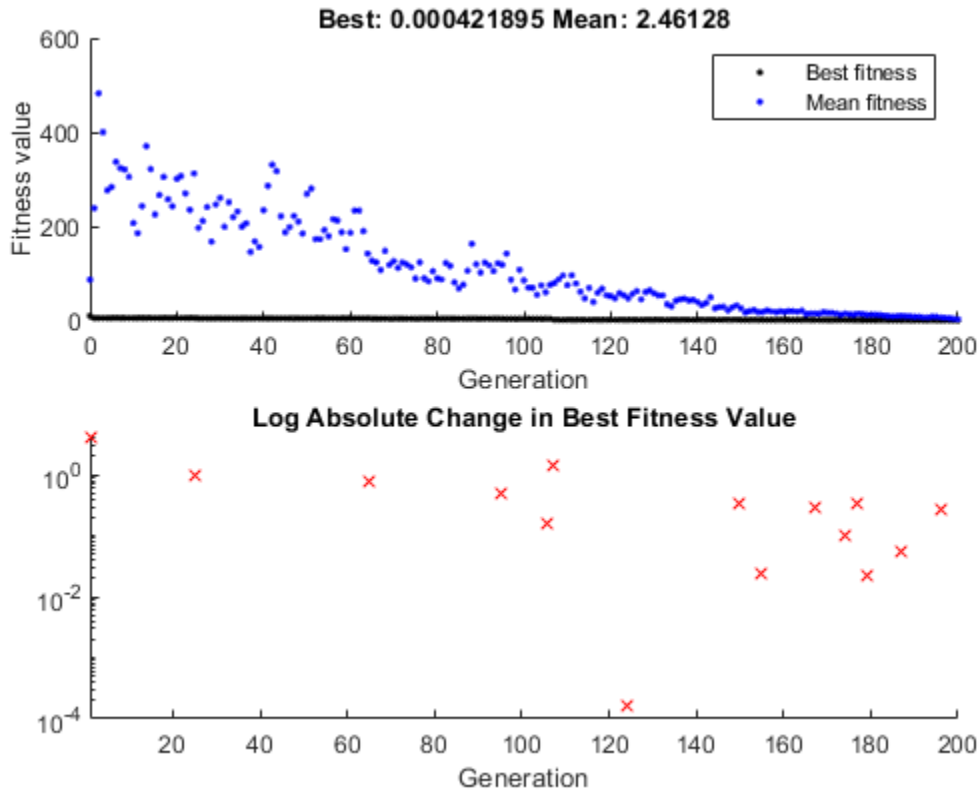
best = min(state.Score); % Best score in the current generation
if state.Generation == 0 % Set last_best to best.
    last_best = best;
else
    change = last_best - best; % Change in best score
    last_best = best;
    if change > 0 % Plot only when the fitness improves
        plot(state.Generation,change,'xr');
    end
end
```

Save the file as `gaplotchange.m` in a folder on the MATLAB path.

Using the Custom Plot Function

To use the custom plot function, include it in the options.

```
rng(100) % For reproducibility
options = optimoptions('ga','PlotFcn',{@gaplotbestf,@gaplotchange});
[x,fval] = ga(@rastriginsfcn,2,[],[],[],[],[],[],[],[],options)
```



Optimization terminated: maximum number of generations exceeded.

x =

```
-0.0003    0.0014
```

fval =

```
4.2189e-04
```

The plot shows only changes that are greater than 0, which are improvements in best fitness. The logarithmic scale enables you to see small changes in the best fitness function that the upper plot does not reveal.

How the Plot Function Works

The plot function uses information contained in the following structures, which the genetic algorithm passes to the function as input arguments:

- `options` — The current options settings
- `state` — Information about the current generation

- `flag` — Current status of the algorithm

The most important lines of the plot function are the following:

- `persistent last_best`

Creates the persistent variable `last_best`—the best score in the previous generation. Persistent variables are preserved over multiple calls to the plot function.

- `xlim([1,options.MaxGenerations]);`

```
axx = gca;
```

```
axx.YScale = 'log';
```

Sets up the plot before the algorithm starts. `options.MaxGenerations` is the maximum number of generations.

- `best = min(state.Score)`

The field `state.Score` contains the scores of all individuals in the current population. The variable `best` is the minimum score. For a complete description of the fields of the structure `state`, see “Structure of the Plot Functions” on page 17-24.

- `change = last_best - best`

The variable `change` is the best score at the previous generation minus the best score in the current generation.

- `if change > 0`

Plot only if there is a change in the best fitness.

- `plot(state.Generation,change,'xr')`

Plots the change at the current generation, whose number is contained in `state.Generation`.

The code for `gaplotchange` contains many of the same elements as the code for `gaplotbestf`, the function that creates the best fitness plot.

See Also

Related Examples

- “Custom Output Function for Genetic Algorithm” on page 8-105
- “Plot Options” on page 17-23

Resume ga

By default, `ga` creates a new initial population each time you run it. However, you might get better results by using the final population from a previous run as the initial population for a new run. To do so, you must have saved the final population from the previous run by calling `ga` with the syntax

```
[x,fval,exitflag,output,final_pop] = ga(@fitnessfcn,nvars);
```

The last output argument is the final population. To run `ga` using `final_pop` as the initial population, enter

```
options = optimoptions('ga','InitialPop',final_pop);  
[x,fval,exitflag,output,final_pop2] = ...  
    ga(@fitnessfcn,nvars,[],[],[],[],[],[],[],options);
```

You can then use `final_pop2`, the final population from the second run, as the initial population for a third run.

For example, minimize Ackley's function, a function of two variables that is available when you run this example.

```
rng(100) % For reproducibility  
[x,fval,exitflag,output,final_pop] = ga(@ackleyfcn,2);
```

```
Optimization terminated: average change in the fitness value less than options.FunctionTolerance
```

Examine the best function value.

```
disp(fval)  
  
    3.5527
```

Try to get a better solution by running `ga` from the final population.

```
options = optimoptions('ga','InitialPopulationMatrix',final_pop);  
[x,fval2,exitflag2,output2,final_pop2] = ...  
    ga(@ackleyfcn,2,[],[],[],[],[],[],[],options);
```

```
Optimization terminated: average change in the fitness value less than options.FunctionTolerance
```

```
disp(fval2)  
  
    2.9886
```

The fitness function value improves significantly.

Try once again to improve the solution.

```
options.InitialPopulationMatrix = final_pop2;  
[x,fval3,exitflag3,output3,final_pop3] = ...  
    ga(@ackleyfcn,2,[],[],[],[],[],[],[],options);
```

```
Optimization terminated: average change in the fitness value less than options.FunctionTolerance
```

```
disp(fval3)  
  
    2.9846
```

This time the improvement is insignificant.

Copyright 2020–2022 The MathWorks, Inc.

See Also

More About

- “How the Genetic Algorithm Works” on page 8-15
- “Reproduce Results” on page 8-67

Options and Outputs

In this section...

“Running ga with the Default Options” on page 8-64
 “Setting Options at the Command Line” on page 8-64
 “Additional Output Arguments” on page 8-65

Running ga with the Default Options

To run the genetic algorithm with the default options, call `ga` with the syntax

```
[x,fval] = ga(@fitnessfun, nvars)
```

The input arguments to `ga` are

- `@fitnessfun` — A function handle to the file that computes the fitness function. “Compute Objective Functions” on page 2-2 explains how to write this file.
- `nvars` — The number of independent variables for the fitness function.

The output arguments are

- `x` — The final point
- `fval` — The value of the fitness function at `x`

For a description of additional input and output arguments, see the reference page for `ga`.

You can run the example described in “Minimize Rastrigin's Function” on page 8-4 from the command line by entering

```
rng(1,'twister') % For reproducibility
% Define Rastrigin's function
rastriginsfcn = @(pop)10.0 * size(pop,2) + sum(pop.^2 - 10.0*cos(2*pi.*pop),2);
[x,fval] = ga(rastriginsfcn,2)
```

This returns

```
Optimization terminated:
  average change in the fitness value less than options.FunctionTolerance.
```

```
x =
  -1.0421  -1.0018
```

```
fval =
  2.4385
```

Setting Options at the Command Line

You can specify any of the options that are available for `ga` by passing options as an input argument to `ga` using the syntax

```
[x,fval] = ga(@fitnessfun,nvars,[],[],[],[],[],[],[],options)
```

This syntax does not specify any linear equality, linear inequality, or nonlinear constraints.

You create options using the function `optimoptions`.


```
options = optimoptions(@ga);
```

This returns `options` with the default values for its fields. `ga` uses these default values if you do not pass in `options` as an input argument.

The value of each option is stored in a field of `options`, such as `options.PopulationSize`. You can display any of these values by entering `options` followed by a period and the name of the field. For example, to display the size of the population for the genetic algorithm, enter

```
options.PopulationSize
```

```
ans =
```

```
'50 when numberOfVariables <= 5, else 200'
```

To create `options` with a field value that is different from the default — for example to set `PopulationSize` to 100 instead of its default value 50 — enter

```
options = optimoptions('ga','PopulationSize',100);
```

This creates `options` with all values set to their defaults except for `PopulationSize`, which is set to 100.

If you now enter,

```
ga(@fitnessfun,nvars,[],[],[],[],[],[],[],[],options)
```

`ga` runs the genetic algorithm with a population size of 100.

If you subsequently decide to change another field in `options`, such as setting `PlotFcn` to `@gaplotbestf`, which plots the best fitness function value at each generation, call `optimoptions` with the syntax

```
options = optimoptions(options,'PlotFcn',@plotbestf);
```

This preserves the current values of all fields of `options` except for `PlotFcn`, which is changed to `@plotbestf`. Note that if you omit the input argument `options`, `optimoptions` resets `PopulationSize` to its default value.

You can also set both `PopulationSize` and `PlotFcn` with the single command

```
options = optimoptions('ga','PopulationSize',100,'PlotFcn',@plotbestf);
```

Additional Output Arguments

To get more information about the performance of the genetic algorithm, you can call `ga` with the syntax

```
[x,fval,exitflag,output,population,scores] = ga(@fitnessfcn, nvars)
```

Besides `x` and `fval`, this function returns the following additional output arguments:

- `exitflag` — Integer value corresponding to the reason the algorithm terminated
- `output` — Structure containing information about the performance of the algorithm at each generation

- `population` — Final population
- `scores` — Final scores

See the `ga` reference page for more information about these arguments.

See Also

`ga`

More About

- “Genetic Algorithm Options” on page 17-23
- “Population Diversity” on page 8-72

Reproduce Results

Because the genetic algorithm is stochastic—that is, it makes random choices—you get slightly different results each time you run the genetic algorithm. The algorithm uses the default MATLAB pseudorandom number stream. For more information about random number streams, see `RandStream`. Each time `ga` calls the stream, its state changes. So that the next time `ga` calls the stream, it returns a different random number. This is why the output of `ga` differs each time you run it.

If you need to reproduce your results exactly, you can call `ga` with an output argument that contains the current state of the default stream, and then reset the state to this value before running `ga` again. For example, to reproduce the output of `ga` applied to Rastrigin's function, call `ga` with the syntax

```
rng(1,'twister') % for reproducibility
% Define Rastrigin's function
rastriginsfcn = @(pop)10.0 * size(pop,2) + sum(pop.^2 - 10.0*cos(2*pi.*pop),2);
[x,fval,exitflag,output] = ga(rastriginsfcn, 2);
```

Suppose the results are

```
x,fval,exitflag
```

```
x =
   -1.0421   -1.0018

fval =
    2.4385

exitflag =
    1
```

The state of the stream is stored in `output.rngstate`. To reset the state, enter

```
stream = RandStream.getGlobalStream;
stream.State = output.rngstate.State;
```

If you now run `ga` a second time, you get the same results as before:

```
[x,fval,exitflag] = ga(rastriginsfcn, 2)
```

```
Optimization terminated: average change in the fitness value less than options.FunctionTolerance
```

```
x =
   -1.0421   -1.0018

fval =
    2.4385

exitflag =
    1
```

Note If you do not need to reproduce your results, it is better not to set the state of the stream, so that you get the benefit of the randomness in the genetic algorithm.

See Also

More About

- “Resume ga” on page 8-62

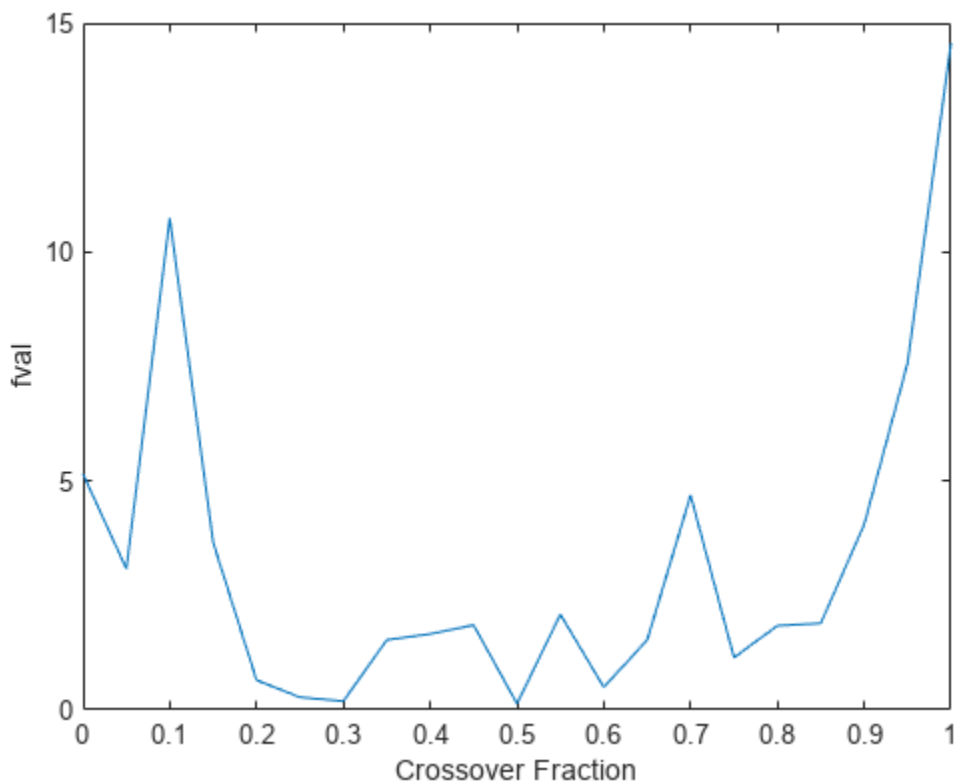
Run ga from a File

The command-line interface enables you to run the genetic algorithm many times, with different options settings, using a file. For example, you can run the genetic algorithm with different settings for Crossover fraction to see which one gives the best results. The following code runs the function `ga` 21 times, varying `options.CrossoverFraction` from 0 to 1 in increments of 0.05, and records the results.

```
options = optimoptions('ga','MaxGenerations',300,'Display','none');
rng default % for reproducibility
record=[];
for n=0:.05:1
    options = optimoptions(options,'CrossoverFraction',n);
    [x,fval]=ga(@rastriginsfcn,2,[],[],[],[],[],[],[],[],options);
    record = [record; fval];
end
```

Plot the values of `fval` against the crossover fraction.

```
plot(0:.05:1, record);
xlabel('Crossover Fraction');
ylabel('fval')
```

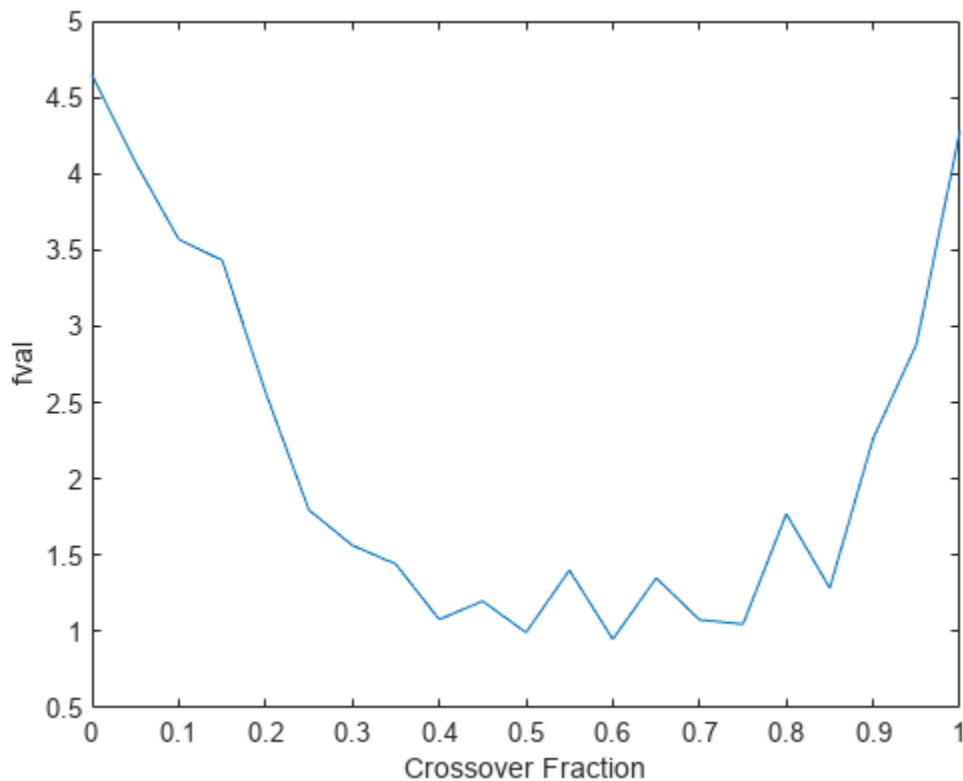


The plot suggests that you get the best results by setting `options.CrossoverFraction` to a value somewhere between 0.4 and 0.8.

You can get a smoother plot of `fval` as a function of the crossover fraction by running `ga` 20 times and averaging the values of `fval` for each crossover fraction.

```
options = optimoptions('ga','MaxGenerations',300,'Display','none');
rng default % for reproducibility
record=[]; fval = zeros(20,1);
for n=0:.05:1
    options = optimoptions(options,'CrossoverFraction', n);
    for i = 1:20
        [x,fval(i)]=ga(@rastriginsfcn,2,[],[],[],[],[],[],[],[],options);
    end
    meanf = mean(fval);
    record = [record; meanf];
end

plot(0:.05:1, record);
xlabel('Crossover Fraction');
ylabel('fval')
```



This plot also suggests the range of best choices for `options.CrossoverFraction` is 0.4 to 0.8.

See Also

`ga`

More About

- “Constrained Minimization Using the Genetic Algorithm” on page 8-27
- “Coding and Minimizing a Fitness Function Using the Genetic Algorithm” on page 8-22

Population Diversity

In this section...

“Importance of Population Diversity” on page 8-72

“Set Initial Range” on page 8-72

“Custom Plot Function and Linear Constraints in ga” on page 8-75

“Setting the Population Size” on page 8-79

Importance of Population Diversity

One of the most important factors that determines the performance of the genetic algorithm performs is the *diversity* of the population. If the average distance between individuals is large, the diversity is high; if the average distance is small, the diversity is low. Getting the right amount of diversity is a matter of trial and error. If the diversity is too high or too low, the genetic algorithm might not perform well.

This section explains how to control diversity by setting the initial range of the population. The topic “Setting the Amount of Mutation” in “Vary Mutation and Crossover” on page 8-83 describes how the amount of mutation affects diversity.

This section also explains how to set the population size on page 8-79.

Set Initial Range

By default, `ga` creates a random initial population using a creation function. You can specify the range of the vectors in the initial population in the `InitialPopulationRange` option.

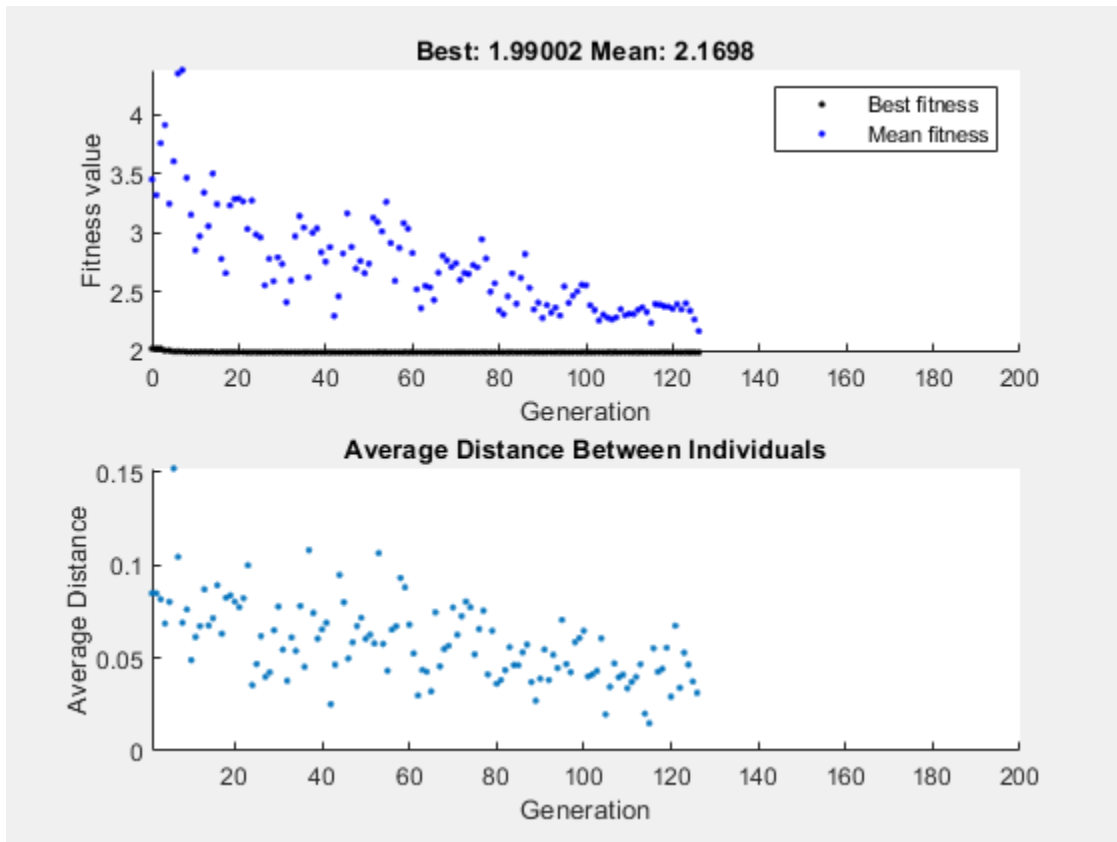
Note: The initial range restricts the range of the points in the initial population by specifying the lower and upper bounds. Subsequent generations can contain points whose entries do not lie in the initial range. Set upper and lower bounds for all generations using the `lb` and `ub` input arguments.

If you know approximately where the solution to a problem lies, specify the initial range so that it contains your guess for the solution. However, the genetic algorithm can find the solution even if it does not lie in the initial range, if the population has enough diversity.

This example shows how the initial range affects the performance of the genetic algorithm. The example uses Rastrigin's function, described in “Minimize Rastrigin's Function” on page 8-4. The minimum value of the function is 0, which occurs at the origin.

```
rng(1) % For reproducibility
fun = @rastriginsfcn;
nvar = 2;
options = optimoptions('ga','PlotFcn',{ 'gaplotbestf','gaplotdistance'},...
    'InitialPopulationRange',[1;1.1]);
[x,fval] = ga(fun,nvar,[],[],[],[],[],[],[],options)
```

Optimization terminated: average change in the fitness value less than options.FunctionTolerance



```
x = 1x2
```

```
0.9942 0.9950
```

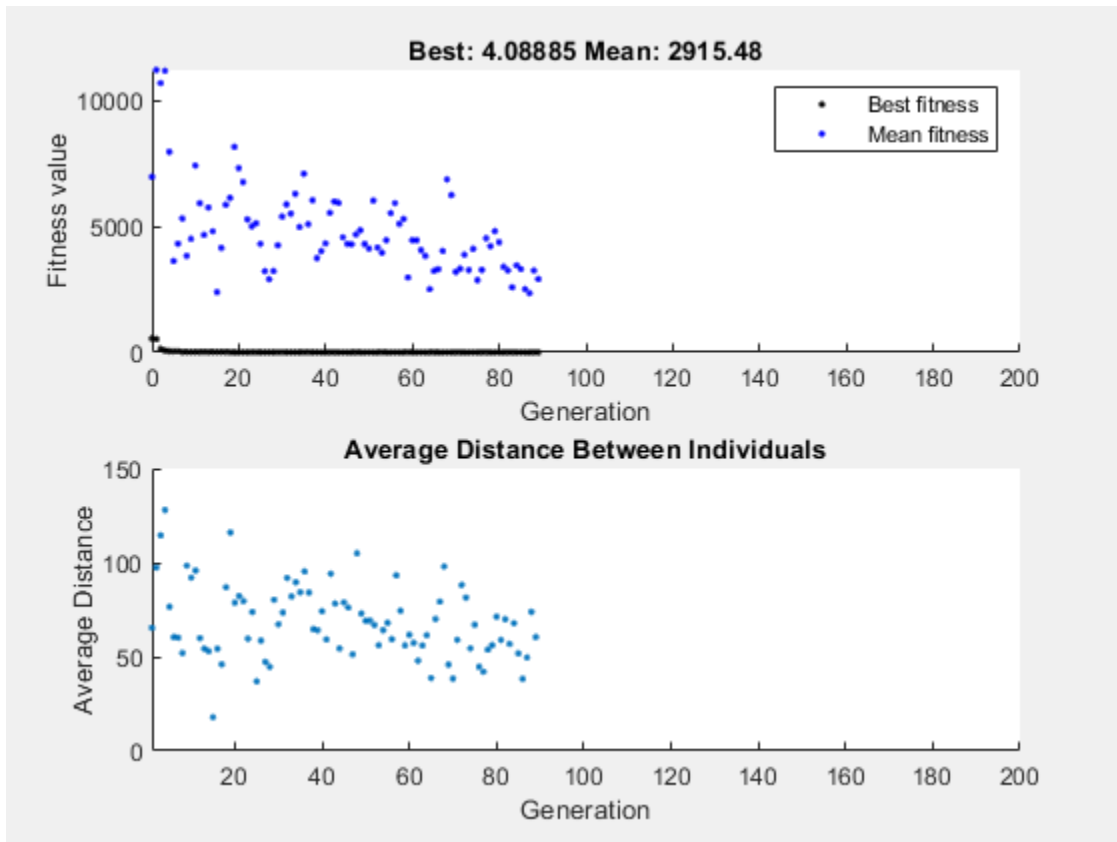
```
fval = 1.9900
```

The upper plot, which displays the best fitness at each generation, shows little progress in lowering the fitness value. The lower plot shows the average distance between individuals at each generation, which is a good measure of the diversity of a population. For this setting of initial range, there is too little diversity for the algorithm to make progress.

Next, try setting the `InitialPopulationRange` to `[1;100]`. This time the results are more variable. The current random number setting causes a fairly typical result.

```
options.InitialPopulationRange = [1;100];
[x,fval] = ga(fun,nvar,[],[],[],[],[],[],[],[],options)
```

```
Optimization terminated: average change in the fitness value less than options.FunctionTolerance
```



```
x = 1x2
```

```
0.9344 -1.0792
```

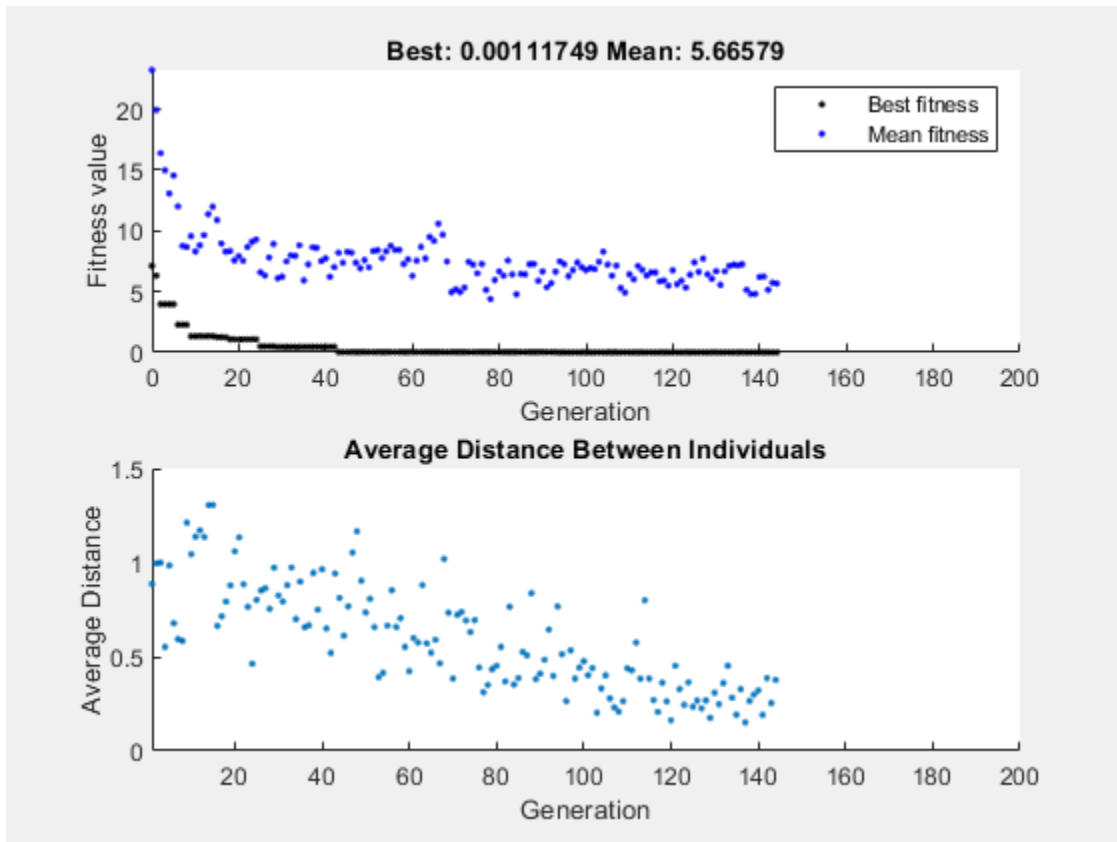
```
fval = 4.0889
```

This time, the genetic algorithm makes progress, but because the average distance between individuals is so large, the best individuals are far from the optimal solution.

Now set the `InitialPopulationRange` to `[1;2]`. This setting is well-suited to the problem.

```
options.InitialPopulationRange = [1;2];
[x,fval] = ga(fun,nvar,[],[],[],[],[],[],[],[],options)
```

```
Optimization terminated: average change in the fitness value less than options.FunctionTolerance
```



```
x = 1x2
```

```
0.0013 0.0020
```

```
fval = 0.0011
```

The suitable diversity usually causes ga to return a better result than in the previous two cases.

Custom Plot Function and Linear Constraints in ga

This example shows how `@gacreationlinearfeasible`, the default creation function for linearly constrained problems, creates a population for `ga`. The population is well-dispersed, and is biased to lie on the constraint boundaries. The example uses a custom plot function.

Fitness Function

The fitness function is `lincontest6`, which is available when you run this example. `lincontest6` is a quadratic function of two variables:

$$f(x) = \frac{x_1^2}{2} + x_2^2 - x_1x_2 - 2x_1 - 6x_2.$$

Custom Plot Function

Save the following code to a file on your MATLAB® path named `gaplotshowpopulation2`.

```
function state = gaplotshowpopulation2(~,state,flag,fcn)
%gaplotshowpopulation2 Plots the population and linear constraints in 2-d.
% STATE = gaplotshowpopulation2(OPTIONS,STATE,FLAG) plots the population
% in two dimensions.
%
% Example:
%   fun = @lincontest6;
%   options = gaoptimset('PlotFcn',{@gaplotshowpopulation2,fun});
%   [x,fval,exitflag] = ga(fun,2,A,b,[],[],lb,[],[],options);

% This plot function works in 2-d only
if size(state.Population,2) > 2
    return;
end
if nargin < 4
    fcn = [];
end
% Dimensions to plot
dimensionsToPlot = [1 2];

switch flag
    % Plot initialization
    case 'init'
        pop = state.Population(:,dimensionsToPlot);
        plotHandle = plot(pop(:,1),pop(:,2),'*');
        set(plotHandle,'Tag','gaplotshowpopulation2')
        title('Population plot in two dimension','interp','none')
        xlabelStr = sprintf('%s %s','Variable ', num2str(dimensionsToPlot(1)));
        ylabelStr = sprintf('%s %s','Variable ', num2str(dimensionsToPlot(2)));
        xlabel(xlabelStr,'interp','none');
        ylabel(ylabelStr,'interp','none');
        hold on;

        % plot the inequalities
        plot([0 1.5],[2 0.5],'m-.') % x1 + x2 <= 2
        plot([0 1.5],[1 3.5/2],'m-.'); % -x1 + 2*x2 <= 2
        plot([0 1.5],[3 0],'m-.'); % 2*x1 + x2 <= 3
        % plot lower bounds
        plot([0 0], [0 2],'m-.'); % lb = [ 0 0];
        plot([0 1.5], [0 0],'m-.'); % lb = [ 0 0];
        set(gca,'xlim',[-0.7,2.2])
        set(gca,'ylim',[-0.7,2.7])
        axx = gcf;
        % Contour plot the objective function
        if ~isempty(fcn)
            range = [-0.5,2;-0.5,2];
            pts = 100;
            span = diff(range)/(pts - 1);
            x = range(1,1): span(1) : range(1,2);
            y = range(2,1): span(2) : range(2,2);

            pop = zeros(pts * pts,2);
            values = zeros(pts,1);
        end
    end
end
```

```

        k = 1;
        for i = 1:pts
            for j = 1:pts
                pop(k,:) = [x(i),y(j)];
                values(k) = fcn(pop(k,:));
                k = k + 1;
            end
        end
        values = reshape(values,pts,pts);
        contour(x,y,values);
        colorbar
    end
    % Show the initial population
    ax = gca;
    fig = figure;
    copyobj(ax,fig);colorbar
    % Pause for three seconds to view the initial plot, then resume
    figure(axx)
    pause(3);
case 'iter'
    pop = state.Population(:,dimensionsToPlot);
    plotHandle = findobj(get(gca,'Children'),'Tag','gaplotshowpopulation2');
    set(plotHandle,'Xdata',pop(:,1),'Ydata',pop(:,2));
end

```

The custom plot function plots the lines representing the linear inequalities and bound constraints, plots level curves of the fitness function, and plots the population as it evolves. This plot function expects to have not only the usual inputs (`options`, `state`, `flag`), but also a function handle to the fitness function, `@lincontest6` in this example. To generate level curves, the custom plot function needs the fitness function.

Problem Constraints

Include bounds and linear constraints.

```

A = [1,1;-1,2;2,1];
b = [2;2;3];
lb = zeros(2,1);

```

Options to Include Plot Function

Set options to include the plot function when `ga` runs.

```

options = optimoptions('ga','PlotFcns',...
    {@gaplotshowpopulation2,@lincontest6});

```

Run Problem and Observe Population

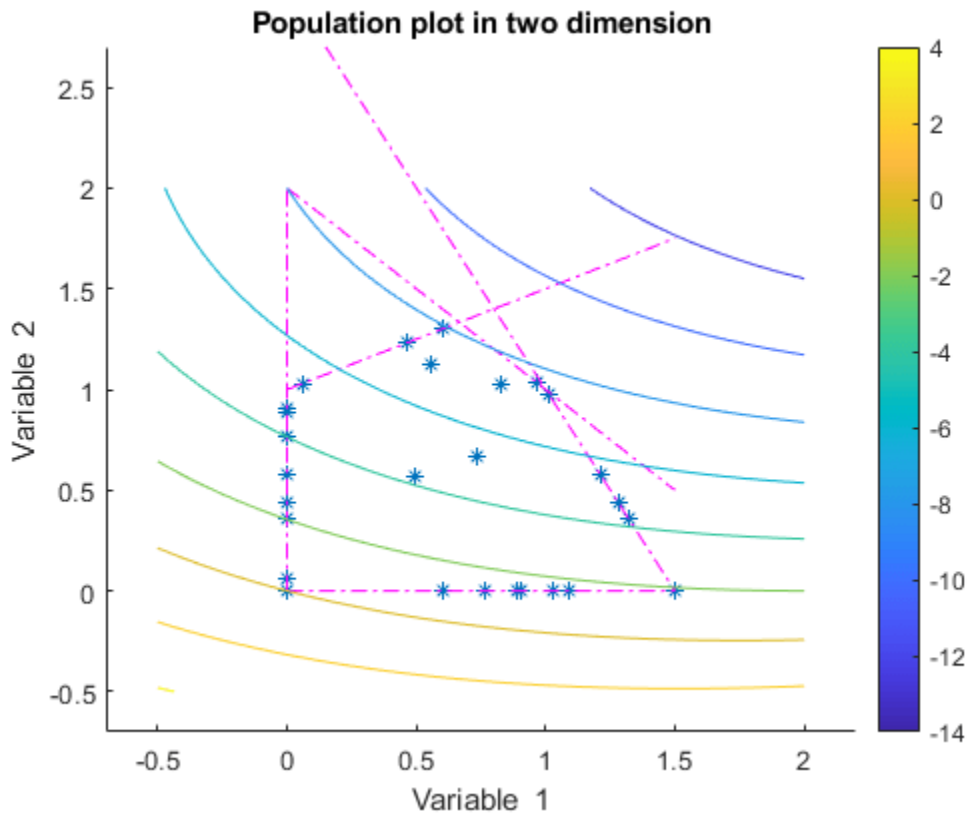
The initial population, in the first plot, has many members on the linear constraint boundaries. The population is reasonably well-dispersed.

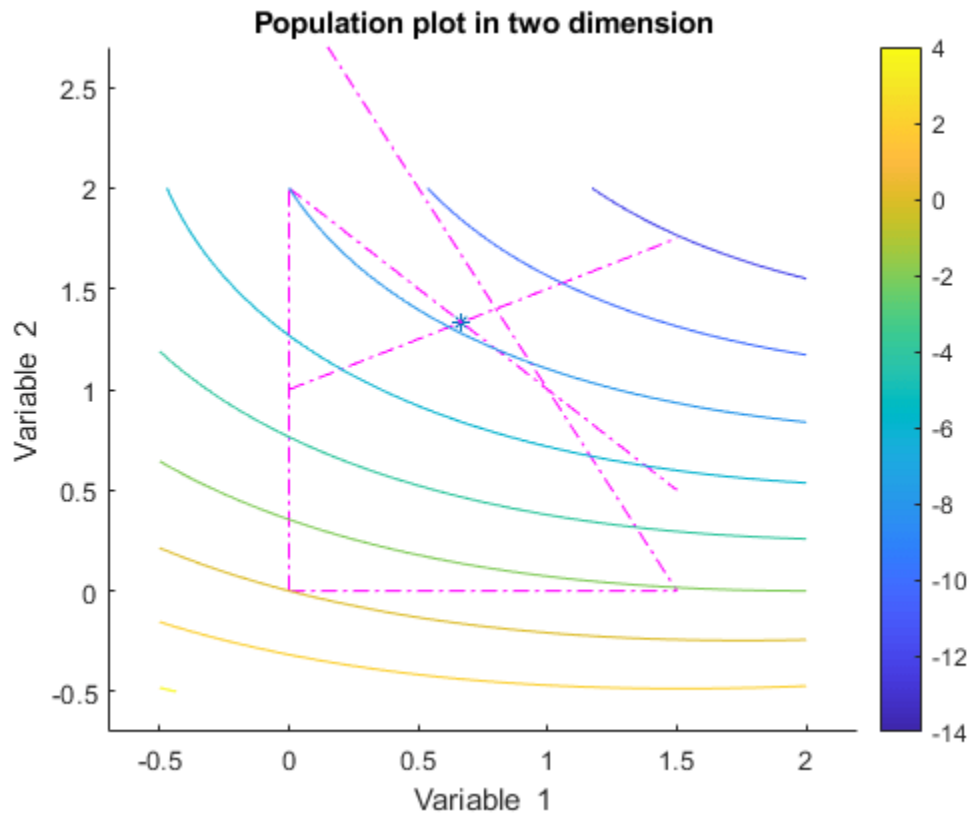
```

rng default % for reproducibility
[x,fval] = ga(@lincontest6,2,A,b,[],[],lb,[],[],options);

```

Optimization terminated: average change in the fitness value less than options.FunctionTolerance





ga converges quickly to a single point, the solution.

Setting the Population Size

The **Population size** field in **Population** options determines the size of the population at each generation. Increasing the population size enables the genetic algorithm to search more points and thereby obtain a better result. However, the larger the population size, the longer the genetic algorithm takes to compute each generation.

Note You should set **Population size** to be at least the value of **Number of variables**, so that the individuals in each population span the space being searched.

You can experiment with different settings for **Population size** that return good results without taking a prohibitive amount of time to run.

See Also

More About

- “Options and Outputs” on page 8-64
- “Global vs. Local Optimization Using ga” on page 8-91

Fitness Scaling

In this section...

“Scaling the Fitness Scores” on page 8-80

“Comparing Rank and Top Scaling” on page 8-81

Scaling the Fitness Scores

Fitness scaling converts the raw fitness scores that are returned by the fitness function to values in a range that is suitable for the selection function. The selection function uses the scaled fitness values to select the parents of the next generation. The selection function assigns a higher probability of selection to individuals with higher scaled values.

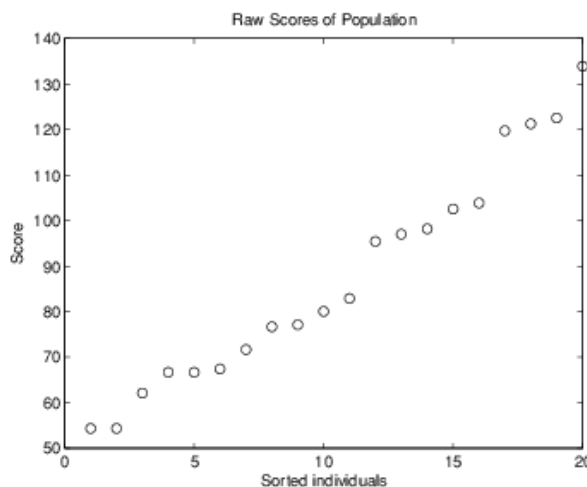
The range of the scaled values affects the performance of the genetic algorithm. If the scaled values vary too widely, the individuals with the highest scaled values reproduce too rapidly, taking over the population gene pool too quickly, and preventing the genetic algorithm from searching other areas of the solution space. On the other hand, if the scaled values vary only a little, all individuals have approximately the same chance of reproduction and the search will progress very slowly.

The default fitness scaling option, Rank, scales the raw scores based on the rank of each individual instead of its score. The rank of an individual is its position in the sorted scores: the rank of the most fit individual is 1, the next most fit is 2, and so on. The rank scaling function assigns scaled values so that

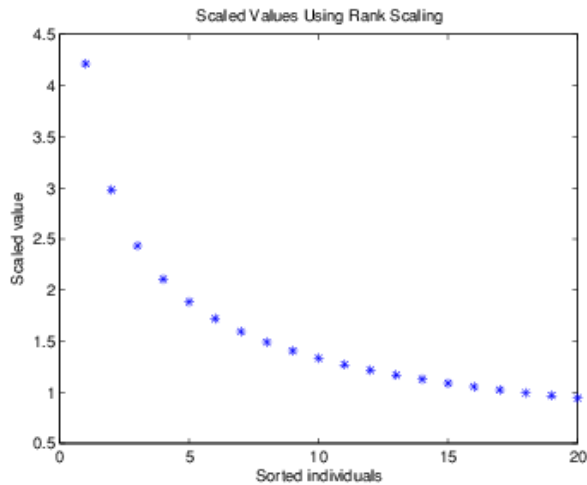
- The scaled value of an individual with rank n is proportional to $1/\sqrt{n}$.
- The sum of the scaled values over the entire population equals the number of parents needed to create the next generation.

Rank fitness scaling removes the effect of the spread of the raw scores.

The following plot shows the raw scores of a typical population of 20 individuals, sorted in increasing order.



The following plot shows the scaled values of the raw scores using rank scaling.

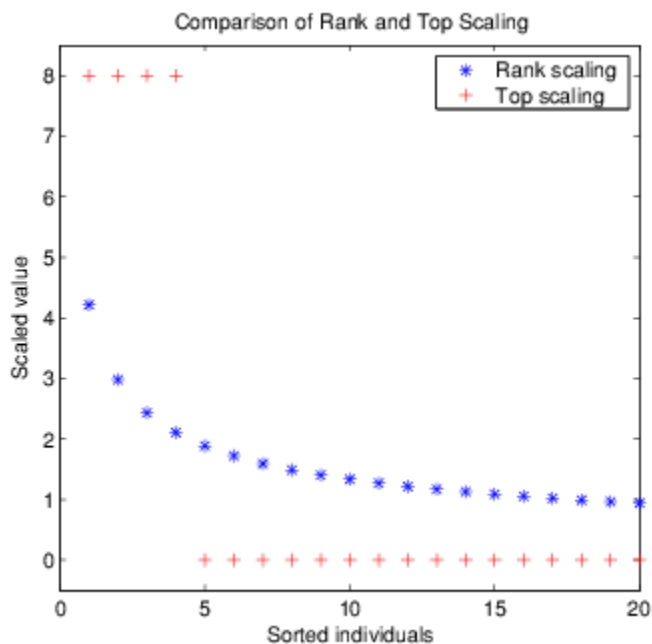


Because the algorithm minimizes the fitness function, lower raw scores have higher scaled values. Also, because rank scaling assigns values that depend only on an individual's rank, the scaled values shown would be the same for any population of size 20 and number of parents equal to 32.

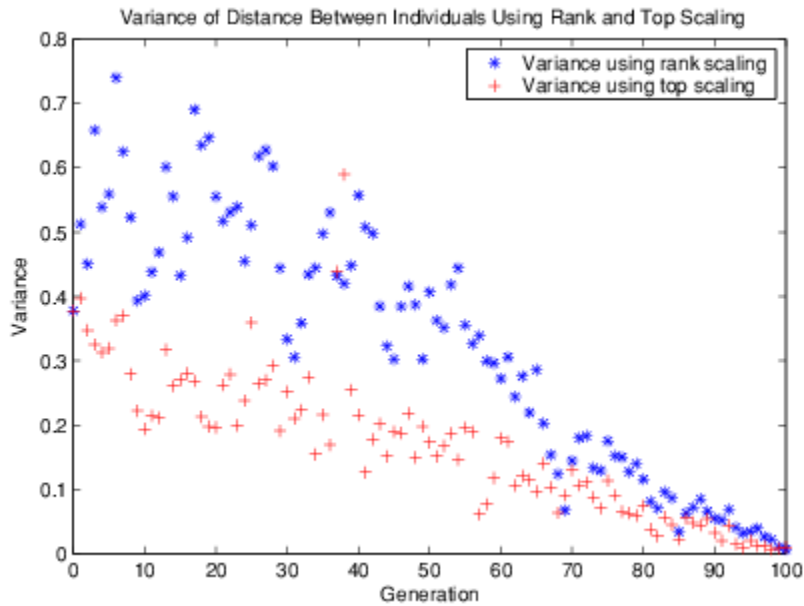
Comparing Rank and Top Scaling

To see the effect of scaling, you can compare the results of the genetic algorithm using rank scaling with one of the other scaling options, such as **Top**. By default, top scaling assigns 40 percent of the fittest individuals to the same scaled value and assigns the rest of the individuals to value 0. Using the default selection function, only 40 percent of the fittest individuals can be selected as parents.

The following figure compares the scaled values of a population of size 20 with number of parents equal to 32 using rank and top scaling.



Because top scaling restricts parents to the fittest individuals, it creates less diverse populations than rank scaling. The following plot compares the variances of distances between individuals at each generation using rank and top scaling.



See Also

External Websites

- "How the Genetic Algorithm Works" on page 8-15

Vary Mutation and Crossover

Setting the Amount of Mutation

The genetic algorithm applies mutations using the `MutationFcn` option. The default mutation option, `@mutationgaussian`, adds a random number, or mutation, chosen from a Gaussian distribution, to each entry of the parent vector. Typically, the amount of mutation, which is proportional to the standard deviation of the distribution, decreases at each new generation. You can control the average amount of mutation that the algorithm applies to a parent in each generation through the `Scale` and `Shrink` inputs that you include in a cell array:

```
options = optimoptions('ga',...
    'MutationFcn',{@mutationgaussian Scale Shrink});
```

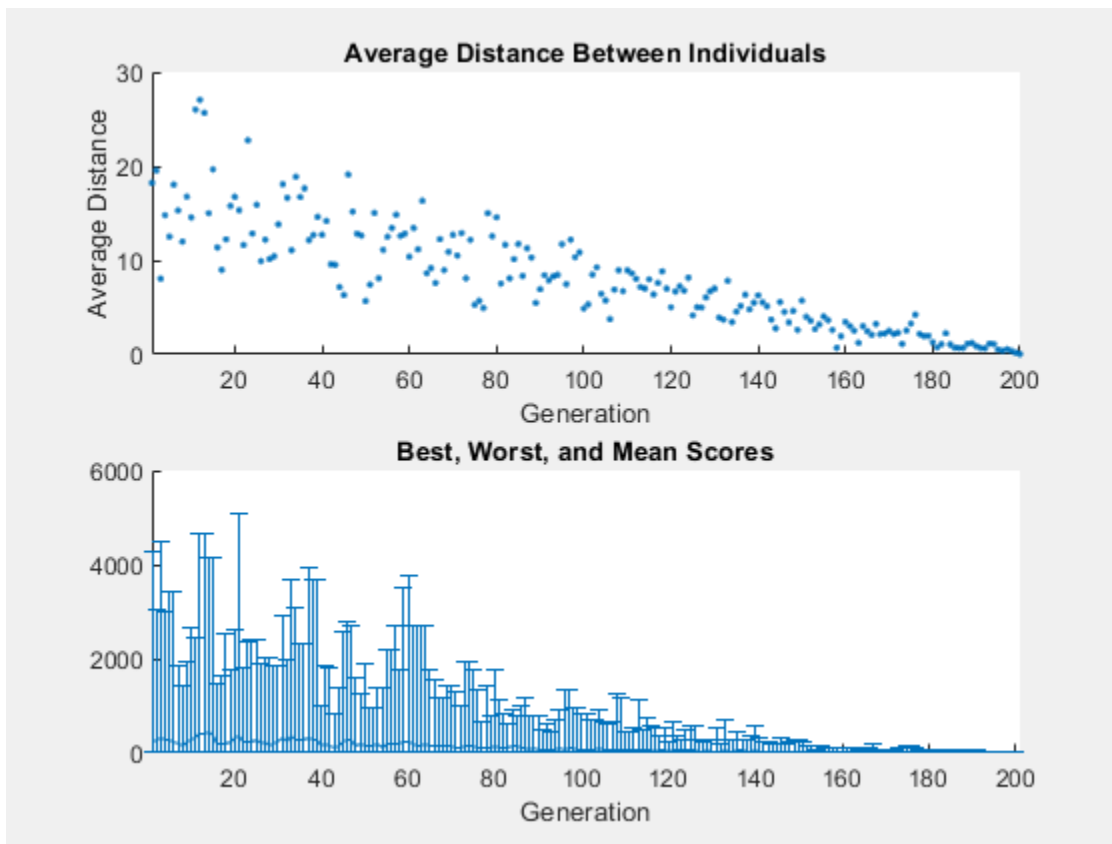
`Scale` and `Shrink` are scalars with default values 1 each.

- `Scale` controls the standard deviation of the mutation at the first generation. This value is `Scale` multiplied by the range of the initial population, which you specify by the `InitialPopulationRange` option.
- `Shrink` controls the rate at which the average amount of mutation decreases. The standard deviation decreases linearly so that its final value equals $1 - \text{Shrink}$ times its initial value at the first generation. For example, if `Shrink` has the default value of 1, then the amount of mutation decreases to 0 at the final step.

You can see the effect of mutation by selecting the plot functions `@gaplotdistance` and `@gaplotrange`, and then running the genetic algorithm on a problem such as the one described in “Minimize Rastrigin's Function” on page 8-4. The following figure shows the plot after setting the random number generator.

```
rng default % For reproducibility
options = optimoptions('ga','PlotFcn',{@gaplotdistance,@gaplotrange},...
    'MaxStallGenerations',200); % to get a long run
[x,fval] = ga(@rastriginsfcn,2,[],[],[],[],[],[],[],[],options);
```

Optimization terminated: maximum number of generations exceeded.

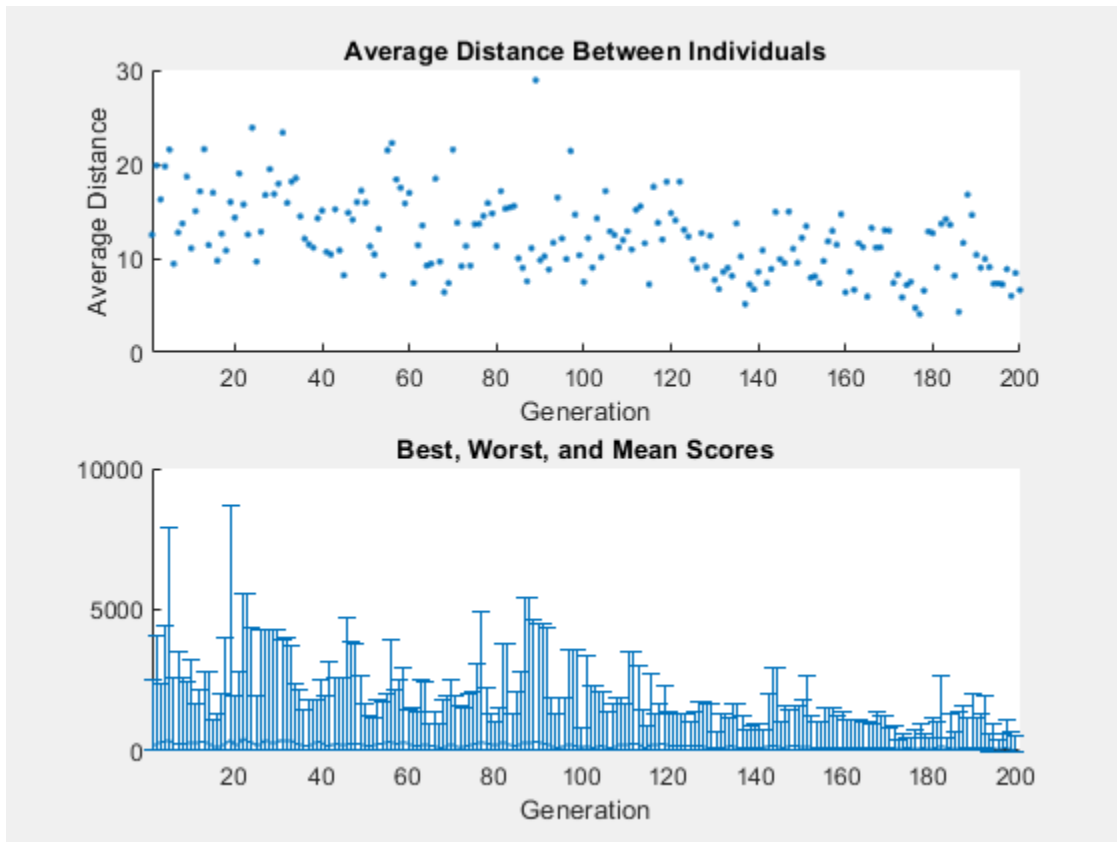


The upper plot displays the average distance between points in each generation. As the amount of mutation decreases, so does the average distance between individuals, which is approximately 0 at the final generation. The lower plot displays a vertical line at each generation, showing the range from the smallest to the largest fitness value, as well as mean fitness value. As the amount of mutation decreases, so does the range. These plots show that reducing the amount of mutation decreases the diversity of subsequent generations.

For comparison, the following figure shows the same plots when you set `Shrink` to 0.5.

```
options = optimoptions('ga',options,...
    'MutationFcn',{@mutationgaussian,1,.5});
[x,fval] = ga(@rastriginsfcn,2,[],[],[],[],[],[],[],[],options);
```

Optimization terminated: maximum number of generations exceeded.



This time, the average amount of mutation decreases by a factor of 1/2 by the final generation. As a result, the average distance between individuals decreases less than before.

Setting the Crossover Fraction

The `CrossoverFraction` option specifies the fraction of each population, other than elite children, that are made up of crossover children. A crossover fraction of 1 means that all children other than elite individuals are crossover children, while a crossover fraction of 0 means that all children are mutation children. The following example shows that neither of these extremes is an effective strategy for optimizing a function.

The example uses the fitness function whose value at a point is the sum of the absolute values of the coordinates at the points. That is,

$$f(x_1, x_2, \dots, x_n) = |x_1| + |x_2| + \dots + |x_n|.$$

You can define this function as an anonymous function by setting the fitness function to

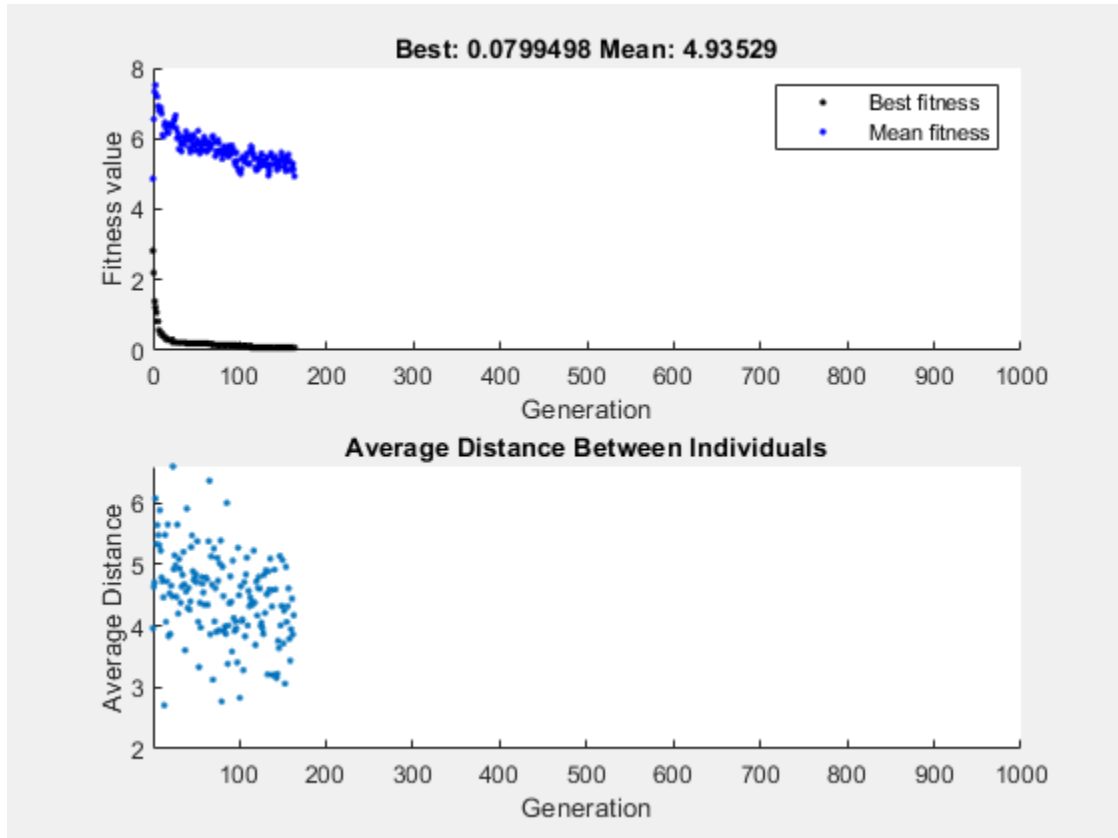
```
fun = @(x) sum(abs(x));
```

Run the example with the default value of 0.8 as the `CrossoverFraction` option.

```
nvar = 10;
options = optimoptions('ga',...
    'InitialPopulationRange', [-1;1],...
    'PlotFcn',{@gaplotbestf,@gaplotdistance});
```

```
rng(14, 'twister') % For reproducibility
[x, fval] = ga(fun, nvar, [], [], [], [], [], [], [], [], options)
```

Optimization terminated: average change in the fitness value less than options.FunctionTolerance



```
x = 1x10
```

```
-0.0020 -0.0134 -0.0067 -0.0028 -0.0241 -0.0118 0.0021 0.0113 -0.0021 -0.0
```

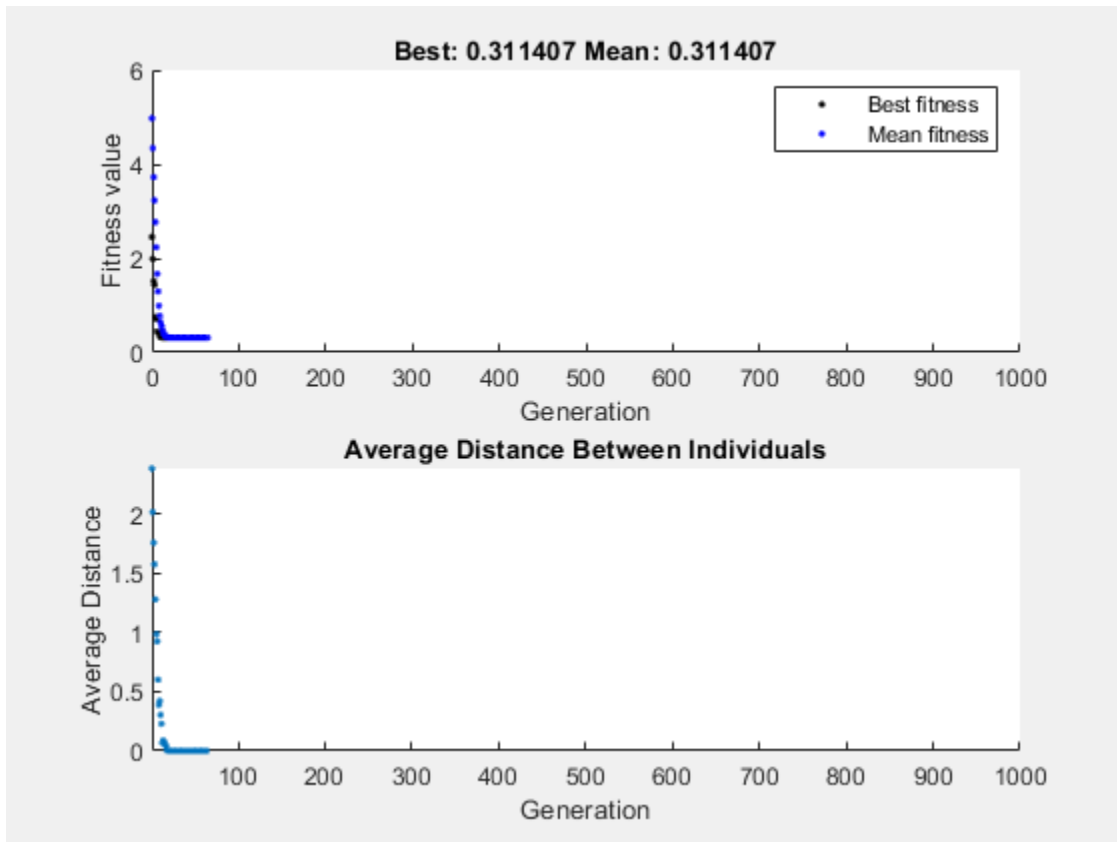
```
fval = 0.0799
```

Crossover Without Mutation

To see how the genetic algorithm performs when there is no mutation, set the `CrossoverFraction` option to `1.0` and rerun the solver.

```
options.CrossoverFraction = 1;
[x, fval] = ga(fun, nvar, [], [], [], [], [], [], [], [], options)
```

Optimization terminated: average change in the fitness value less than options.FunctionTolerance



```
x = 1x10
```

```
-0.0275 -0.0043 0.0372 -0.0118 -0.0377 -0.0444 -0.0258 -0.0520 0.0174 0.0
```

```
fval = 0.3114
```

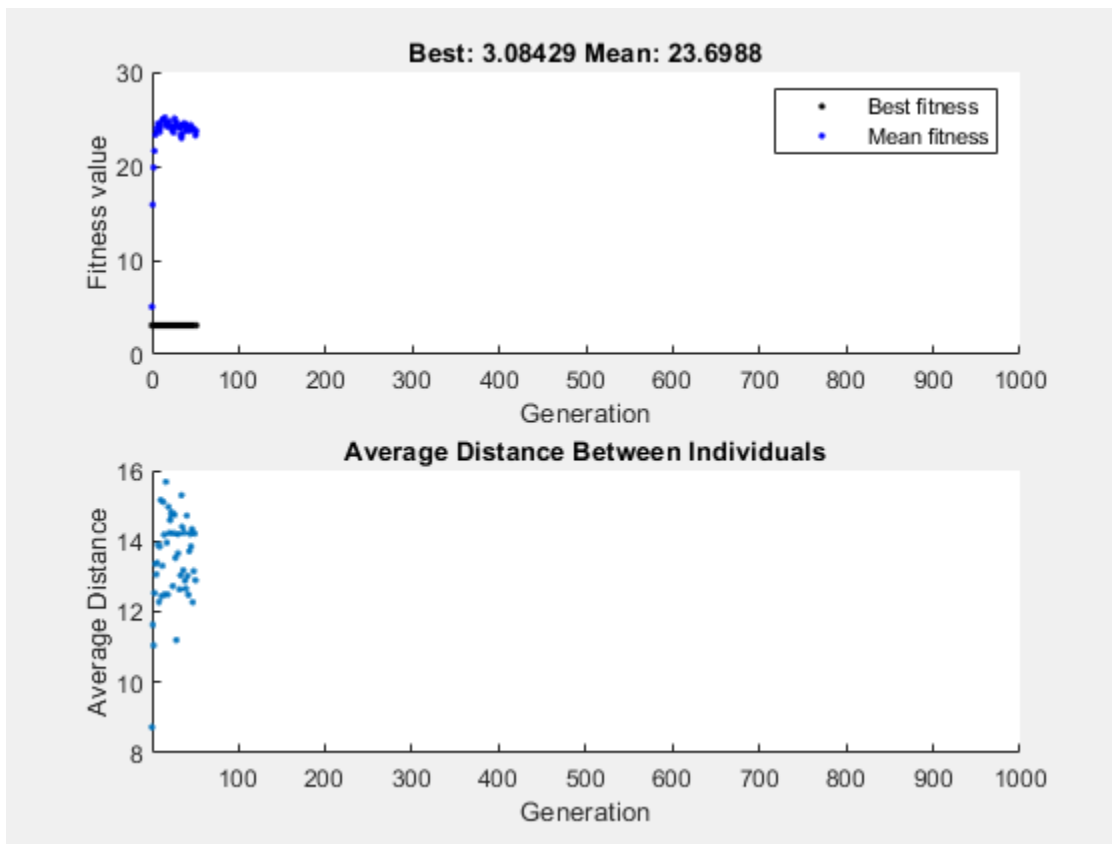
In this case, the algorithm selects genes from the individuals in the initial population and recombines them. The algorithm cannot create any new genes because there is no mutation. The algorithm generates the best individual that it can using these genes at generation number 8, where the best fitness plot becomes level. After this, it creates new copies of the best individual, which are then are selected for the next generation. By generation number 17, all individuals in the population are the same, namely, the best individual. When this occurs, the average distance between individuals is 0. Since the algorithm cannot improve the best fitness value after generation 8, it stalls after 50 more generations, because Stall generations is set to 50.

Mutation Without Crossover

To see how the genetic algorithm performs when there is no crossover, set the `CrossoverFraction` option to 0.

```
options.CrossoverFraction = 0;
[x,fval] = ga(fun,nvar,[],[],[],[],[],[],[],options)
```

```
Optimization terminated: average change in the fitness value less than options.FunctionTolerance
```



```
x = 1x10
```

```
0.4014 0.0538 0.7824 0.1930 0.0513 -0.4801 0.9988 -0.0059 0.0875 0.0
```

```
fval = 3.0843
```

In this case, the random changes that the algorithm applies never improve the fitness value of the best individual at the first generation. While it improves the individual genes of other individuals, as you can see in the upper plot by the decrease in the mean value of the fitness function, these improved genes are never combined with the genes of the best individual because there is no crossover. As a result, the best fitness plot is level and the algorithm stalls at generation number 50.

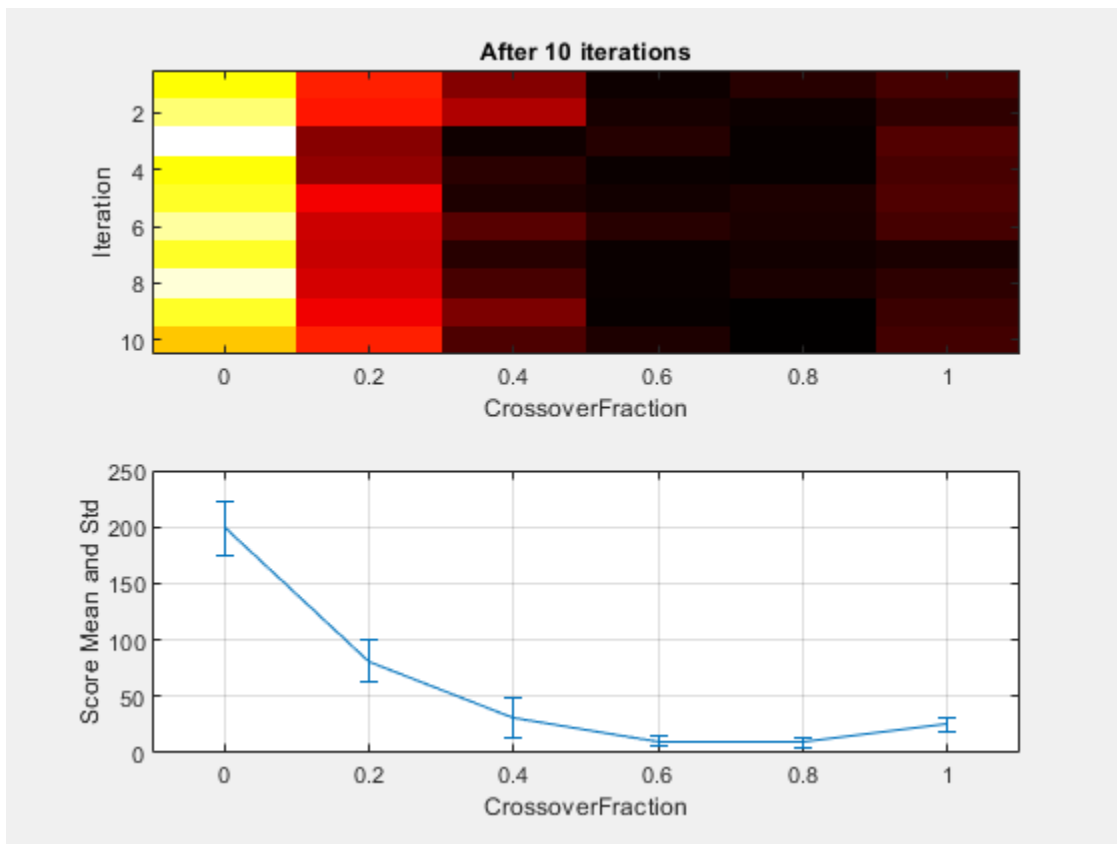
Comparing Results for Varying Crossover Fractions

The example `deterministicstudy.m`, which is included when you run this example, compares the results of applying the genetic algorithm to Rastrigin's function with the `CrossoverFraction` option set to 0, 0.2, 0.4, 0.6, 0.8, and 1. The example runs for 10 generations. At each generation, the example plots the means and standard deviations of the best fitness values in all the preceding generations, for each value of the `CrossoverFraction` option.

Run the example.

```
deterministicstudy
```

```
Optimization terminated: average change in the fitness value less than options.FunctionTolerance
Optimization terminated: average change in the fitness value less than options.FunctionTolerance
```

The lower plot shows the means and standard deviations of the best fitness values over 10 generations, for each of the values of the crossover fraction. The upper plot shows a color-coded display of the best fitness values in each generation.

For this fitness function, setting Crossover fraction to 0.8 yields the best result. However, for another fitness function, a different setting for Crossover fraction might yield the best result.

See Also

ga

More About

- “How the Genetic Algorithm Works” on page 8-15
- “Custom Output Function for Genetic Algorithm” on page 8-105

Global vs. Local Optimization Using ga

Searching for a Global Minimum

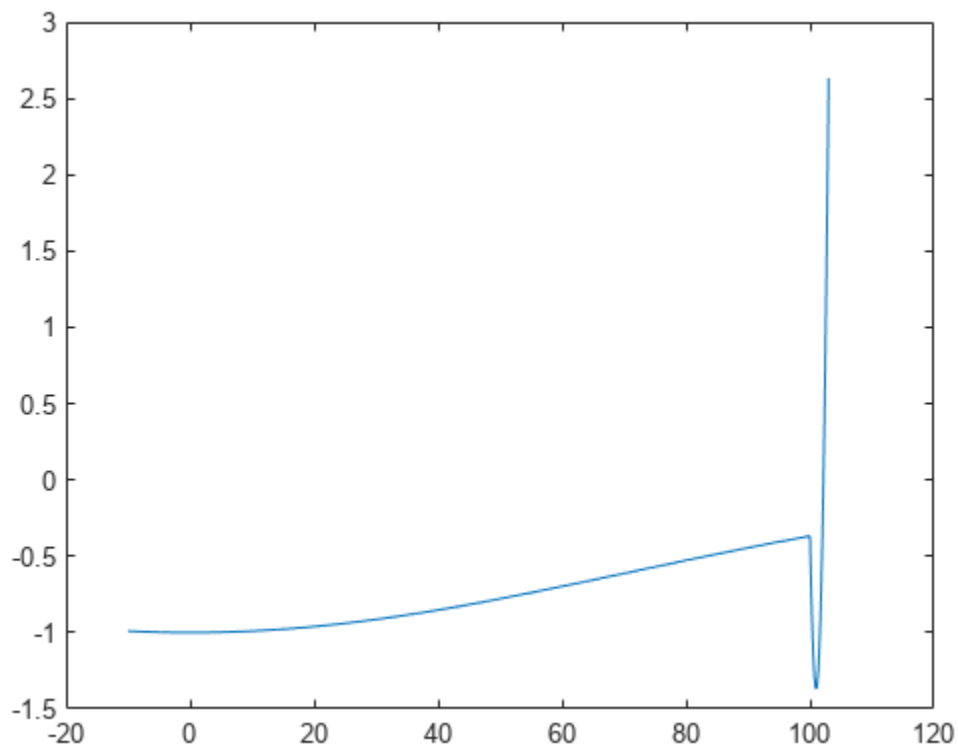
Sometimes the goal of an optimization is to find the global minimum or maximum of a function—a point where the function value is smaller or larger than at any other point in the search space. However, optimization algorithms sometimes return a local minimum—a point where the function value is smaller than at nearby points, but possibly greater than at a distant point in the search space. The genetic algorithm can sometimes overcome this deficiency with the right settings.

As an example, consider the following function.

$$f(x) = \begin{cases} -\exp\left(-\left(\frac{x}{100}\right)^2\right) & \text{for } x \leq 100, \\ -\exp(-1) + (x - 100)(x - 102) & \text{for } x > 100. \end{cases}$$

Plot the function.

```
t = -10:.1:103;
for ii = 1:length(t)
    y(ii) = two_min(t(ii));
end
plot(t,y)
```



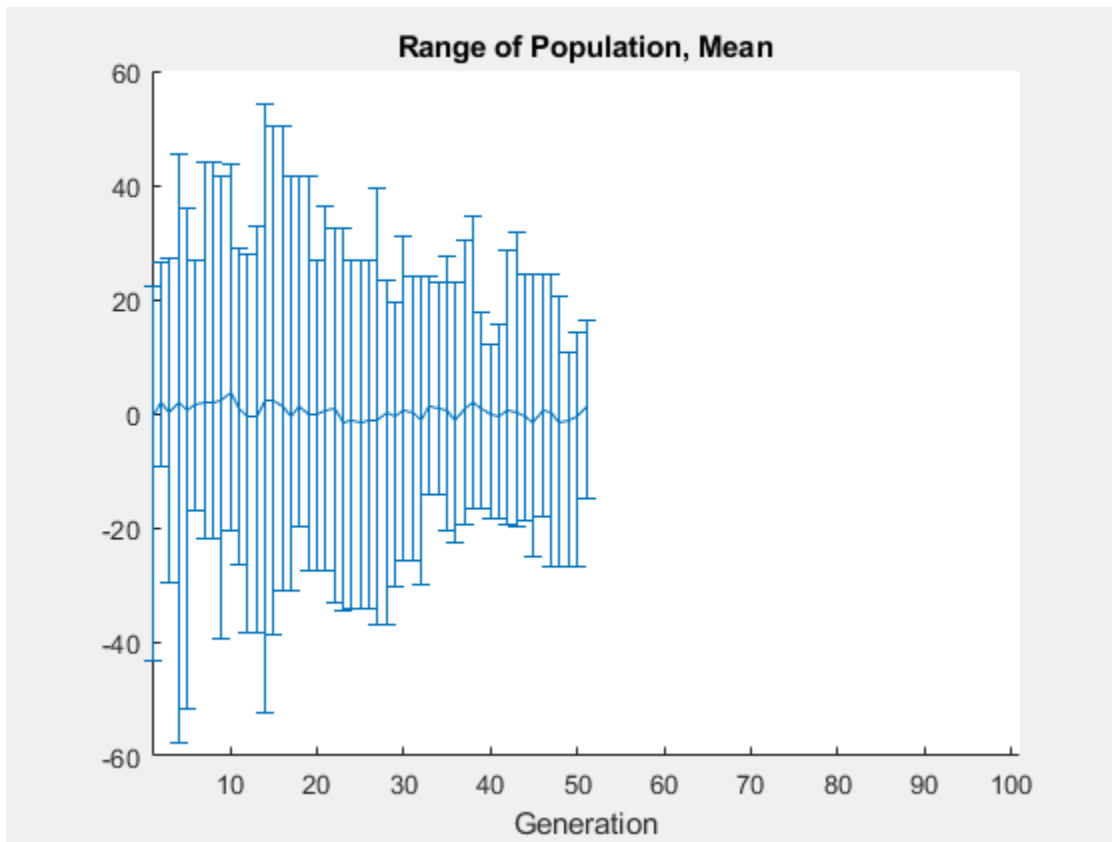
The function has two local minima, one at $x = 0$, where the function value is -1 , and the other at $x = 101$, where the function value is $-1 - 1/e$. Since the latter value is smaller, the global minimum occurs at $x = 101$.

Run ga Using Default Parameters

The code for the `two_min` helper function is at the end of this example on page 8-93. Run `ga` with default parameters to minimize the `two_min` function. Use the `gaplot1drange` helper function (included at the end of this example on page 8-94) to plot the range of the `ga` population at each iteration.

```
rng default % For reproducibility
options = optimoptions('ga','PlotFcn',@gaplot1drange);
[x,fval] = ga(@two_min,1,[],[],[],[],[],[],[],options)
```

Optimization terminated: average change in the fitness value less than options.FunctionTolerance



```
x = -0.0688
```

```
fval = -1.0000
```

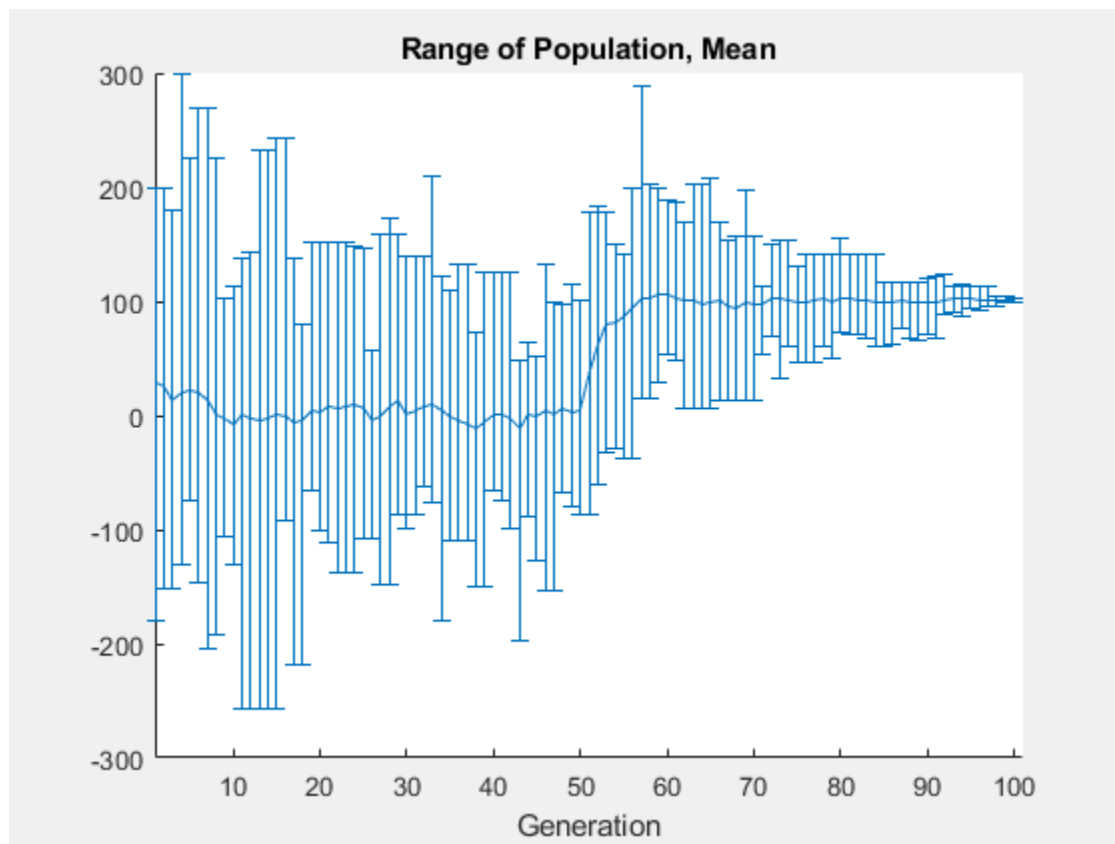
The genetic algorithm returns a point very close to the local minimum at $x = 0$. Note that all individuals lie between -60 and 60 . The population never explores points near the global minimum at $x = 101$.

Increase Initial Range

One way to make the genetic algorithm explore a wider range of points—that is, to increase the diversity of the populations—is to increase the initial range. The initial range does not have to include the point $x = 101$, but it must be large enough so that the algorithm generates individuals near $x = 101$. Set the `InitialPopulationRange` option to `[-10;90]` and rerun the solver.

```
options.InitialPopulationRange = [-10;90];
[x,fval] = ga(@two_min,1,[],[],[],[],[],[],[],[],options)
```

Optimization terminated: maximum number of generations exceeded.



```
x = 100.9783
```

```
fval = -1.3674
```

This time, the custom plot shows a much wider range of individuals. There are individuals near 101 from early on, and the population mean begins to converge to 101.

Helper Functions

This code creates the `two_min` helper function.

```
function y = two_min(x)
if x <= 100
    y = -exp(-(x/100)^2);
else
```

```

    y = -exp(-1) + (x-100)*(x-102);
end
end

```

This code creates the `gaplotldrange` helper function.

```

function state = gaplotldrange(options,state,flag)
%gaplotldrange Plots the mean and the range of the population.
% STATE = gaplotldrange(OPTIONS,STATE,FLAG) plots the mean and the range
% (highest and the lowest) of individuals (1-D only).
%
% Example:
% Create options that use gaplotldrange
% as the plot function
% options = optimoptions('ga','PlotFcn',@gaplotldrange);
%
% Copyright 2012-2014 The MathWorks, Inc.

if isinf(options.MaxGenerations) || size(state.Population,2) > 1
    title('Plot Not Available','interp','none');
    return;
end
generation = state.Generation;
score = state.Population;
smean = mean(score);
Y = smean;
L = smean - min(score);
U = max(score) - smean;

switch flag

    case 'init'
        set(gca,'xlim',[1,options.MaxGenerations+1]);
        plotRange = errorbar(generation,Y,L,U);
        set(plotRange,'Tag','gaplotldrange');
        title('Range of Population, Mean','interp','none')
        xlabel('Generation','interp','none')
    case 'iter'
        plotRange = findobj(get(gca,'Children'),'Tag','gaplotldrange');
        newX = [get(plotRange,'Xdata') generation];
        newY = [get(plotRange,'Ydata') Y];
        newL = [get(plotRange,'Ldata') L];
        newU = [get(plotRange,'Udata') U];
        set(plotRange,'Xdata',newX,'Ydata',newY,'Ldata',newL,'Udata',newU);
end
end

```

See Also

More About

- “What Is Global Optimization?” on page 1-24
- “Population Diversity” on page 8-72
- “Isolated Global Minimum” on page 4-86

Hybrid Scheme in the Genetic Algorithm

This example shows how to use a hybrid scheme to optimize a function using the genetic algorithm and another optimization method. `ga` can quickly reach a neighborhood of a local minimum, but it can require many function evaluations to achieve convergence. To speed the solution process, first run `ga` for a small number of generations to approach an optimum point. Then use the solution from `ga` as the initial point for another optimization solver to perform a faster and more efficient local search.

Rosenbrock's Function

This example uses Rosenbrock's function (also known as DeJong's second function) as the fitness function:

$$f(x) = 100(x(2) - x(1)^2)^2 + (1 - x(1))^2.$$

Rosenbrock's function is notorious in optimization because of the slow convergence most methods exhibit when trying to minimize this function. Rosenbrock's function has a unique minimum at the point $x^* = (1,1)$, where it has a function value $f(x^*) = 0$.

The code for Rosenbrock's function is in the `dejong2fcn` file.

```
type dejong2fcn.m

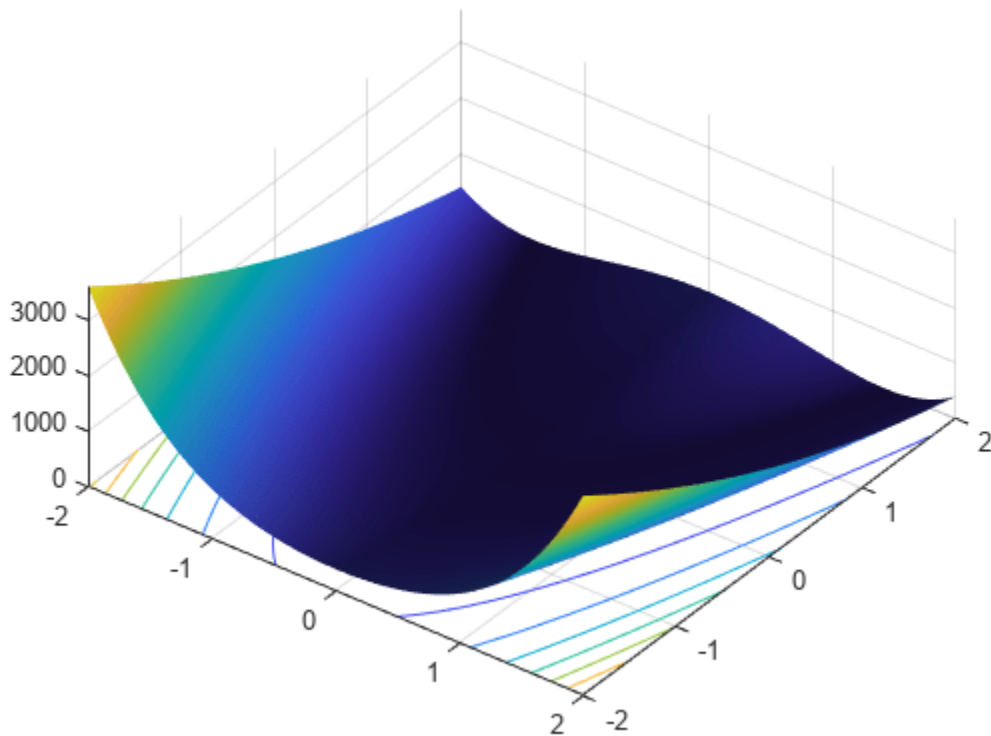
function scores = dejong2fcn(pop)
%DEJONG2FCN Compute DeJongs second function.
%This function is also known as Rosenbrock's function

% Copyright 2003-2004 The MathWorks, Inc.

scores = zeros(size(pop,1),1);
for i = 1:size(pop,1)
    p = pop(i,:);
    scores(i) = 100 * (p(1)^2 - p(2)) ^2 + (1 - p(1))^2;
end
```

Plot Rosenbrock's function over the range $-2 \leq x(1) \leq 2$; $-2 \leq x(2) \leq 2$.

```
plotobjective(@dejong2fcn, [-2 2; -2 2]);
```



Genetic Algorithm Solution

First, use `ga` alone to find the minimum of Rosenbrock's function.

```
FitnessFcn = @dejong2fcn;
numberOfVariables = 2;
```

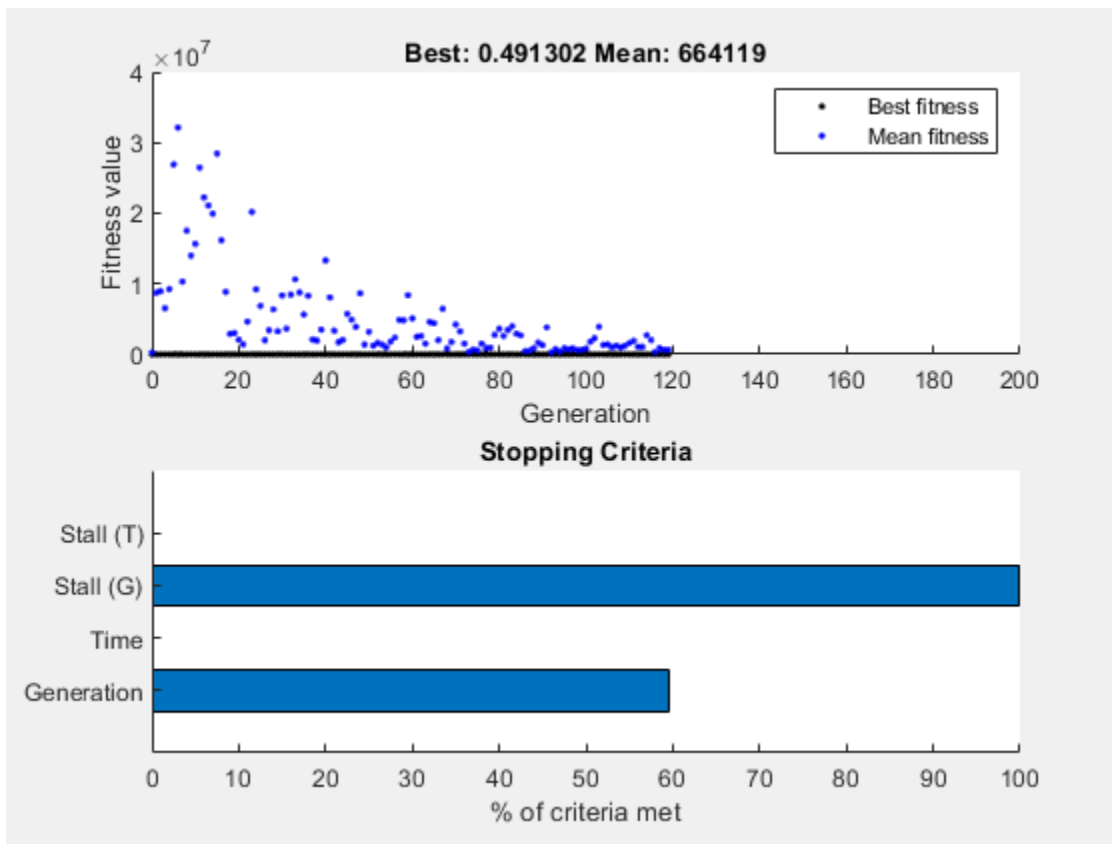
Include plot functions to monitor the optimization process.

```
options = optimoptions(@ga, 'PlotFcn', {@gaplotbestf, @gaplotstopping});
```

Set the random number stream for reproducibility, and run `ga` using the options.

```
rng default
[x, fval] = ga(FitnessFcn, numberOfVariables, [], [], [], [], [], [], [], options)
```

Optimization terminated: average change in the fitness value less than options.FunctionTolerance



```
x = 1x2
```

```
    0.3454    0.1444
```

```
fval = 0.4913
```

Using the default stopping criteria, `ga` does not provide a very accurate solution. You can change the stopping criteria to try to find a more accurate solution, but `ga` requires many function evaluations to approach the global optimum $x^* = (1,1)$.

Instead, perform a more efficient local search that starts where `ga` stops by using the hybrid function option in `ga`.

Adding a Hybrid Function

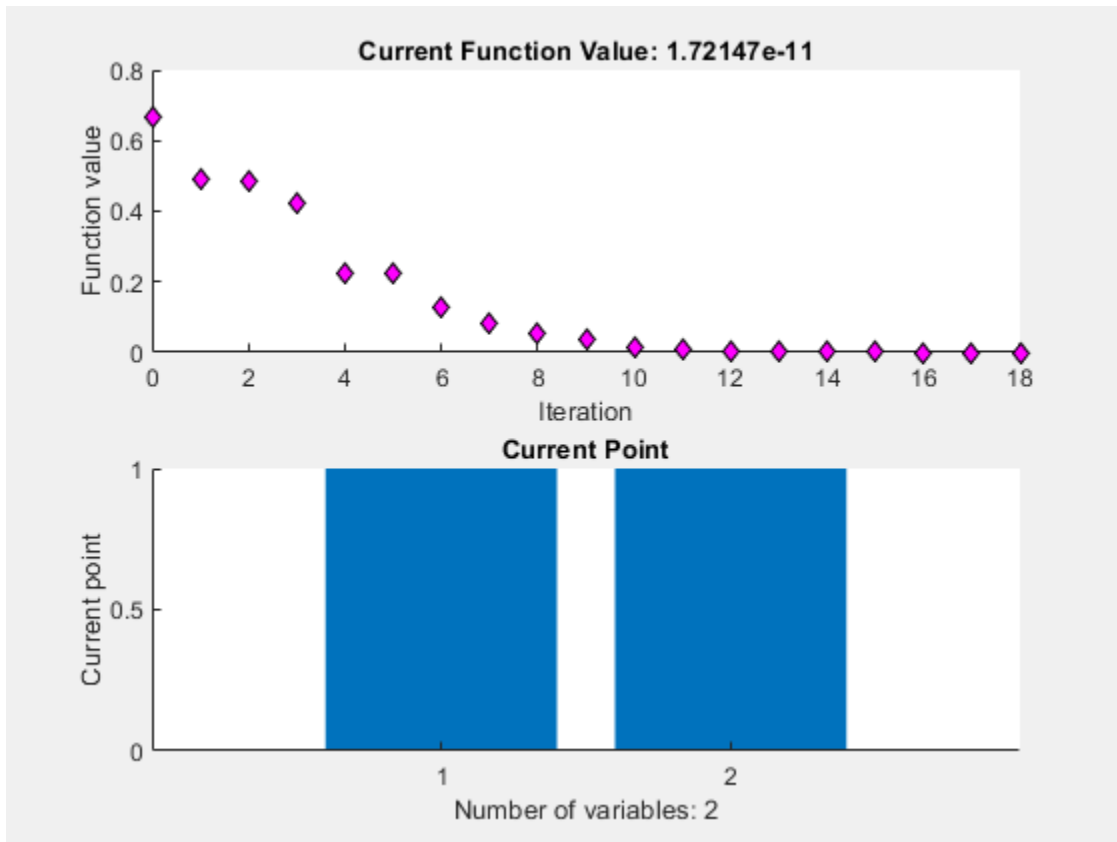
A hybrid function begins from the point where `ga` stops. Hybrid function choices are `fminsearch`, `patternsearch`, or `fminunc`. Because this optimization example is smooth and unconstrained, use `fminunc` as the hybrid function. Provide `fminunc` with plot options as an additional argument when specifying the hybrid function.

```
fminuncOptions = optimoptions(@fminunc,'PlotFcn',{'optimplotfval','optimplotx'});
options = optimoptions(options,'HybridFcn',{@fminunc, fminuncOptions});
```

Run `ga` again with `fminunc` as the hybrid function.

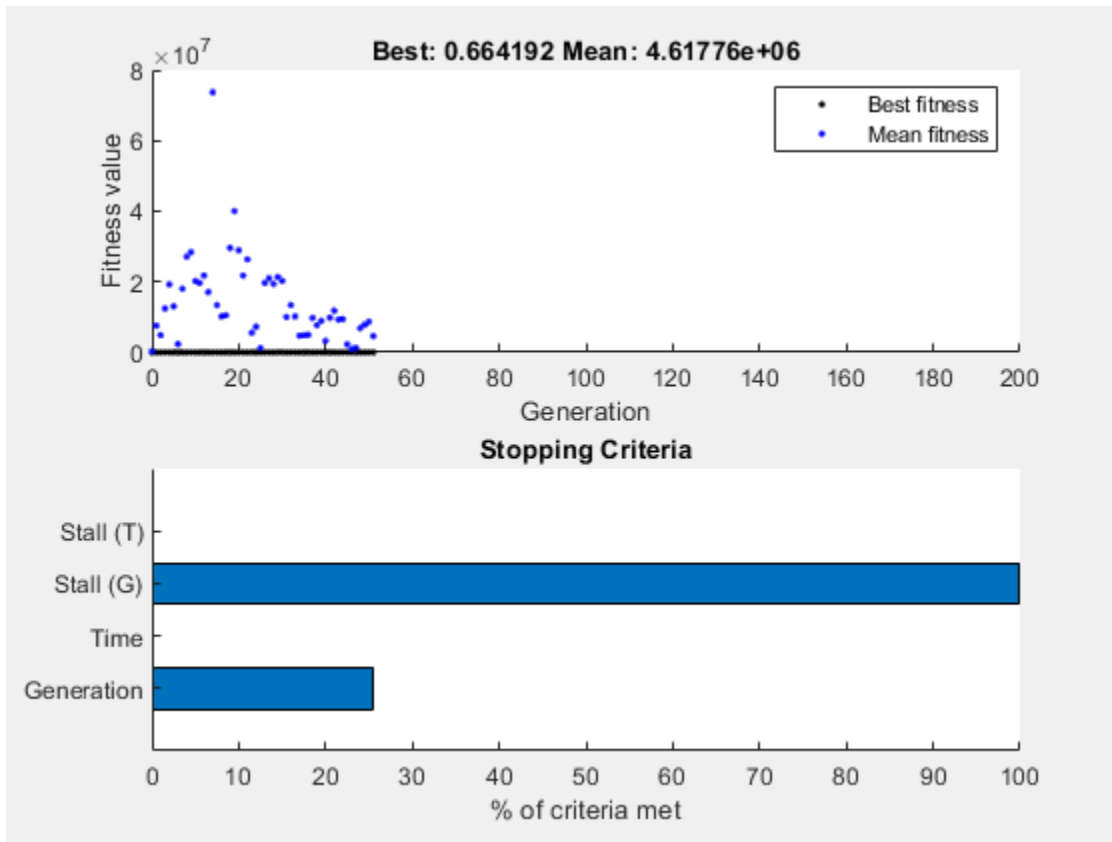
```
[x,fval,exitflag,output] = ga(FitnessFcn,numberOfVariables,[],[],[],[],[],[],[],options)
```

Optimization terminated: average change in the fitness value less than options.FunctionTolerance



Local minimum found.

Optimization completed because the size of the gradient is less than the value of the optimality tolerance.



```
x = 1x2
```

```
1.0000 1.0000
```

```
fval = 1.7215e-11
```

```
exitflag = 1
```

```
output = struct with fields:
```

```
  problemtype: 'unconstrained'
```

```
  rngstate: [1x1 struct]
```

```
  generations: 51
```

```
  funccount: 2534
```

```
  message: 'Optimization terminated: average change in the fitness value less than option
```

```
  maxconstraint: []
```

```
  hybridflag: 1
```

The `ga` plot shows the best and mean values of the population in every generation. The plot title identifies the best value found by `ga` when it stops. The hybrid function `fminunc` starts from the best point found by `ga`. The `fminunc` plot shows the solution `x` and `fval`, which result from using `ga` and `fminunc` together. In this case, using a hybrid function improves the accuracy and efficiency of the

solution. The output `.hybridflag` field shows that `fminunc` stops with exit flag 1, indicating that `x` is a true local minimum.

See Also

More About

- “Options and Outputs” on page 8-64
- “Global vs. Local Optimization Using `ga`” on page 8-91

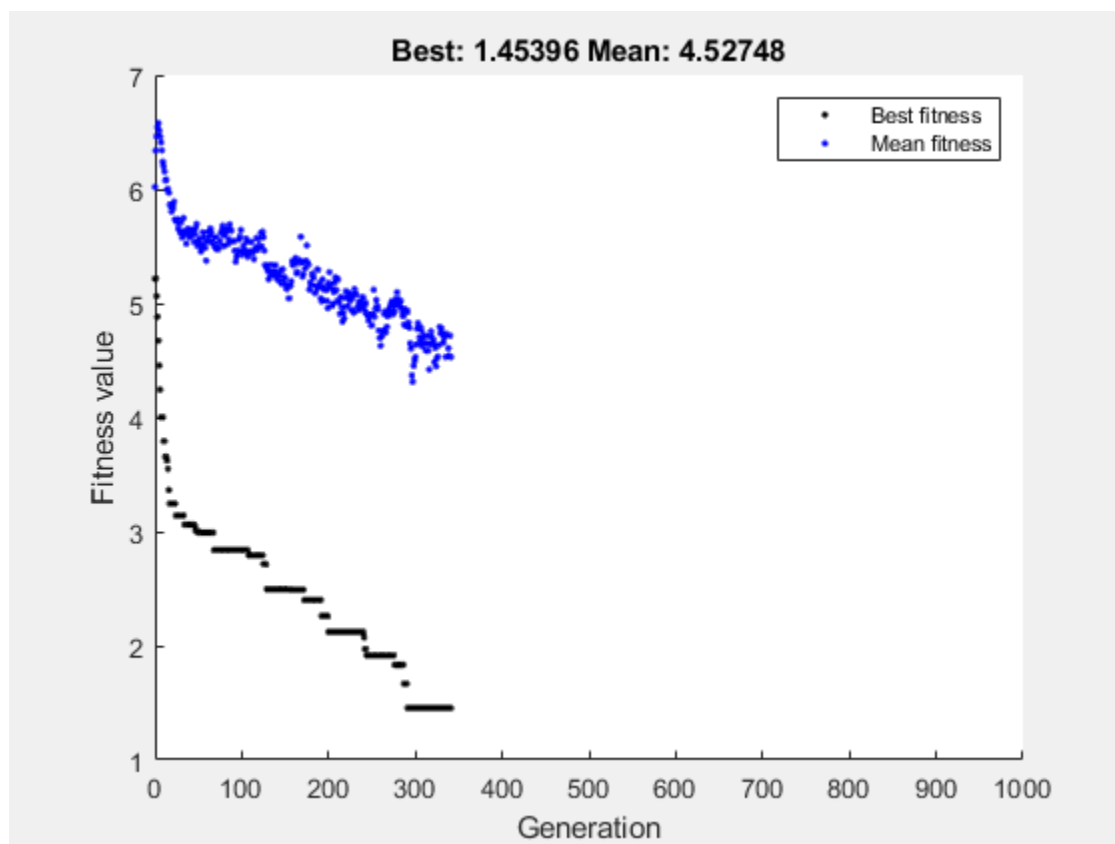
Set Maximum Number of Generations and Stall Generations

The `MaxGenerations` option determines the maximum number of generations the genetic algorithm takes; see “Stopping Conditions for the Algorithm” on page 8-18. Increasing `MaxGenerations` can improve the final result. The related `MaxStallGenerations` option controls the number of steps `ga` looks over to see whether it is making progress. Increasing `MaxStallGenerations` can enable `ga` to continue when the algorithm needs more function evaluations to find a better solution.

For example, optimize `rastriginsfcn` using 10 variables with default parameters. To observe the solver's progress as it approaches the minimum value of 0, optimize the logarithm of the function.

```
rng default % For reproducibility
fun = @(x)log(rastriginsfcn(x));
nvar = 10;
options = optimoptions('ga','PlotFcn','gaplotbestf');
[x,fval] = ga(fun,nvar,[],[],[],[],[],[],[],[],options)
```

Optimization terminated: average change in the fitness value less than options.FunctionTolerance



```
x = 1×10
```

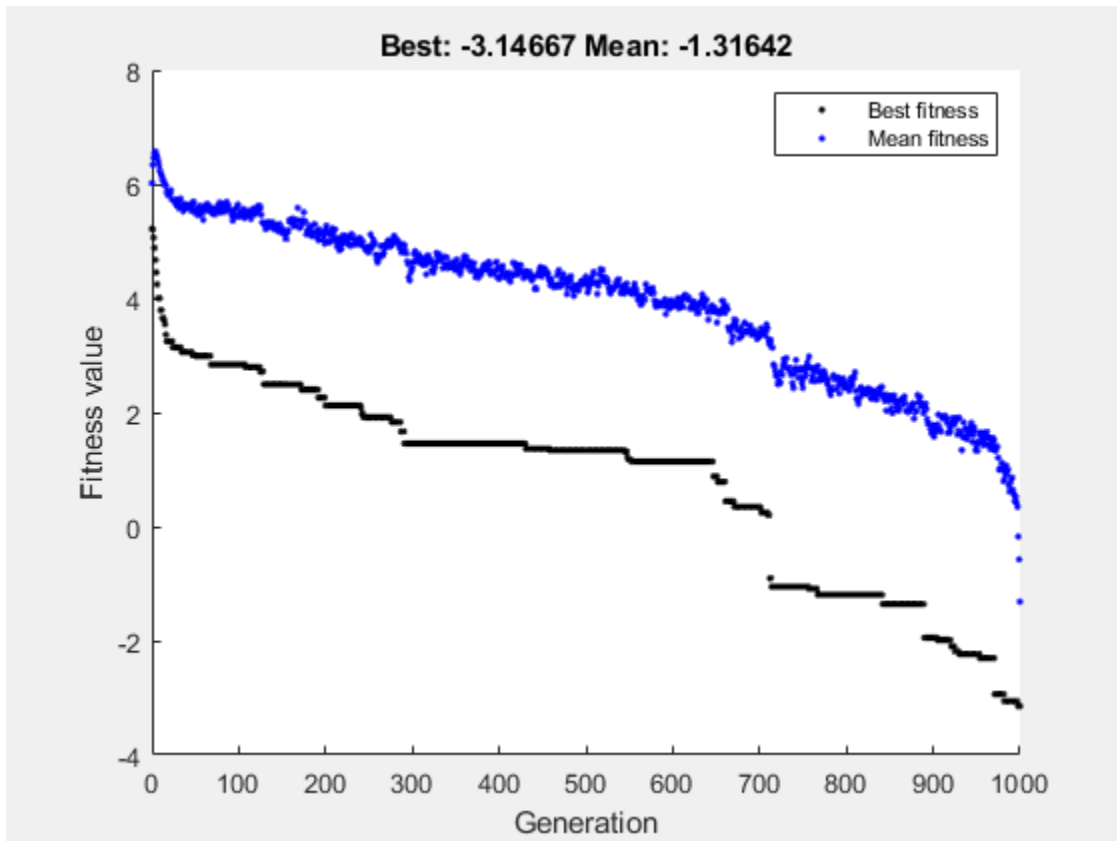
```
-0.0495 -0.0670 -0.0485 0.0174 -0.0087 0.0275 -0.0383 0.0620 -1.0047 -0.0
```

```
fval = 1.4540
```

As `ga` approaches the optimal point at the origin, it stalls. To obtain a better solution, set the stall generation limit to 500 and the generation limit to 1000.

```
options = optimoptions(options, 'MaxStallGenerations', 500, 'MaxGenerations', 1000);
rng default % For reproducibility
[x, fval] = ga(fun, nvar, [], [], [], [], [], [], [], options)
```

Optimization terminated: maximum number of generations exceeded.



```
x = 1x10
```

```
    0.0025    -0.0039    0.0021    -0.0030    -0.0053    0.0033    0.0080    0.0012    0.0006    0.
```

```
fval = -3.1467
```

This time the solver approaches the true minimum much more closely.

See Also

More About

- “Options and Outputs” on page 8-64
- “Population Diversity” on page 8-72

Vectorize the Fitness Function

In this section...

“Vectorize for Speed” on page 8-103

“Vectorized Constraints” on page 8-104

Vectorize for Speed

The genetic algorithm usually runs faster if you *vectorize* the fitness function. This means that the genetic algorithm only calls the fitness function once, but expects the fitness function to compute the fitness for all individuals in the current population at once. To vectorize the fitness function,

- Write the file that computes the function so that it accepts a matrix with arbitrarily many rows, corresponding to the individuals in the population. For example, to vectorize the function

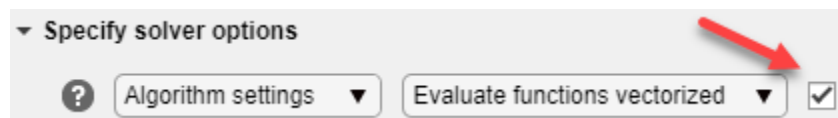
$$f(x_1, x_2) = x_1^2 - 2x_1x_2 + 6x_1 + x_2^2 - 6x_2$$

write the file using the following code:

```
z = x(:,1).^2 - 2*x(:,1).*x(:,2) + 6*x(:,1) + x(:,2).^2 - 6*x(:,2);
```

The colon in the first entry of x indicates all the rows of x , so that $x(:, 1)$ is a vector. The \wedge and \cdot operators perform elementwise operations on the vectors.

- At the command line, set the UseVectorized option to true using optimoptions.
- In the **Optimize** Live Editor task, ensure that the **Algorithm settings > Evaluate functions vectorized** setting has a check mark.



Note The fitness function, and any nonlinear constraint function, must accept an arbitrary number of rows to use the **Vectorize** option. ga sometimes evaluates a single row even during a vectorized calculation.

The following comparison, run at the command line, shows the improvement in speed with vectorization. The function `fun` is a vectorized version of Rastrigin's function; see “Minimize Rastrigin's Function” on page 8-4.

```
fun = @(pop)10.0 * size(pop,2) + sum(pop.^2 - 10.0*cos(2*pi.*pop),2);
options = optimoptions('ga','PopulationSize',2000);
tic
ga(fun,20,[],[],[],[],[],[],[],options);
toc
```

```
Optimization terminated: maximum number of generations exceeded.
Elapsed time is 2.511456 seconds.
```

```
options = optimoptions(options,'UseVectorized',true);
tic;
```

```
ga(fun,20,[],[],[],[],[],[],[],options);  
toc
```

```
Optimization terminated: maximum number of generations exceeded.  
Elapsed time is 1.451496 seconds.
```

Vectorized Constraints

If there are nonlinear constraints, the objective function and the nonlinear constraints all need to be vectorized in order for the algorithm to compute in a vectorized manner.

“Vectorize the Objective and Constraint Functions” on page 6-83 contains an example of how to vectorize both for the solver `patternsearch`. The syntax is nearly identical for `ga`. The only difference is that `patternsearch` can have its patterns appear as either row or column vectors; the corresponding vectors for `ga` are the population vectors, which are always rows.

See Also

More About

- “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11
- “Compute Objective Functions” on page 2-2

Custom Output Function for Genetic Algorithm

This example shows the use of a custom output function in the genetic algorithm solver `ga`. The custom output function performs the following tasks:

- Plot the range of the first two components of the population as a rectangle. The left and lower sides of the rectangle are at the minima of $x(1)$ and $x(2)$ respectively, and the right and upper sides are at the respective maxima.
- Halt the iterations when the best function value drops below 0.1 (the minimum value of the objective function is 0).
- Record the entire population in a variable named `gapopulationhistory` in your MATLAB® workspace every 10 generations.
- Modify the initial crossover fraction to the custom value 0.2 , and then update it back to the default 0.8 after 25 generations. The initial setting of 0.2 causes the first several iterations to search primarily at random via mutation. The later setting of 0.8 causes the following iterations to search primarily via combinations of existing population members.

Objective Function

The objective function is for four-dimensional x whose first two components are integer-valued.

```
function f = gaintobj(x)
f = rastriginsfcn([x(1)-6 x(2)-13]);
f = f + rastriginsfcn([x(3)-3*pi x(4)-5*pi]);
```

Output Function

The custom output function sets up the plot during initialization, and maintains the plot during iterations. The output function also pauses the iterations for 0.1 s so you can see the plot as it develops.

```
function [state,options,optchanged] = gaoutfun(options,state,flag)
persistent h1 history r
optchanged = false;
switch flag
case 'init'
    h1 = figure;
    ax = gca;
    ax.XLim = [0 21];
    ax.YLim = [0 21];
    l1 = min(state.Population(:,1));
    m1 = max(state.Population(:,1));
    l2 = min(state.Population(:,2));
    m2 = max(state.Population(:,2));
    r = rectangle(ax,'Position',[l1 l2 m1-l1 m2-l2]);
    history(:, :, 1) = state.Population;
    assignin('base','gapopulationhistory',history);
case 'iter'
    % Update the history every 10 generations.
    if rem(state.Generation,10) == 0
        ss = size(history,3);
```

```

        history(:,:,ss+1) = state.Population;
        assignin('base','gapopulationhistory',history);
    end
    % Find the best objective function, and stop if it is low.
    ibest = state.Best(end);
    ibest = find(state.Score == ibest,1,'last');
    bestx = state.Population(ibest,:);
    bestf = gaintobj(bestx);
    if bestf <= 0.1
        state.StopFlag = 'y';
        disp('Got below 0.1')
    end
    % Update the plot.
    figure(h1)
    l1 = min(state.Population(:,1));
    m1 = max(state.Population(:,1));
    l2 = min(state.Population(:,2));
    m2 = max(state.Population(:,2));
    r.Position = [l1 l2 m1-l1 m2-l2];
    pause(0.1)
    % Update the fraction of mutation and crossover after 25 generations.
    if state.Generation == 25
        options.CrossoverFraction = 0.8;
        optchanged = true;
    end
case 'done'
    % Include the final population in the history.
    ss = size(history,3);
    history(:,:,ss+1) = state.Population;
    assignin('base','gapopulationhistory',history);
end

```

Problem Setup and Solution

Set the lower and upper bounds.

```
lb = [1 1 -30 -30];
ub = [20 20 70 70];
```

Set the integer variables and number of variables.

```
intcon = [1 2];
nvar = 4;
```

Set options to call the custom output function, and to initially have little crossover.

```
options = optimoptions('ga','OutputFcn',@gaoutfun,'CrossoverFraction',0.2);
```

For reproducibility, set the random number generator.

```
rng default
```

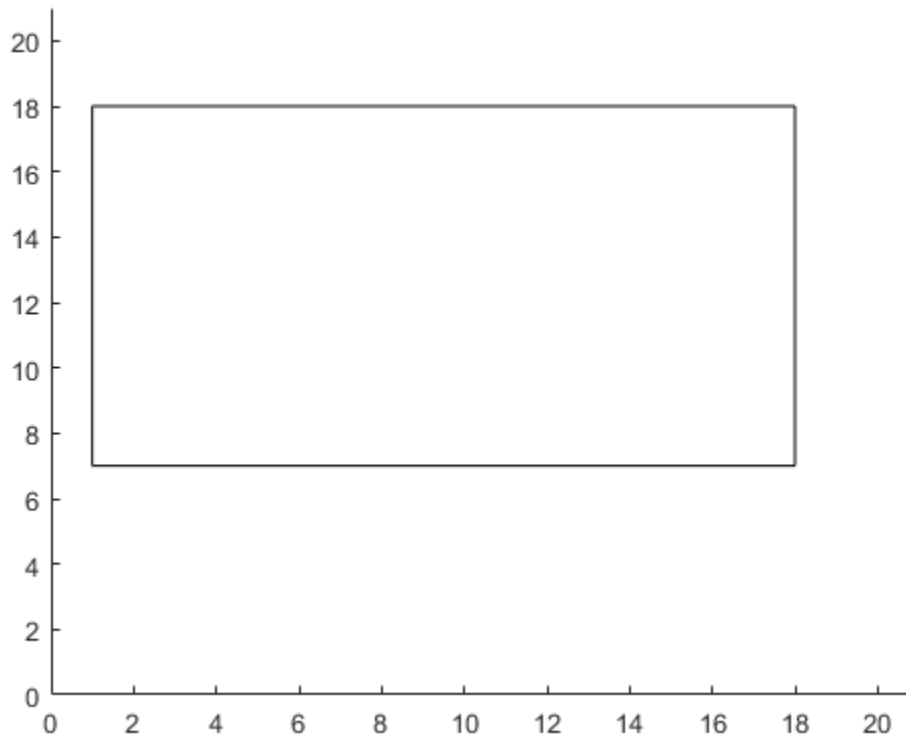
Set the objective function and call the solver.

```
fun = @gaintobj;
[x,fval] = ga(fun,nvar,[],[],[],[],lb,ub,[],intcon,options)
```

```
Got below 0.1
Optimization terminated: y
```

```
x =  
    6.0000    13.0000    9.4201    15.7052
```

```
fval =  
    0.0059
```



The output function halted the solver.

View the size of the recorded history.

```
disp(size(gapopulationhistory))
```

```
    40     4     6
```

There are six records of the 40-by-4 population matrix (40 individuals, each a 4-element row vector).

See Also

Related Examples

- “Create Custom Plot Function” on page 8-59
- “Output Function Options” on page 17-41

Custom Data Type Optimization Using the Genetic Algorithm

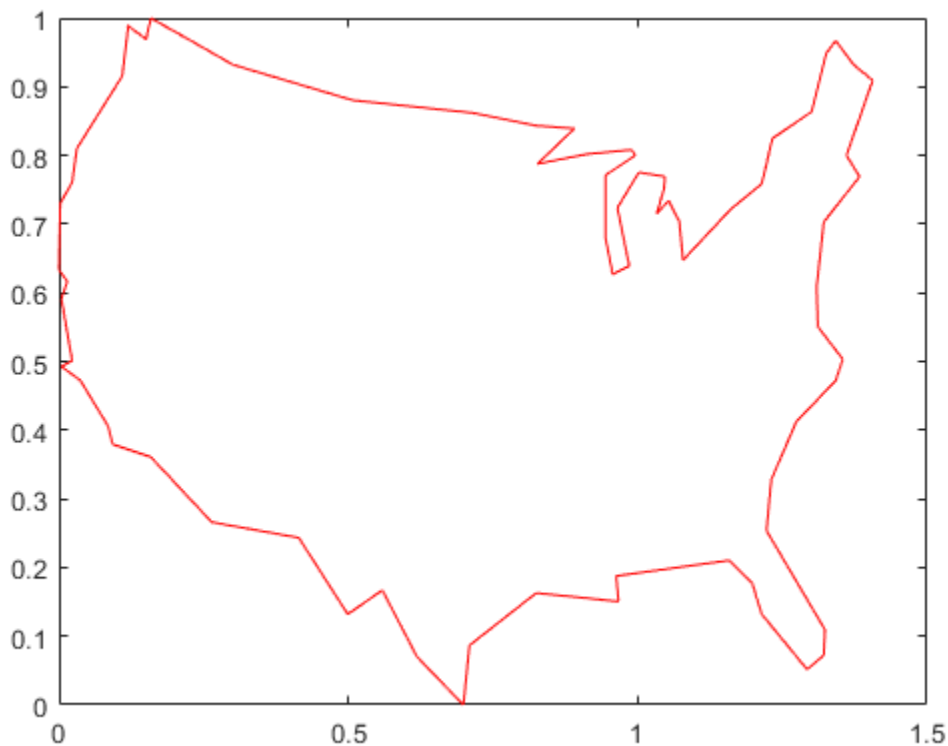
This example shows how to use the genetic algorithm to minimize a function using a custom data type. The genetic algorithm is customized to solve the traveling salesman problem.

Traveling Salesman Problem

The traveling salesman problem is an optimization problem where there is a finite number of cities, and the cost of travel between each city is known. The goal is to find an ordered set of all the cities for the salesman to visit such that the cost is minimized. To solve the traveling salesman problem, we need a list of city locations and distances, or cost, between each of them.

Our salesman is visiting cities in the United States. The file `usborder.mat` contains a map of the United States in the variables `x` and `y`, and a geometrically simplified version of the same map in the variables `xx` and `yy`.

```
load('usborder.mat','x','y','xx','yy');
plot(x,y,'Color','red'); hold on;
```



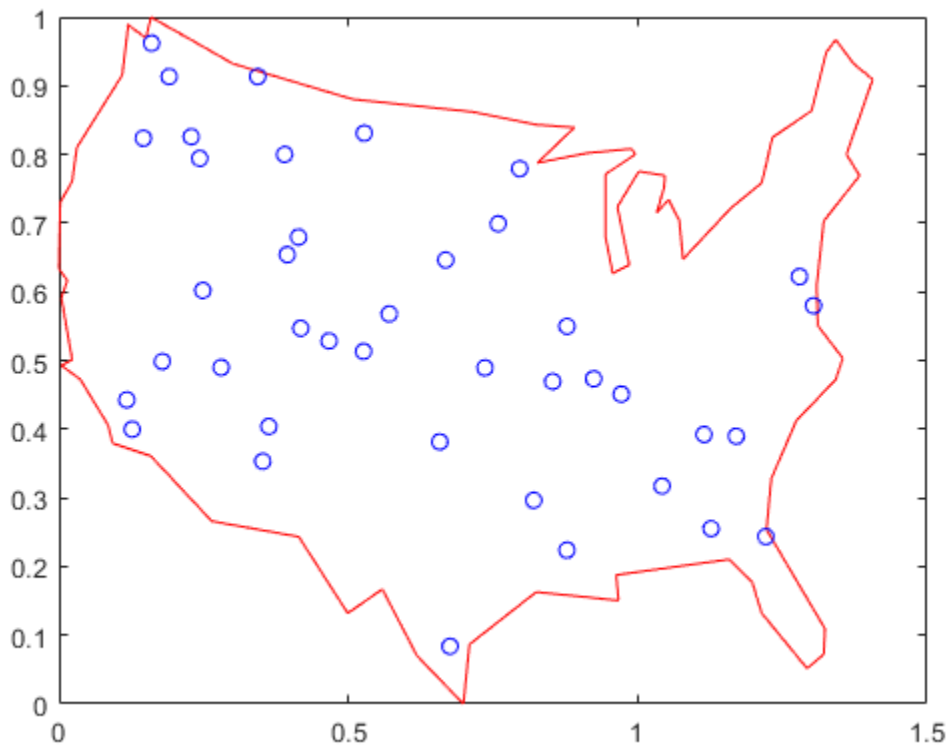
We will generate random locations of cities inside the border of the United States. We can use the `inpolygon` function to make sure that all the cities are inside or very close to the US boundary.

```
cities = 40;
locations = zeros(cities,2);
n = 1;
```

```

while (n <= cities)
    xp = rand*1.5;
    yp = rand;
    if inpolygon(xp,yp,xx,yy)
        locations(n,1) = xp;
        locations(n,2) = yp;
        n = n+1;
    end
end
end
plot(locations(:,1),locations(:,2), 'bo');

```



Blue circles represent the locations of the cities where the salesman needs to travel and deliver or pickup goods. Given the list of city locations, we can calculate the distance matrix for all the cities.

```

distances = zeros(cities);
for count1=1:cities,
    for count2=1:count1,
        x1 = locations(count1,1);
        y1 = locations(count1,2);
        x2 = locations(count2,1);
        y2 = locations(count2,2);
        distances(count1,count2)=sqrt((x1-x2)^2+(y1-y2)^2);
        distances(count2,count1)=distances(count1,count2);
    end;
end;

```

Customizing the Genetic Algorithm for a Custom Data Type

By default, the genetic algorithm solver solves optimization problems based on double and binary string data types. The functions for creation, crossover, and mutation assume the population is a matrix of type double, or logical in the case of binary strings. The genetic algorithm solver can also work on optimization problems involving arbitrary data types. You can use any data structure you like for your population. For example, a custom data type can be specified using a MATLAB® cell array. In order to use `ga` with a population of type cell array you must provide a creation function, a crossover function, and a mutation function that will work on your data type, e.g., a cell array.

Required Functions for the Traveling Salesman Problem

This section shows how to create and register the three required functions. An individual in the population for the traveling salesman problem is an ordered set, and so the population can easily be represented using a cell array. The custom creation function for the traveling salesman problem will create a cell array, say `P`, where each element represents an ordered set of cities as a permutation vector. That is, the salesman will travel in the order specified in `P{i}`. The creation function will return a cell array of size `PopulationSize`.

type `create_permutations.m`

```
function pop = create_permutations(NVARS,FitnessFcn,options)
%CREATE_PERMUTATIONS Creates a population of permutations.
% POP = CREATE_PERMUTATION(NVARS,FITNESSFCN,OPTIONS) creates a population
% of permutations POP each with a length of NVARS.
%
% The arguments to the function are
% NVARS: Number of variables
% FITNESSFCN: Fitness function
% OPTIONS: Options structure used by the GA
%
% Copyright 2004-2007 The MathWorks, Inc.

totalPopulationSize = sum(options.PopulationSize);
n = NVARS;
pop = cell(totalPopulationSize,1);
for i = 1:totalPopulationSize
    pop{i} = randperm(n);
end
```

The custom crossover function takes a cell array, the population, and returns a cell array, the children that result from the crossover.

type `crossover_permutation.m`

```
function xoverKids = crossover_permutation(parents,options,NVARS, ...
    FitnessFcn,thisScore,thisPopulation)
% CROSSOVER_PERMUTATION Custom crossover function for traveling salesman.
% XOVERKIDS = CROSSOVER_PERMUTATION(PARENTS,OPTIONS,NVARS, ...
% FITNESSFCN,THISSCORE,THISPOPULATION) crossovers PARENTS to produce
% the children XOVERKIDS.
%
% The arguments to the function are
% PARENTS: Parents chosen by the selection function
% OPTIONS: Options created from OPTIMOPTIONS
```

```

%     NVARs: Number of variables
%     FITNESSFCN: Fitness function
%     STATE: State structure used by the GA solver
%     THISSCORE: Vector of scores of the current population
%     THISPOPULATION: Matrix of individuals in the current population

%     Copyright 2004-2015 The MathWorks, Inc.

nKids = length(parents)/2;
xoverKids = cell(nKids,1); % Normally zeros(nKids,NVARs);
index = 1;

for i=1:nKids
    % here is where the special knowledge that the population is a cell
    % array is used. Normally, this would be thisPopulation(parents(index),:);
    parent = thisPopulation{parents(index)};
    index = index + 2;

    % Flip a section of parent1.
    p1 = ceil((length(parent) - 1) * rand);
    p2 = p1 + ceil((length(parent) - p1 - 1) * rand);
    child = parent;
    child(p1:p2) = fliplr(child(p1:p2));
    xoverKids{i} = child; % Normally, xoverKids(i,:);
end

```

The custom mutation function takes an individual, which is an ordered set of cities, and returns a mutated ordered set.

type `mutate_permutation.m`

```

function mutationChildren = mutate_permutation(parents ,options,NVARs, ...
    FitnessFcn, state, thisScore,thisPopulation,mutationRate)
%     MUTATE_PERMUTATION Custom mutation function for traveling salesman.
%     MUTATIONCHILDREN = MUTATE_PERMUTATION(PARENTS,OPTIONS,NVARs, ...
%     FITNESSFCN,STATE,THISSCORE,THISPOPULATION,MUTATIONRATE) mutate the
%     PARENTS to produce mutated children MUTATIONCHILDREN.
%
%     The arguments to the function are
%     PARENTS: Parents chosen by the selection function
%     OPTIONS: Options created from OPTIMOPTIONS
%     NVARs: Number of variables
%     FITNESSFCN: Fitness function
%     STATE: State structure used by the GA solver
%     THISSCORE: Vector of scores of the current population
%     THISPOPULATION: Matrix of individuals in the current population
%     MUTATIONRATE: Rate of mutation

%     Copyright 2004-2015 The MathWorks, Inc.

% Here we swap two elements of the permutation
mutationChildren = cell(length(parents),1);% Normally zeros(length(parents),NVARs);
for i=1:length(parents)
    parent = thisPopulation{parents(i)}; % Normally thisPopulation(parents(i),:)
    p = ceil(length(parent) * rand(1,2));
    child = parent;
    child(p(1)) = parent(p(2));

```



```

    child(p(2)) = parent(p(1));
    mutationChildren{i} = child; % Normally mutationChildren(i,:)
end

```

We also need a fitness function for the traveling salesman problem. The fitness of an individual is the total distance traveled for an ordered set of cities. The fitness function also needs the distance matrix to calculate the total distance.

type `traveling_salesman_fitness.m`

```

function scores = traveling_salesman_fitness(x,distances)
%TRAVELING_SALESMAN_FITNESS Custom fitness function for TSP.
% SCORES = TRAVELING_SALESMAN_FITNESS(X,DISTANCES) Calculate the fitness
% of an individual. The fitness is the total distance traveled for an
% ordered set of cities in X. DISTANCE(A,B) is the distance from the city
% A to the city B.

% Copyright 2004-2007 The MathWorks, Inc.

scores = zeros(size(x,1),1);
for j = 1:size(x,1)
    % here is where the special knowledge that the population is a cell
    % array is used. Normally, this would be pop(j,:);
    p = x{j};
    f = distances(p(end),p(1));
    for i = 2:length(p)
        f = f + distances(p(i-1),p(i));
    end
    scores(j) = f;
end

```

`ga` will call our fitness function with just one argument `x`, but our fitness function has two arguments: `x`, `distances`. We can use an anonymous function to capture the values of the additional argument, the `distances` matrix. We create a function handle `FitnessFcn` to an anonymous function that takes one input `x`, but calls `traveling_salesman_fitness` with `x`, and `distances`. The variable, `distances` has a value when the function handle `FitnessFcn` is created, so these values are captured by the anonymous function.

```

%distances defined earlier
FitnessFcn = @(x) traveling_salesman_fitness(x,distances);

```

We can add a custom plot function to plot the location of the cities and the current best route. A red circle represents a city and the blue lines represent a valid path between two cities.

type `traveling_salesman_plot.m`

```

function state = traveling_salesman_plot(options,state,flag,locations)
% TRAVELING_SALESMAN_PLOT Custom plot function for traveling salesman.
% STATE = TRAVELING_SALESMAN_PLOT(OPTIONS,STATE,FLAG,LOCATIONS) Plot city
% LOCATIONS and connecting route between them. This function is specific
% to the traveling salesman problem.

% Copyright 2004-2006 The MathWorks, Inc.
persistent x y xx yy
if strcmpi(flag,'init')
    load('usborder.mat','x','y','xx','yy');

```

```

end
plot(x,y,'Color','red');
axis([-0.1 1.5 -0.2 1.2]);

hold on;
[unused,i] = min(state.Score);
genotype = state.Population{i};

plot(locations(:,1),locations(:,2),'bo');
plot(locations(genotype,1),locations(genotype,2));
hold off

```

Once again we will use an anonymous function to create a function handle to an anonymous function which calls `traveling_salesman_plot` with the additional argument `locations`.

```

%locations defined earlier
my_plot = @(options,state,flag) traveling_salesman_plot(options, ...
    state,flag,locations);

```

Genetic Algorithm Options Setup

First, we will create an options container to indicate a custom data type and the population range.

```

options = optimoptions(@ga, 'PopulationType', 'custom', 'InitialPopulationRange', ...
    [1;cities]);

```

We choose the custom creation, crossover, mutation, and plot functions that we have created, as well as setting some stopping conditions.

```

options = optimoptions(options, 'CreationFcn', @create_permutations, ...
    'CrossoverFcn', @crossover_permutation, ...
    'MutationFcn', @mutate_permutation, ...
    'PlotFcn', my_plot, ...
    'MaxGenerations', 500, 'PopulationSize', 60, ...
    'MaxStallGenerations', 200, 'UseVectorized', true);

```

Finally, we call the genetic algorithm with our problem information.

```

numberOfVariables = cities;
[x,fval,reason,output] = ...
    ga(FitnessFcn,numberOfVariables,[],[],[],[],[],[],[],options)

```

Optimization terminated: maximum number of generations exceeded.

x =

1x1 cell array

```
{[14 12 36 3 5 11 40 25 38 37 7 30 28 10 23 21 27 4 1 29 26 31 9 24 ... ]}
```

fval =

5.3846

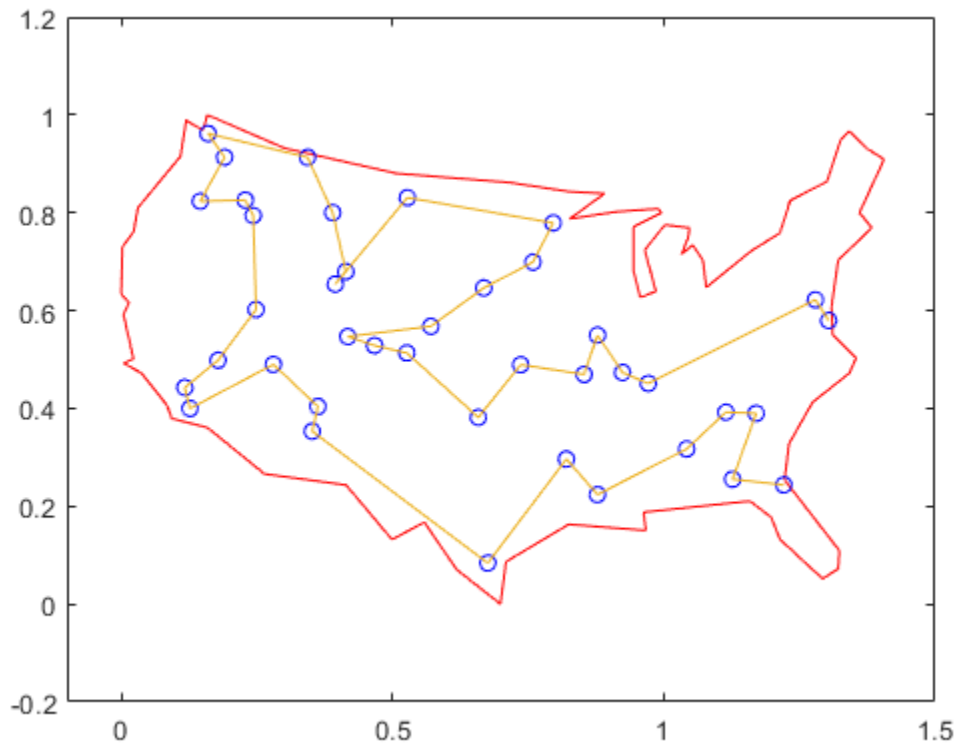
reason =

0

```
output =
```

```
struct with fields:
```

```
    problemtype: 'unconstrained'  
      rngstate: [1x1 struct]  
    generations: 500  
    funccount: 28563  
      message: 'Optimization terminated: maximum number of generations exceeded.'  
    maxconstraint: []  
    hybridflag: []
```



The plot shows the location of the cities in blue circles as well as the path found by the genetic algorithm that the salesman will travel. The salesman can start at either end of the route and at the end, return to the starting city to get back home.

See Also

More About

- “Traveling Salesman Problem: Solver-Based”

When to Use a Hybrid Function

A hybrid function is a function that continues the optimization after the original solver terminates.

These Global Optimization Toolbox solvers can automatically run a hybrid function, or second solver, after they finish:

- `ga`
- `gamultiobj`
- `particleswarm`
- `simulannealbnd`

To run a hybrid function, set the `HybridFcn` option to the second solver.

A hybrid function can obtain a more accurate solution, starting from the relatively rough solution found by the first solver, in the following circumstances:

- Whether or not the objective function has nonsmooth regions, if the solution is in a smooth region with smooth constraints, then use a hybrid function from Optimization Toolbox, such as `fmincon`.
- If the objective function or a constraint is nonsmooth near the solution, then use `patternsearch` as a hybrid function.
- Suppose that the problem has multiple local minima, and you want to obtain an accurate global solution. The single-objective solvers can search for the vicinity of a global solution, but do not necessarily obtain an extremely accurate result. If the objective function is smooth, then use a hybrid function from Optimization Toolbox, such as `fminunc`.
- For smooth multiobjective problems, a hybrid function usually improves on solutions from `gamultiobj`.

To see which solvers are available as hybrid functions, refer to the `options` input argument on the reference page for the original solver. To tune the hybrid function, you can include a separate set of options for the hybrid function. For example, if the hybrid function is `fmincon`:

```
hybridopts = optimoptions('fmincon','OptimalityTolerance',1e-10);
options = optimoptions('ga','HybridFcn',{ 'fmincon', hybridopts });
[x,fval] = ga(fun,nvars,A,b,Aeq,beq,lb,ub,nonlcon,options)
```

See Also

`ga` | `gamultiobj` | `particleswarm` | `simulannealbnd`

More About

- “Hybrid Scheme in the Genetic Algorithm” on page 8-95
- “Tune Particle Swarm Optimization Process” on page 10-14
- “Design Optimization of a Welded Beam” on page 14-62

Problem-Based Genetic Algorithm

- “Minimize Rastrigins' Function Using ga, Problem-Based” on page 9-2
- “Solve a Mixed-Integer Engineering Design Problem Using the Genetic Algorithm, Problem-Based” on page 9-5
- “Set Options in Problem-Based Approach Using varindex” on page 9-17
- “Constrained Minimization Using ga, Problem-Based” on page 9-19

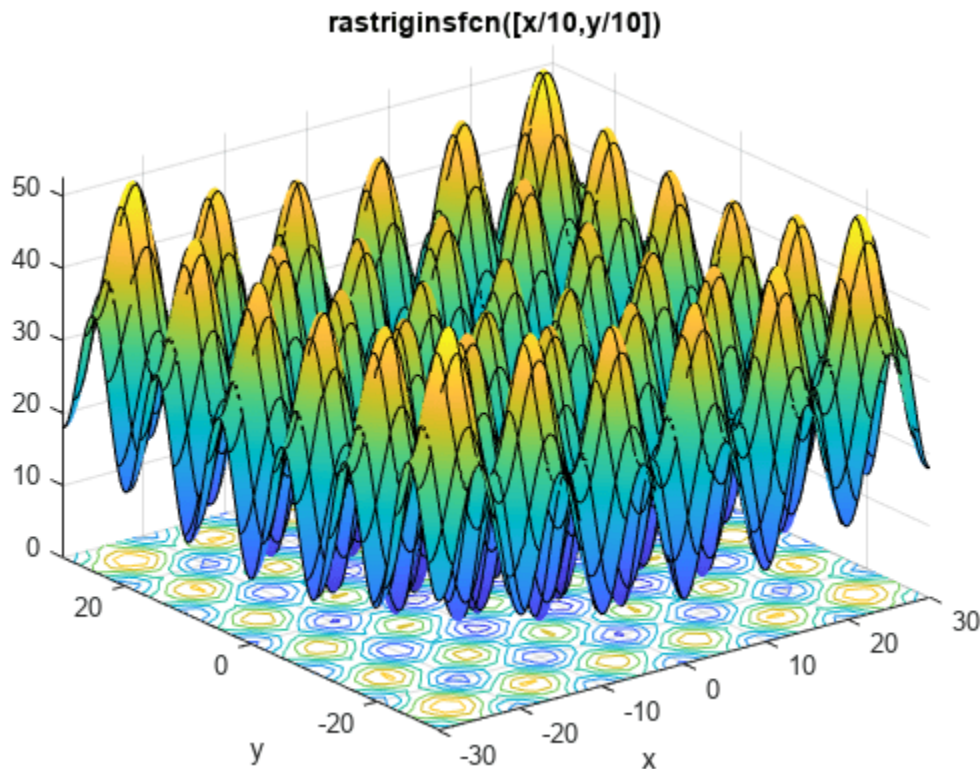
Minimize Rastrigin's Function Using ga, Problem-Based

This example shows how to minimize a function with multiple minima using the genetic algorithm in the problem-based approach. For two variables x and y , Rastrigin's function is defined as follows.

```
ras = @(x, y) 20 + x.^2 + y.^2 - 10*(cos(2*pi*x) + cos(2*pi*y));
```

Plot the function scaled by 10 in each direction.

```
rf3 = @(x, y) ras(x/10, y/10);
fsurf(rf3, [-30 30], "ShowContours", "on")
title("rastriginsfcn([x/10,y/10])")
xlabel("x")
ylabel("y")
```



The function has many local minima and a global minimum value of 0 that is attained at $x = 0$, $y = 0$. See "What Is Global Optimization?" on page 1-24

Create optimization variables x and y . Specify that the variables are bounded by ± 100 .

```
x = optimvar("x", "LowerBound", -100, "UpperBound", 100);
y = optimvar("y", "LowerBound", -100, "UpperBound", 100);
```

Create an optimization problem with the objective function `rastriginsfcn(x)`.

```
prob = optimproblem("Objective", ras(x,y));
```

Note: If you have a nonlinear function that is not composed of polynomials, rational expressions, and elementary functions such as `exp`, then convert the function to an optimization expression by using `fcn2optimexpr`. See “Convert Nonlinear Function to Optimization Expression” and “Supported Operations for Optimization Variables and Expressions”.

Create `ga` options to use the `gaplotbestf` plot function.

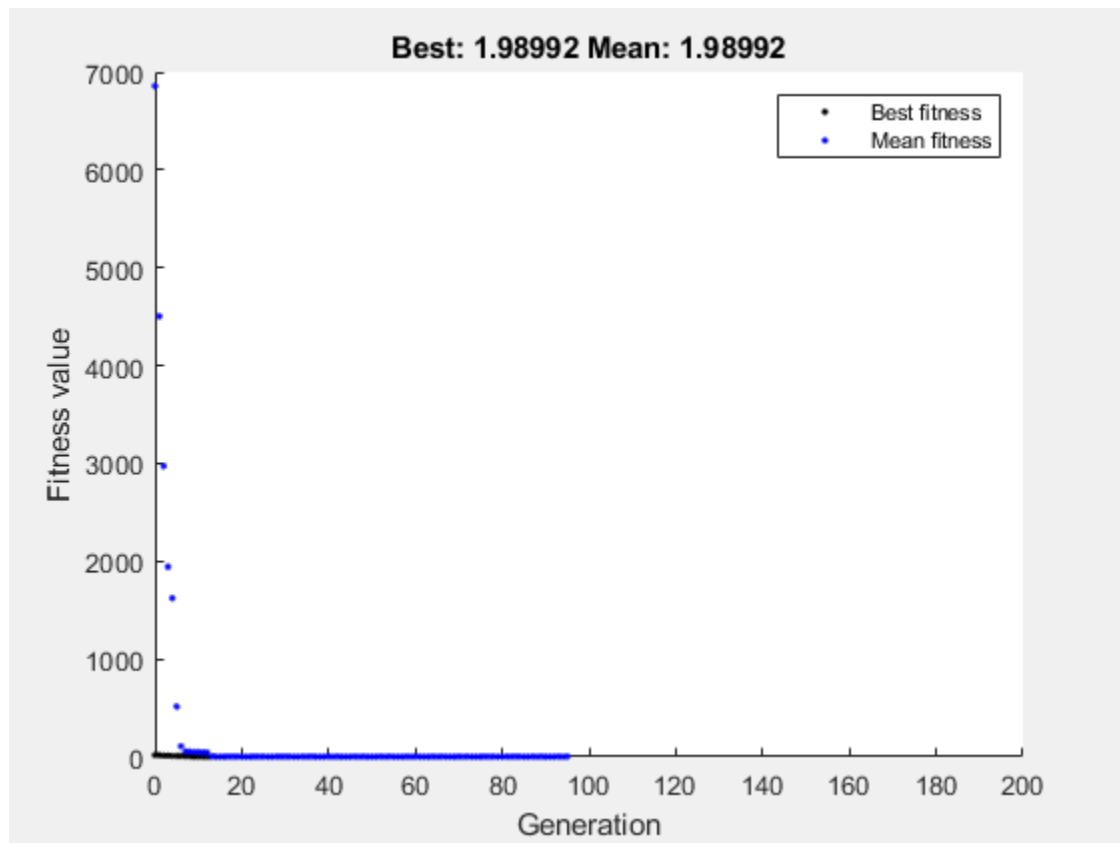
```
options = optimoptions("ga","PlotFcn","gaplotbestf");
```

Solve the problem using `ga` as the solver.

```
rng default % For reproducibility
[sol,fval] = solve(prob,"Solver","ga","Options",options)
```

Solving problem using `ga`.

Optimization terminated: average change in the fitness value less than options.FunctionTolerance



```
sol = struct with fields:
    x: 0.9950
    y: 0.9950
```

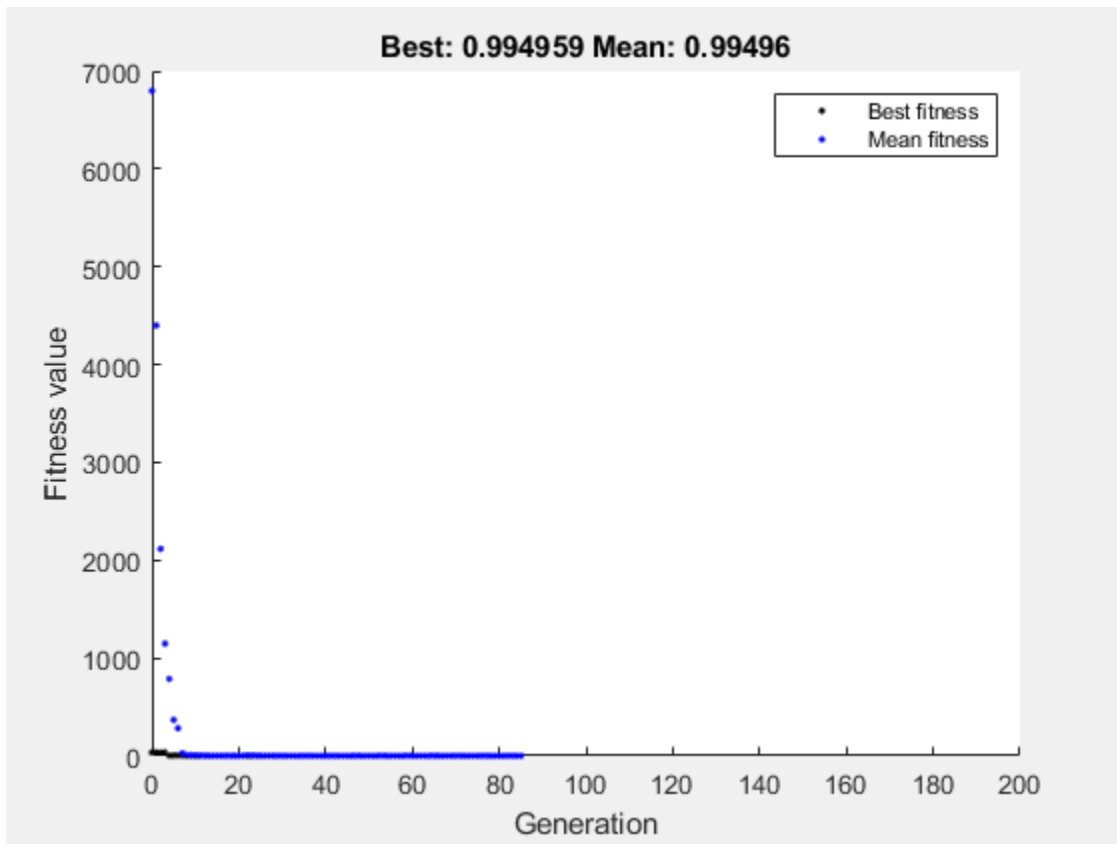
```
fval = 1.9899
```

Is the resulting function value the lowest minimum? Perform the search again. Because `ga` is a stochastic algorithm, the results can differ.

```
[sol2,fval2] = solve(prob,"Solver","ga","Options",options)
```

Solving problem using ga.

Optimization terminated: average change in the fitness value less than options.FunctionTolerance



```
sol2 = struct with fields:
  x: 0.9950
  y: -4.9289e-06
```

```
fval2 = 0.9950
```

The second solution is better because it has a lower function value. A solution returned by `ga` is not guaranteed to be a global solution.

See Also

`ga` | `fcn2optimexpr` | `solve`

Related Examples

- “Genetic Algorithm”

Solve a Mixed-Integer Engineering Design Problem Using the Genetic Algorithm, Problem-Based

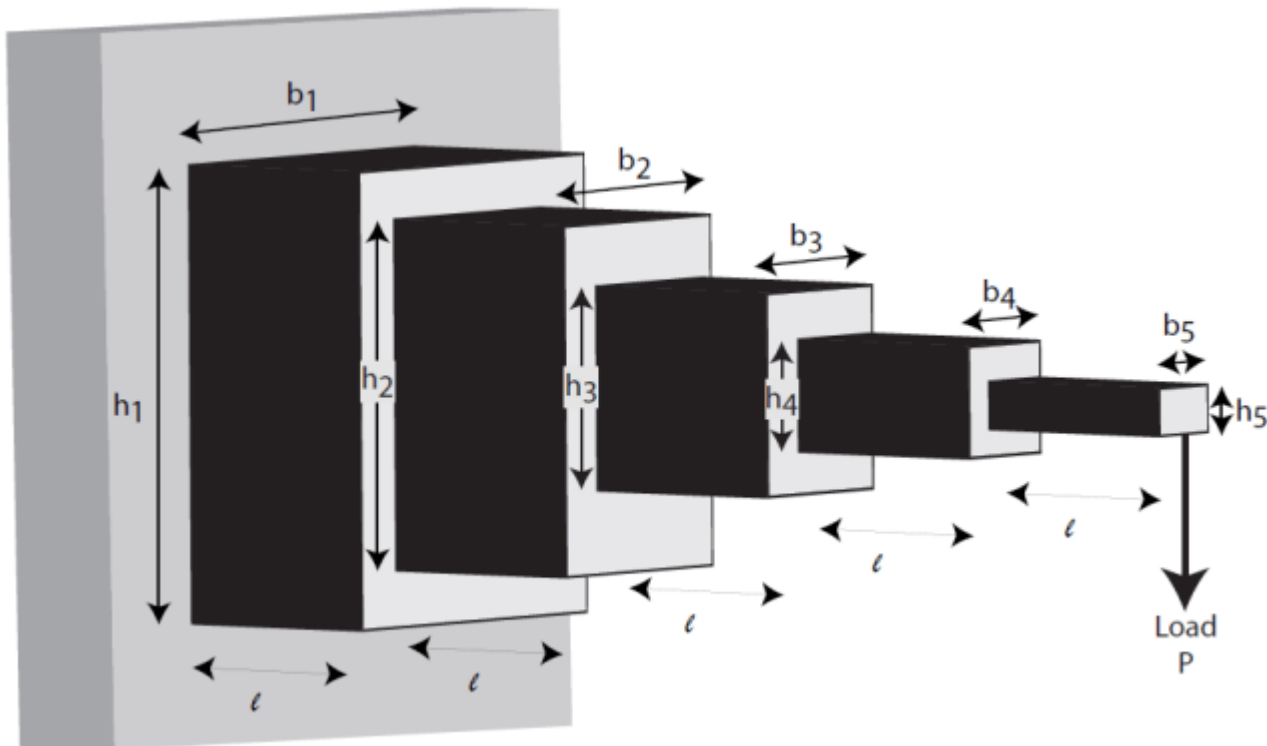
This example shows how to solve a mixed-integer engineering design problem using the genetic algorithm (ga) solver in Global Optimization Toolbox. The example uses the problem-based approach. For a version using the solver-based approach, see “Solve a Mixed-Integer Engineering Design Problem Using the Genetic Algorithm” on page 8-47.

The problem illustrated in this example involves the design of a stepped cantilever beam. In particular, the beam must be able to carry a prescribed end load. The optimization problem is to minimize the beam volume subject to various engineering design constraints.

This problem is described in Thanedar and Vanderplaats [1] on page 9-16.

Stepped Cantilever Beam Design Problem

A stepped cantilever beam is supported at one end, and a load is applied at the free end, as shown in the following figure. The beam must be able to support the given load P at a fixed distance L from the support. Designers of the beam can vary the width (b_i) and height (h_i) of each step, or section. Each section of the cantilever has the same length, $l = L_1$.



Volume of the Beam

The volume of the beam V is the sum of the volume of the individual sections.

$$V = l(b_1h_1 + b_2h_2 + b_3h_3 + b_4h_4 + b_5h_5).$$

Constraints on the Design: Bending Stress

Consider a single cantilever beam, with the center of coordinates at the center of its cross section at the free end of the beam. The bending stress at a point (x, y, z) in the beam is given by the equation

$$\sigma_b = M(x)y/I,$$

where $M(x)$ is the bending moment at x , x is the distance from the end load, and I is the area moment of inertia of the beam.

In the stepped cantilever beam shown in the figure, the maximum moment of each section of the beam is PD_i , where D_i is the maximum distance from the end load P for each section of the beam. Therefore, the maximum stress for the i th section of the beam σ_i is given by

$$\sigma_i = PD_i(h_i/2)/I_i,$$

where the maximum stress occurs at the edge of the beam, $y = h_i/2$. The area moment of inertia of the i th section of the beam is given by

$$I_i = b_i h_i^3 / 12.$$

Substituting this expression into the equation for σ_i gives

$$\sigma_i = 6PD_i/b_i h_i^2.$$

The bending stress in each part of the cantilever must not exceed the maximum allowable stress σ_{max} . Therefore, the five bending stress constraints (one for each step of the cantilever) are:

$$\frac{6Pl}{b_5 h_5^2} \leq \sigma_{max}$$

$$\frac{6P(2l)}{b_4 h_4^2} \leq \sigma_{max}$$

$$\frac{6P(3l)}{b_3 h_3^2} \leq \sigma_{max}$$

$$\frac{6P(4l)}{b_2 h_2^2} \leq \sigma_{max}$$

$$\frac{6P(5l)}{b_1 h_1^2} \leq \sigma_{max}$$

Constraints on the Design: End Deflection

You can calculate the end deflection of the cantilever using Castigliano's second theorem, which states that

$$\delta = \frac{\partial U}{\partial P},$$

where δ is the deflection of the beam, and U is the energy stored in the beam due to the applied force P .

The energy stored in a cantilever beam is given by

$$U = \int_0^L M^2/2EI \, dx,$$

where M is the moment of the applied force at x .

Given that $M = Px$ for a cantilever beam, you can write the preceding equation as

$$U = P^2/2E \int_0^l [(x+4l)^2/I_1 + (x+3l)^2/I_2 + (x+2l)^2/I_3 + (x+l)^2/I_4 + x^2/I_5] \, dx,$$

where I_n is the area moment of inertia of the n th part of the cantilever. Evaluating the integral gives this expression for U

$$U = (P^2/2)(l^3/3E)(61/I_1 + 37/I_2 + 19/I_3 + 7/I_4 + 1/I_5).$$

Applying Castigliano's theorem, the end deflection of the beam is given by

$$\delta = Pl^3/3E(61/I_1 + 37/I_2 + 19/I_3 + 7/I_4 + 1/I_5).$$

The end deflection of the cantilever δ must be less than the maximum allowable deflection δ_{max} , which gives the constraint

$$Pl^3/3E(61/I_1 + 37/I_2 + 19/I_3 + 7/I_4 + 1/I_5) \leq \delta_{max}.$$

Constraints on the Design: Aspect Ratio

For each step of the cantilever, the aspect ratio must not exceed a maximum allowable aspect ratio a_{max} . That is,

$$h_i/b_i \leq a_{max} \text{ for } i = 1, \dots, 5.$$

Constraints on the Design: Bounds and Integer Constraints

The first step of the beam can be machined to the nearest centimeter only. That is, b_1 and h_1 must be integers. The remaining variables are continuous. The bounds on the variables are:

$$1 \leq b_1 \leq 5$$

$$30 \leq h_1 \leq 65$$

$$2.4 \leq b_2, b_3 \leq 3.1$$

$$45 \leq h_2, h_3 \leq 60$$

$$1 \leq b_4, b_5 \leq 5$$

$$30 \leq h_4, h_5 \leq 65$$

Design Parameters for This Problem

For the problem in this example, the end load that the beam must support is $P = 50000N$.

The beam lengths and maximum end deflection are:

- Total beam length, $L = 500cm$
- Length of an individual section of the beam, $l = L_1 = 100cm$
- Maximum beam end deflection, $\delta_{max} = 2.7cm$

The maximum allowed stress in each step of the beam is $\sigma_{max} = 14000N/cm^2$.

Young's modulus of each step of the beam is $E = 2 \times 10^7 N/cm^2$.

Problem-Based Setup

Create optimization variables for this problem. The width and height variables for the first section of the beam are of type integer, so you must create them separately from the other four variables, which are continuous.

```
b1 = optimvar("b1","Type","integer","LowerBound",1,"UpperBound",5);
h1 = optimvar("h1","Type","integer","LowerBound",30,"UpperBound",65);
bc = optimvar("bc",4,"LowerBound",[2.4 2.4 1 1],"UpperBound",[3.1 3.1 5 5]);
hc = optimvar("hc",4,"LowerBound",[45 45 30 30],"UpperBound",[60 60 65 65]);
```

For convenience, put the height and width variables into single variables. You can then express the objective and constraints easily in terms of these variables.

```
h = [h1;hc];
b = [b1;bc];
```

Create an optimization problem with the volume of the beam as the objective function, where each step (or section) of the beam is $L_1 = 100$ cm long: $\text{volume} = L_1 \sum h_i w_i$.

```
L_1 = 100; % Length of each step of the cantilever
prob = optimproblem("Objective",L_1*dot(h,b));
```

Create the constraints on the stress.

```
P = 50000; % End load
E = 2e7; % Young's modulus in N/cm^2
deltaMax = 2.7; % Maximum end deflection
sigmaMax = 14000; % Maximum stress in each section of the beam
aMax = 20; % Maximum aspect ratio in each section of the beam

stress = 6*P*L_1./(b.*(h.^2));
stepnum = (5:-1:1)';
stress = stress.*stepnum;
prob.Constraints.stress = stress <= sigmaMax;
```

Create the constraint on the deflection.

```
deflectionMultiplier = (P*L_1^3/E)*[244 148 76 28 4];
bh3 = 1./(b.*(h.^3));
prob.Constraints.deflection = deflectionMultiplier*bh3 <= deltaMax;
```

Create the constraints on the aspect ratio.

```
prob.Constraints.aspect = h <= aMax*b;
```

Review the problem setup.

```
show(prob)
```

```
OptimizationProblem :

Solve for:
  b1, bc, h1, hc
where:
  b1, h1 integer

minimize :
  100*h1*b1 + 100*hc(1)*bc(1) + 100*hc(2)*bc(2) + 100*hc(3)*bc(3) + 100*hc(4)*bc(4)

subject to stress:
  arg_LHS <= arg_RHS

where:

  arg2 = zeros(5, 1);
  arg1 = zeros(5, 1);
  arg1(1) = h1;
  arg1(2:5) = hc;
  arg2(1) = b1;
  arg2(2:5) = bc;
  arg_LHS = ((300000000 ./ (arg2(:) .* arg1(:).^2)) .* extraParams{1});
  arg2 = 14000;
  arg1 = arg2(ones(1,5));
  arg_RHS = arg1(:);

extraParams{1}:

  5
  4
  3
  2
  1

subject to deflection:
  arg_LHS <= 2.7

where:

  arg2 = zeros(5, 1);
  arg1 = zeros(5, 1);
  arg1(1) = h1;
  arg1(2:5) = hc;
  arg2(1) = b1;
  arg2(2:5) = bc;
  arg_LHS = (extraParams{1} * (1 ./ (arg2(:) .* arg1(:).^3)));
```

```
extraParams{1}:  
    610000    370000    190000    70000    10000  
  
subject to aspect:  
-20*b1 + h1 <= 0  
-20*bc(1) + hc(1) <= 0  
-20*bc(2) + hc(2) <= 0  
-20*bc(3) + hc(3) <= 0  
-20*bc(4) + hc(4) <= 0  
  
variable bounds:  
1 <= b1 <= 5  
  
2.4 <= bc(1) <= 3.1  
2.4 <= bc(2) <= 3.1  
1 <= bc(3) <= 5  
1 <= bc(4) <= 5  
  
30 <= h1 <= 65  
  
45 <= hc(1) <= 60  
45 <= hc(2) <= 60  
30 <= hc(3) <= 65  
30 <= hc(4) <= 65
```

Solve the Problem

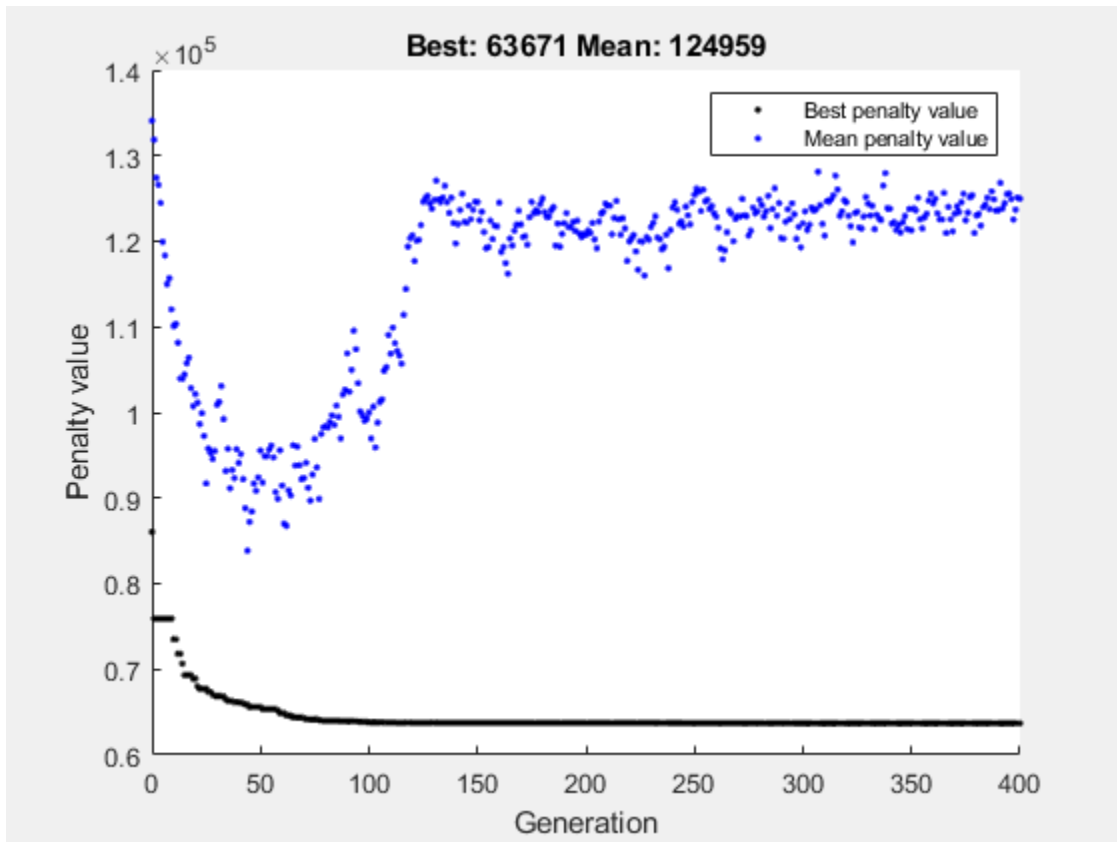
Set options to use a moderate population size of 150, a moderate maximum number of generations of 400, a slightly larger than default elite count of 10, a small function tolerance of 1e-8, and a plot function showing the function value during the iterations.

```
opts = optimoptions(@ga, ...  
    'PopulationSize', 150, ...  
    'MaxGenerations', 400, ...  
    'EliteCount', 10, ...  
    'FunctionTolerance', 1e-8, ...  
    'PlotFcn', @gaplotbestf);
```

Solve the problem, specifying the ga solver and the options.

```
rng default % For reproducibility  
[sol,fval,exitflag] = solve(prob,"Solver","ga","Options",opts)
```

Solving problem using ga.



Optimization terminated: maximum number of generations exceeded.

```
sol = struct with fields:
    b1: 3
    bc: [4x1 double]
    h1: 60
    hc: [4x1 double]
```

```
fval = 6.3671e+04
```

```
exitflag =
    SolverLimitExceeded
```

View the solution variables and their bounds.

```
widths = [sol.b1;sol.bc];
heights = [sol.h1;sol.hc];
widthbounds = [b1.LowerBound b1.UpperBound;
    bc.LowerBound bc.UpperBound];
heightbounds = [h1.LowerBound h1.UpperBound;
    hc.LowerBound hc.UpperBound];
T = table(widths,heights,widthbounds,heightbounds,...
    'VariableNames',["Width" "Height" "Width Bounds" "Height Bounds"])
```

```
T=5x4 table
    Width    Height    Width Bounds    Height Bounds
```

3	60	1	5	30	65
2.8105	56.211	2.4	3.1	45	60
2.6183	52.301	2.4	3.1	45	60
2.2351	43.822	1	5	30	65
1.902	33.566	1	5	30	65

The solution is not at any of the bounds. The first solution variables are integer valued, as specified.

Add Discrete Noninteger Variable Constraints

Suppose the engineers learn that the second and third steps of the cantilever can have widths and heights selected from a standard set only. With the addition of this constraint, the problem is identical to the one solved in [1].

First, delineate the extra constraints to add to the optimization:

- The width of the second and third steps of the beam must be selected from the set [2.4, 2.6, 2.8, 3.1] cm.
- The height of the second and third steps of the beam must be selected from the set [45, 50, 55, 60] cm.

To solve this problem, you need to specify the variables $wc(1)$, $wc(2)$, $hc(1)$, and $hc(2)$ as discrete variables. Ideally, you would use $S(x_j)$ as the discrete value, where S represents the allowable set of values and x_j represents a problem variable. However, you cannot use an optimization variable as an index. You can get around this problem by calling `fcn2optimexpr`.

```
widthlist = [2.4,2.6,2.8,3.1];
heightlist = [45 50 55 60];
b23 = optimvar("w23",2,"Type","integer",...
    "LowerBound",1,"UpperBound",length(widthlist));
h23 = optimvar("h23",2,"Type","integer",...
    "LowerBound",1,"UpperBound",length(heightlist));
b45 = optimvar("b45",2,"LowerBound",1,"UpperBound",5);
h45 = optimvar("h45",2,"LowerBound",30,"UpperBound",65);
% Preferred syntax is we = [widthlist(b23(1));widthlist(b23(2))];
% However, this syntax is illegal.
% Instead call fcn2optimexpr.
we = fcn2optimexpr(@(x)[widthlist(x(1));widthlist(x(2))],b23);
he = fcn2optimexpr(@(x)[heightlist(x(1));heightlist(x(2))],h23);
```

As you did earlier, create the expressions b and h to represent the variables.

```
b = [b1;we;b45];
h = [h1;he;h45];
```

The remainder of the problem formulation is the same as earlier.

```
prob2 = optimproblem("Objective",L_1*dot(h,b));
```

Create the constraints on the stress.

```
stress = 6*P*L_1./(b.*(h.^2));
stepnum = (5:-1:1)';
```



```
stress = stress.*stepnum;
prob2.Constraints.stress = stress <= sigmaMax;
```

Create the constraint on the deflection.

```
deflectionMultiplier = (P*L_1^3/E)*[244 148 76 28 4];
bh3 = 1./(b.*(h.^3));
prob2.Constraints.deflection = deflectionMultiplier*bh3 <= deltaMax;
```

Create the constraints on the aspect ratio.

```
prob2.Constraints.aspect = h <= aMax*b;
```

Review the problem setup.

```
show(prob2)
```

```
OptimizationProblem :
```

```
Solve for:
```

```
    b1, b45, h1, h23, h45, w23
```

```
where:
```

```
    b1, h1, h23, w23 integer
```

```
minimize :
```

```
    (100 .* (arg1(:).' * arg2(:)))
```

```
where:
```

```
    arg2 = zeros(5, 1);
    arg1 = zeros(5, 1);
    anonymousFunction2 = @(x)[heightlist(x(1));heightlist(x(2))];
    arg1(1) = h1;
    arg1(2:3) = anonymousFunction2(h23);
    arg1(4:5) = h45;
    anonymousFunction1 = @(x)[widthlist(x(1));widthlist(x(2))];
    arg2(1) = b1;
    arg2(2:3) = anonymousFunction1(w23);
    arg2(4:5) = b45;
```

```
subject to stress:
```

```
    arg_LHS <= arg_RHS
```

```
where:
```

```
    arg2 = zeros(5, 1);
    arg1 = zeros(5, 1);
    anonymousFunction2 = @(x)[heightlist(x(1));heightlist(x(2))];
    arg1(1) = h1;
    arg1(2:3) = anonymousFunction2(h23);
    arg1(4:5) = h45;
    anonymousFunction1 = @(x)[widthlist(x(1));widthlist(x(2))];
    arg2(1) = b1;
    arg2(2:3) = anonymousFunction1(w23);
    arg2(4:5) = b45;
    arg_LHS = ((300000000 ./ (arg2(:) .* arg1(:).^2)) .* extraParams{1});
    arg2 = 14000;
    arg1 = arg2(ones(1,5));
```

```

        arg_RHS = arg1(:);
    extraParams{1}:
        5
        4
        3
        2
        1

subject to deflection:
    arg_LHS <= 2.7

where:
    arg2 = zeros(5, 1);
    arg1 = zeros(5, 1);
    anonymousFunction2 = @(x)[heightlist(x(1));heightlist(x(2))];
    arg1(1) = h1;
    arg1(2:3) = anonymousFunction2(h23);
    arg1(4:5) = h45;
    anonymousFunction1 = @(x)[widthlist(x(1));widthlist(x(2))];
    arg2(1) = b1;
    arg2(2:3) = anonymousFunction1(w23);
    arg2(4:5) = b45;
    arg_LHS = (extraParams{1} * (1 ./ (arg2(:) .* arg1(:).^3)));

    extraParams{1}:
        610000      370000      190000      70000      10000

subject to aspect:
    arg_LHS <= arg_RHS

where:
    arg1 = zeros(5, 1);
    arg1(1) = h1;
    anonymousFunction2 = @(x)[heightlist(x(1));heightlist(x(2))];
    arg1(2:3) = anonymousFunction2(h23);
    arg1(4:5) = h45;
    arg_LHS = arg1(:);
    arg1 = zeros(5, 1);
    anonymousFunction1 = @(x)[widthlist(x(1));widthlist(x(2))];
    arg1(1) = b1;
    arg1(2:3) = anonymousFunction1(w23);
    arg1(4:5) = b45;
    arg_RHS = (20 .* arg1(:));

variable bounds:
    1 <= b1 <= 5

```

```

1 <= b45(1) <= 5
1 <= b45(2) <= 5

30 <= h1 <= 65

1 <= h23(1) <= 4
1 <= h23(2) <= 4

30 <= h45(1) <= 65
30 <= h45(2) <= 65

1 <= w23(1) <= 4
1 <= w23(2) <= 4

```

Solve the Problem with Discrete Variable Constraints

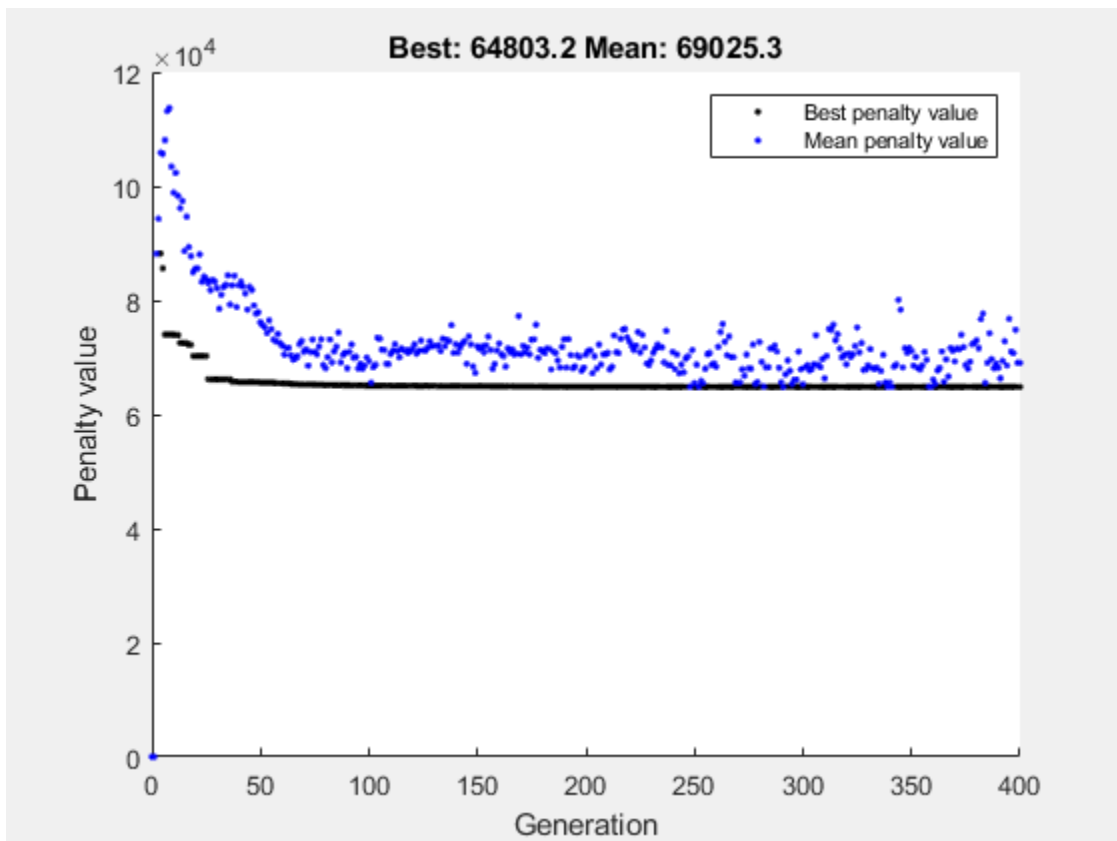
Solve the problem, specifying the ga solver and the options.

```

rng default % For reproducibility
[sol2,fval2,exitflag2] = solve(prob2,"Solver","ga","Options",opts)

```

Solving problem using ga.



Optimization terminated: maximum number of generations exceeded.

```

sol2 = struct with fields:
    b1: 3
    b45: [2x1 double]

```

```

h1: 60
h23: [2x1 double]
h45: [2x1 double]
w23: [2x1 double]

fval2 = 6.4803e+04

exitflag2 =
    SolverLimitExceeded

```

The objective value increases, because adding constraints can only increase the objective.

View the solution and compare it to its bounds.

```

widths = [sol2.b1;widthlist(sol2.w23(1));widthlist(sol2.w23(2));sol2.b45];
heights = [sol2.h1;heightlist(sol2.h23(1));heightlist(sol2.h23(2));sol2.h45];
widthbounds = [b1.LowerBound b1.UpperBound;
    widthlist(1) widthlist(end);
    widthlist(1) widthlist(end);
    b45.LowerBound b45.UpperBound];
heightbounds = [h1.LowerBound h1.UpperBound;
    heightlist(1) heightlist(end);
    heightlist(1) heightlist(end);
    h45.LowerBound h45.UpperBound];
T = table(widths,heights,widthbounds,heightbounds,...
    'VariableNames',{'Width' 'Height' 'Width Bounds' 'Height Bounds'})

```

```

T=5x4 table
    Width      Height      Width Bounds      Height Bounds
    _____  _____  _____  _____
         3         60         1         5         30         65
        3.1         55         2.4         3.1         45         60
        2.6         50         2.4         3.1         45         60
        2.286       45.72         1         5         30         65
        1.8532       34.004         1         5         30         65

```

The only solution variable that is at a bound is the width of the second section, which is 3.1, its maximum.

References

[1] Thanedar, P. B., and G. N. Vanderplaats. "Survey of Discrete Variable Optimization for Structural Design." *Journal of Structural Engineering* 121 (3), 1995, pp. 301-306.

See Also

ga | fcn2optimexpr | solve

Related Examples

- "Solve a Mixed-Integer Engineering Design Problem Using the Genetic Algorithm" on page 8-47

Set Options in Problem-Based Approach Using varindex

To set certain options when using the problem-based approach, you must convert problem variables to indices by calling `varindex`. For example, the `ga` solver accepts an option named `InitialPopulationRange` that is a two-row matrix. The first row represents the lower limit and the second row represents the upper limit of the problem variables. The columns of the matrix represent individuals in the population, which are the problem variables. To match the column indices to the problem variables, use `varindex`.

For example, set the objective function to the helper function `mrosenbrock`, given at the end of this example on page 9-18. This objective function is close to 0 near the point $x_i = y_i = 1$ for all i . Create 3-D problem variables `x` and `y` in row form, which is the form `ga` expects.

```
x = optimvar("x",1,3);
y = optimvar("y",1,3);
```

Create an optimization problem with the objective function `mrosenbrock(x,y)`.

```
prob = optimproblem("Objective",mrosenbrock(x,y));
```

Set the initial range of the `x` variables to `[-1 2]`, and the range of the `y` variables to `[0 3]`. To do so, find the indices for the variables.

```
xidx = varindex(prob,"x")
```

```
xidx = 1×3
```

```
    1    2    3
```

```
yidx = varindex(prob,"y")
```

```
yidx = 1×3
```

```
    4    5    6
```

Set the initial range as a two-row matrix with the first row containing the lower bounds, and the second row containing the upper bounds.

```
poprange = zeros(2,max([xidx,yidx]));
poprange(1,xidx) = -1;
poprange(2,xidx) = 2;
poprange(1,yidx) = 0;
poprange(2,yidx) = 3;
disp(poprange)
```

```
    -1    -1    -1     0     0     0
     2     2     2     3     3     3
```

Set the random number generator, and solve the problem using the initial range matrix.

```
rng default % For reproducibility
opts = optimoptions("ga","InitialPopulationRange",poprange);
[sol,fval] = solve(prob,"Solver","ga","Options",opts)
```

```
Solving problem using ga.  
Optimization terminated: average change in the fitness value less than options.FunctionTolerance
```

```
sol = struct with fields:  
    x: [1.2583 0.7522 1.2317]  
    y: [1.5830 0.5653 1.5167]
```

```
fval = 0.1818
```

The returned solution has a fairly small objective function value.

Helper Function

This code creates the mosenbrock helper function.

```
function F = mosenbrock(x,y)  
F = [10*(y - x.^2),1 - x];  
F = sum(F.^2,2);  
end
```

See Also

varindex

Related Examples

- “Genetic Algorithm”
- “Problem-Based Global Optimization Setup”

Constrained Minimization Using ga, Problem-Based

This example shows how to minimize an objective function, subject to nonlinear inequality constraints and bounds, using `ga` in the problem-based approach. For a solver-based version of this problem, see “Constrained Minimization Using the Genetic Algorithm” on page 8-27.

Constrained Minimization Problem

For this problem, the fitness function to minimize is a simple function of 2-D variables X and Y .

```
camxy = @(X,Y)(4 - 2.1.*X.^2 + X.^4./3).*X.^2 + X.*Y + (-4 + 4.*Y.^2).*Y.^2;
```

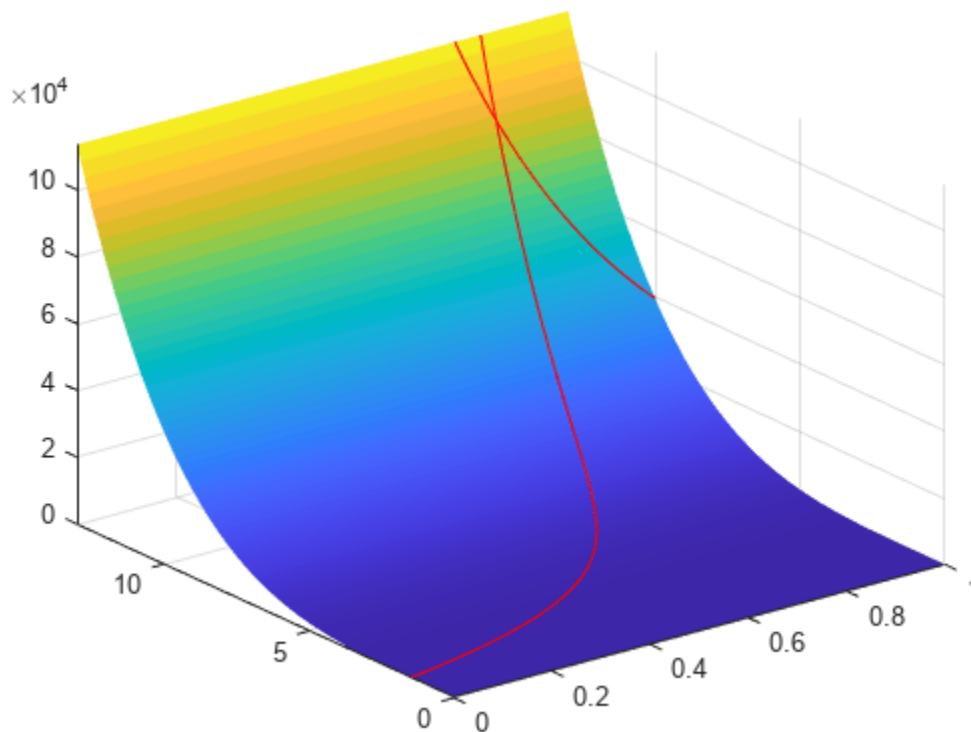
This function is described in Dixon and Szego [1] on page 9-24.

Additionally, the problem has nonlinear constraints and bounds.

```
x*y + x - y + 1.5 <= 0 (nonlinear constraint)
10 - x*y <= 0          (nonlinear constraint)
0 <= x <= 1           (bound)
0 <= y <= 13          (bound)
```

Plot the nonlinear constraint region on a surface plot of the fitness function. The constraints limit the solution to the small region above both red curves.

```
x1 = linspace(0,1);
y1 = (-x1 - 1.5)./(x1 - 1);
y2 = 10./x1;
[X,Y] = meshgrid(x1,linspace(0,13));
Z = camxy(X,Y);
surf(X,Y,Z,"LineStyle","none")
hold on
z1 = camxy(x1,y1);
z2 = camxy(x1,y2);
plot3(x1,y1,z1,'r-',x1,y2,z2,'r-')
xlim([0 1])
ylim([0 13])
zlim([0,max(Z,[],"all")])
hold off
```



Create Optimization Variables, Problem, and Constraints

To set up this problem, create optimization variables x and y . Set the bounds when you create the variables.

```
x = optimvar("x", "LowerBound", 0, "UpperBound", 1);
y = optimvar("y", "LowerBound", 0, "UpperBound", 13);
```

Create the objective as an optimization expression.

```
cam = camxy(x,y);
```

Create an optimization problem with this objective function.

```
prob = optimproblem("Objective", cam);
```

Create the two nonlinear inequality constraints, and include them in the problem.

```
prob.Constraints.cons1 = x*y + x - y + 1.5 <= 0;
prob.Constraints.cons2 = 10 - x*y <= 0;
```

Review the problem.

```
show(prob)
```

```
OptimizationProblem :
```

```
Solve for:
```



```

x, y
minimize :
    (((((4 - (2.1 .* x.^2)) + (x.^4 ./ 3)) .* x.^2) + (x .* y)) + (((-4) + (4 .* y.^2)) .* y)

subject to cons1:
    (((x .* y) + x) - y) + 1.5) <= 0

subject to cons2:
    (10 - (x .* y)) <= 0

variable bounds:
    0 <= x <= 1

    0 <= y <= 13

```

Solve Problem

Solve the problem, specifying the ga solver.

```
[sol,fval] = solve(prob,"Solver","ga")
```

Solving problem using ga.

Optimization finished: average change in the fitness value less than options.FunctionTolerance and

sol = struct with fields:

```

x: 0.8122
y: 12.3103

```

```
fval = 9.1268e+04
```

Add Visualization

To observe the solver's progress, specify options that select two plot functions. The plot function `gaplotbestf` plots the best objective function value at every iteration, and the plot function `gaplotmaxconstr` plots the maximum constraint violation at every iteration. Set these two plot functions in a cell array. Also, display information about the solver's progress in the Command Window by setting the `Display` option to `'iter'`.

```

options = optimoptions(@ga,...
    'PlotFcn',{@gaplotbestf,@gaplotmaxconstr},...
    'Display','iter');

```

Run the solver, including the options argument.

```
[sol,fval] = solve(prob,"Solver","ga","Options",options)
```

Solving problem using ga.

Single objective optimization:

2 Variable(s)

2 Nonlinear inequality constraint(s)

Options:

```

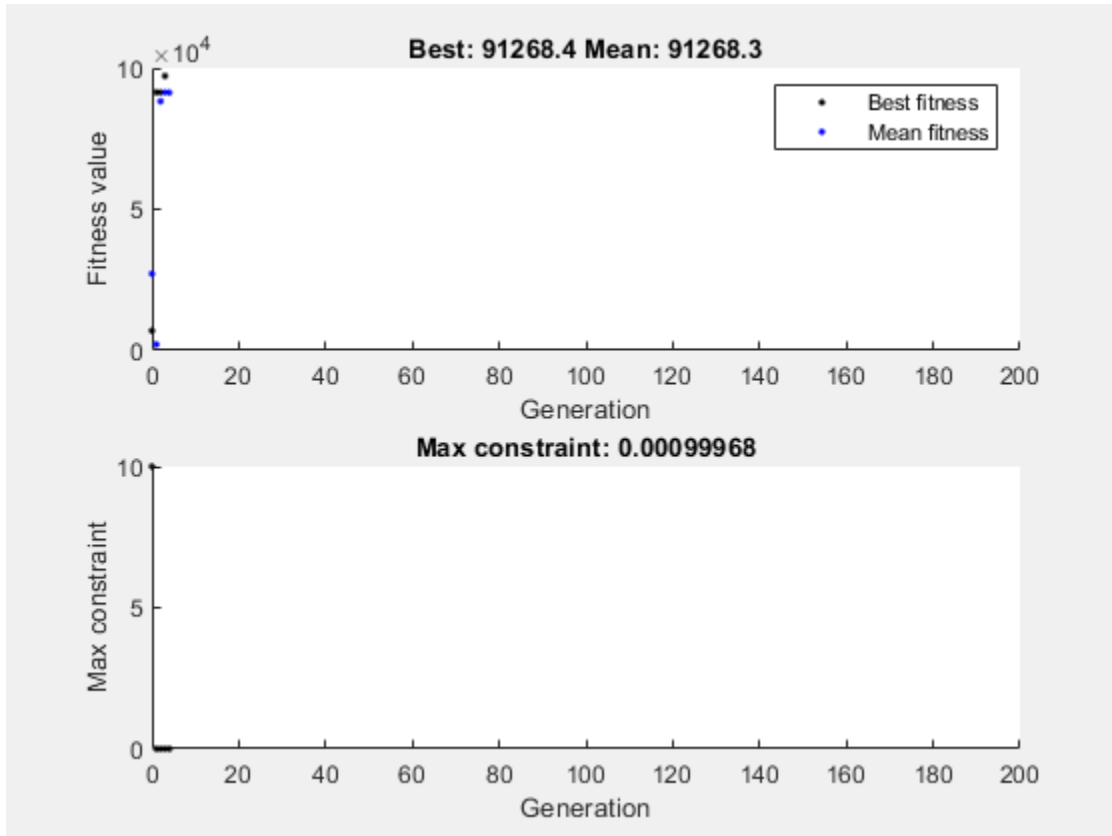
CreationFcn:    @gacreationuniform
CrossoverFcn:   @crossoverscattered
SelectionFcn:   @selectionstochunif

```

```
MutationFcn: @mutationadaptfeasible
```

Generation	Func-count	Best f(x)	Max Constraint	Stall Generations
1	2520	91357.8	0	0
2	4982	91324.1	4.55e-05	0
3	7914	97166.6	0	0
4	16145	91268.4	0.0009997	0

Optimization finished: average change in the fitness value less than options.FunctionTolerance and



```
sol = struct with fields:
  x: 0.8123
  y: 12.3103
```

```
fval = 9.1268e+04
```

Nonlinear constraints cause `ga` to solve many subproblems at each iteration. As shown in both the plots and the iterative display, the solution process has few iterations. However, the Func-count column in the iterative display shows many function evaluations per iteration.

Unsupported Functions

If your objective or nonlinear constraint functions are not supported (see “Supported Operations for Optimization Variables and Expressions”), use `fcn2optimexpr` to convert them to a form suitable for the problem-based approach. For example, suppose that instead of the constraint $xy \geq 10$, you have the constraint $I_1(x) + I_1(y) \geq 10$, where $I_1(x)$ is the modified Bessel function `besseli(1, x)`. (The

Bessel functions are not supported functions.) Create this constraint using `fcn2optimexpr`. First, create an optimization expression for $I_1(x) + I_1(y)$.

```
bfun = fcn2optimexpr(@(t,u)besseli(1,t) + besseli(1,u),x,y);
```

Next, replace the constraint `cons2` with the constraint `bfun >= 10`.

```
prob.Constraints.cons2 = bfun >= 10;
```

Solve the problem. The solution is different because the constraint region is different.

```
[sol2,fval2] = solve(prob,"Solver","ga","Options",options)
```

Solving problem using ga.

Single objective optimization:

2 Variable(s)

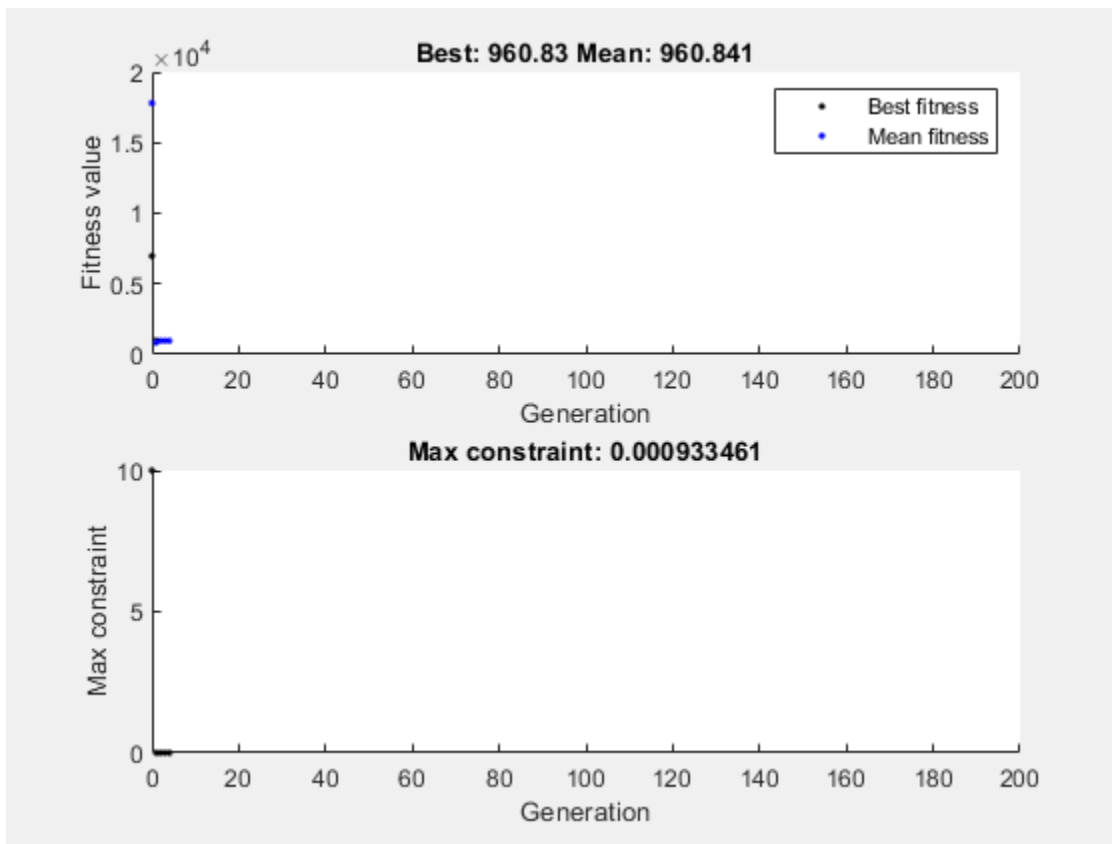
2 Nonlinear inequality constraint(s)

Options:

```
CreationFcn: @gacreationuniform
CrossoverFcn: @crossoverscattered
SelectionFcn: @selectionstochunif
MutationFcn: @mutationadaptfeasible
```

Generation	Func-count	Best f(x)	Max Constraint	Stall Generations
1	2512	974.044	0	0
2	4974	960.998	0	0
3	7436	963.12	0	0
4	12001	960.83	0.0009335	0

Optimization finished: average change in the fitness value less than options.FunctionTolerance and



```
sol2 = struct with fields:
  x: 0.4999
  y: 3.9979
```

```
fval2 = 960.8300
```

References

[1] Dixon, L. C. W., and G .P. Szego (eds.). *Towards Global Optimisation 2*. North-Holland: Elsevier Science Ltd., Amsterdam, 1978.

See Also

ga | solve | fcn2optimexpr

Related Examples

- “Genetic Algorithm”
- “Constrained Minimization Using Pattern Search, Problem-Based” on page 7-4
- “Minimize Rastrigin's Function” on page 8-4

Particle Swarm Optimization

- “What Is Particle Swarm Optimization?” on page 10-2
- “Optimize Function Using particleswarm, Problem-Based” on page 10-3
- “Optimize Using Particle Swarm” on page 10-5
- “Particle Swarm Output Function” on page 10-8
- “Particle Swarm Optimization Algorithm” on page 10-11
- “Tune Particle Swarm Optimization Process” on page 10-14

What Is Particle Swarm Optimization?

Particle swarm is a population-based algorithm. In this respect it is similar to the genetic algorithm. A collection of individuals called particles move in steps throughout a region. At each step, the algorithm evaluates the objective function at each particle. After this evaluation, the algorithm decides on the new velocity of each particle. The particles move, then the algorithm reevaluates.

The inspiration for the algorithm is flocks of birds or insects swarming. Each particle is attracted to some degree to the best location it has found so far, and also to the best location any member of the swarm has found. After some steps, the population can coalesce around one location, or can coalesce around a few locations, or can continue to move.

The `particleswarm` function attempts to optimize using a “Particle Swarm Optimization Algorithm” on page 10-11.

See Also

Related Examples

- “Optimize Using Particle Swarm” on page 10-5

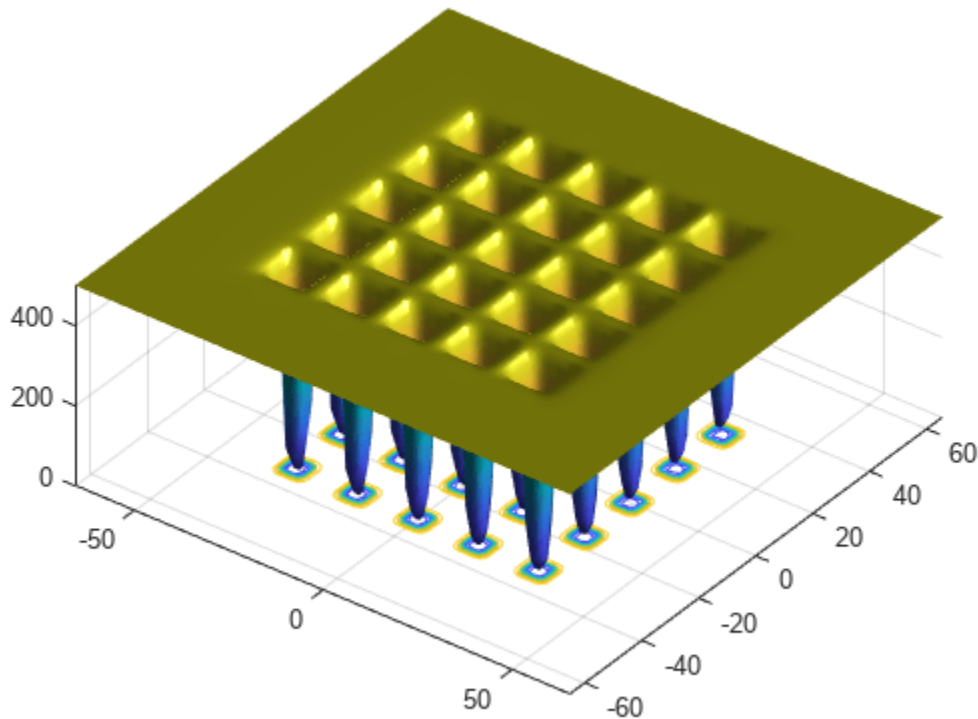
More About

- “Particle Swarm Optimization Algorithm” on page 10-11

Optimize Function Using particleswarm, Problem-Based

This example shows how to minimize a function using particle swarm in the problem-based approach when the objective is a function file, possibly of unknown content (a "black box" function). The function to minimize, `dejong5fcn(x)`, is included when you run this example.

```
dejong5fcn
```



Create a 2-D optimization variable `x`. The `dejong5fcn` function expects the variable to be a row vector, so specify `x` as a 2-element row vector.

```
x = optimvar("x",1,2);
```

To use `dejong5fcn` as the objective function, convert the function to an optimization expression using `fcn2optimexpr`.

```
fun = fcn2optimexpr(@dejong5fcn,x);
```

Create an optimization problem with the objective function `fun`.

```
prob = optimproblem("Objective",fun);
```

Set variable bounds from -50 to 50 in all components. When you specify scalar bounds, the software expands the bounds to all variables.

```
x.LowerBound = -50;  
x.UpperBound = 50;
```

Solve the problem, specifying the particleswarm solver.

```
rng default % For reproducibility  
[sol,fval] = solve(prob,"Solver","particleswarm")
```

Solving problem using particleswarm.

Optimization ended: relative change in the objective value over the last OPTIONS.MaxStallIterations iterations is less than OPTIONS.FunctionTolerance.

```
sol = struct with fields:  
  x: [-31.9751 -31.9719]
```

```
fval = 0.9980
```

See Also

particleswarm | fcn2optimexpr | solve

Related Examples

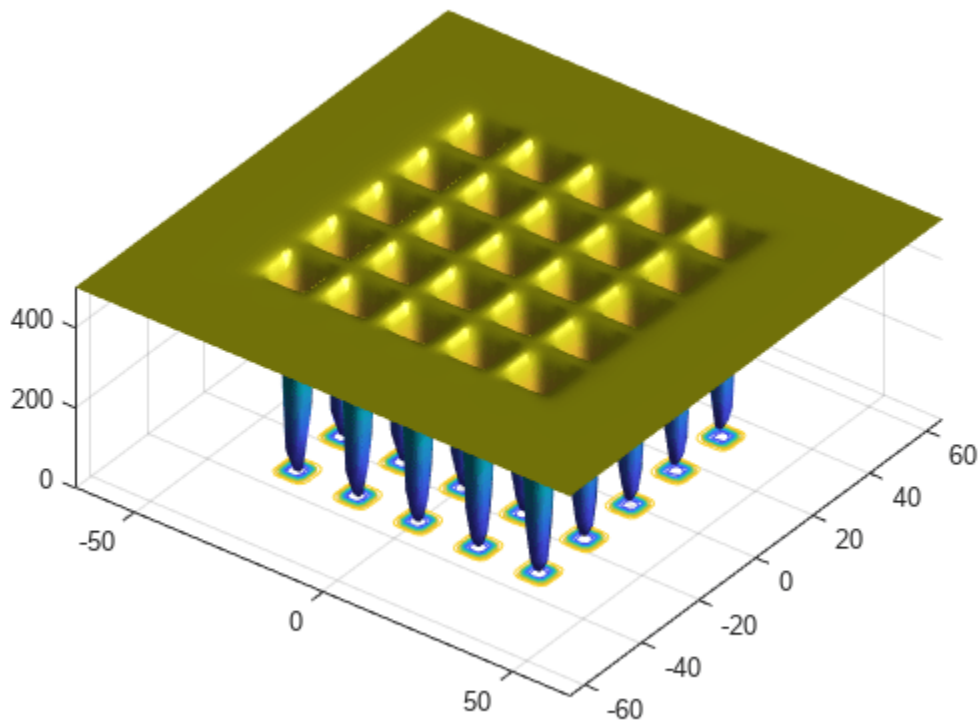
- “Particle Swarm”

Optimize Using Particle Swarm

This example shows how to optimize using the `particleswarm` solver.

The objective function in this example is De Jong's fifth function, which is available when you run this example.

```
dejong5fcn
```



This function has 25 local minima.

Try to find the minimum of the function using the default `particleswarm` settings.

```
fun = @dejong5fcn;
nvars = 2;
rng default % For reproducibility
[x,fval,exitflag] = particleswarm(fun,nvars)
```

```
Optimization ended: relative change in the objective value
over the last OPTIONS.MaxStallIterations iterations is less than OPTIONS.FunctionTolerance.
```

```
x = 1x2
```

```
    -31.9521    -16.0176
```

```
fval = 5.9288
```

```
exitflag = 1
```

Is the solution x the global optimum? It is unclear at this point. Looking at the function plot shows that the function has local minima for components in the range $[-50, 50]$. So restricting the range of the variables to $[-50, 50]$ helps the solver locate a global minimum.

```
lb = [-50;-50];  
ub = -lb;  
[x,fval,exitflag] = particleswarm(fun,nvars,lb,ub)
```

```
Optimization ended: relative change in the objective value  
over the last OPTIONS.MaxStallIterations iterations is less than OPTIONS.FunctionTolerance.
```

```
x = 1×2  
-16.0079 -31.9697
```

```
fval = 1.9920
```

```
exitflag = 1
```

This looks promising: the new solution has lower `fval` than the previous one. But is x truly a global solution? Try minimizing again with more particles, to better search the region.

```
options = optimoptions('particleswarm','SwarmSize',100);  
[x,fval,exitflag] = particleswarm(fun,nvars,lb,ub,options)
```

```
Optimization ended: relative change in the objective value  
over the last OPTIONS.MaxStallIterations iterations is less than OPTIONS.FunctionTolerance.
```

```
x = 1×2  
-31.9781 -31.9784
```

```
fval = 0.9980
```

```
exitflag = 1
```

This looks even more promising. But is this answer a global solution, and how accurate is it? Rerun the solver with a hybrid function. `particleswarm` calls the hybrid function after `particleswarm` finishes its iterations.

```
options.HybridFcn = @fmincon;  
[x,fval,exitflag] = particleswarm(fun,nvars,lb,ub,options)
```

```
Optimization ended: relative change in the objective value  
over the last OPTIONS.MaxStallIterations iterations is less than OPTIONS.FunctionTolerance.
```

```
x = 1×2  
-31.9783 -31.9784
```

```
fval = 0.9980
```

```
exitflag = 1
```

particleswarm found essentially the same solution as before. This gives you some confidence that particleswarm reports a local minimum and that the final x is the global solution.

See Also

More About

- “What Is Particle Swarm Optimization?” on page 10-2
- “Particle Swarm Optimization Algorithm” on page 10-11

Particle Swarm Output Function

This example shows how to use an output function for `particleswarm`. The output function plots the range that the particles occupy in each dimension.

An output function runs after each iteration of the solver. For syntax details, and for the data available to an output function, see the `particleswarm` options reference pages.

Custom Plot Function

This output function draws a plot with one line per dimension. Each line represents the range of the particles in the swarm in that dimension. The plot is log-scaled to accommodate wide ranges. If the swarm converges to a single point, then the range of each dimension goes to zero. But if the swarm does not converge to a single point, then the range stays away from zero in some dimensions.

Copy the following code into a file named `pswplotranges.m` on your MATLAB® path. The code sets up `nplot` subplots, where `nplot` is the number of dimensions in the problem.

```
function stop = pswplotranges(optimValues,state)

stop = false; % This function does not stop the solver
switch state
    case 'init'
        nplot = size(optimValues.swarm,2); % Number of dimensions
        for i = 1:nplot % Set up axes for plot
            subplot(nplot,1,i);
            tag = sprintf('psoplotrange_var_%g',i); % Set a tag for the subplot
            semilogy(optimValues.iteration,0,'-k','Tag',tag); % Log-scaled plot
            ylabel(num2str(i))
        end
        xlabel('Iteration','interp','none'); % Iteration number at the bottom
        subplot(nplot,1,1) % Title at the top
        title('Log range of particles by component')
        setappdata(gcf,'t0',tic); % Set up a timer to plot only when needed
    case 'iter'
        nplot = size(optimValues.swarm,2); % Number of dimensions
        for i = 1:nplot
            subplot(nplot,1,i);
            % Calculate the range of the particles at dimension i
            irange = max(optimValues.swarm(:,i)) - min(optimValues.swarm(:,i));
            tag = sprintf('psoplotrange_var_%g',i);
            plotHandle = findobj(get(gca,'Children'),'Tag',tag); % Get the subplot
            xdata = plotHandle.XData; % Get the X data from the plot
            newX = [xdata optimValues.iteration]; % Add the new iteration
            plotHandle.XData = newX; % Put the X data into the plot
            ydata = plotHandle.YData; % Get the Y data from the plot
            newY = [ydata irange]; % Add the new value
            plotHandle.YData = newY; % Put the Y data into the plot
        end
        if toc(getappdata(gcf,'t0')) > 1/30 % If 1/30 s has passed
            drawnow % Show the plot
            setappdata(gcf,'t0',tic); % Reset the timer
        end
    case 'done'
        % No cleanup necessary
end
```

```
end
```

Objective Function

The `multirosenbrock` function is a generalization of Rosenbrock's function to any even number of dimensions. It has a global minimum of 0 at the point $[1, 1, 1, 1, \dots]$.

```
function F = multirosenbrock(x)
% This function is a multidimensional generalization of Rosenbrock's
% function. It operates in a vectorized manner, assuming that x is a matrix
% whose rows are the individuals.

% Copyright 2014 by The MathWorks, Inc.

N = size(x,2); % assumes x is a row vector or 2-D matrix
if mod(N,2) % if N is odd
    error('Input rows must have an even number of elements')
end

odds = 1:2:N-1;
evens = 2:2:N;
F = zeros(size(x));
F(:,odds) = 1-x(:,odds);
F(:,evens) = 10*(x(:,evens)-x(:,odds).^2);
F = sum(F.^2,2);
```

Set Up and Run Problem

Set the `multirosenbrock` function as the objective function. Use four variables. Set a lower bound of -10 and an upper bound of 10 on each variable.

```
fun = @multirosenbrock;
nvar = 4; % A 4-D problem
lb = -10*ones(nvar,1); % Bounds to help the solver converge
ub = lb;
```

Set options to use the output function.

```
options = optimoptions(@particleswarm, 'OutputFcn', @pswplotranges);
```

Set the random number generator to get reproducible output. Then call the solver.

```
rng default % For reproducibility
[x, fval, eflag] = particleswarm(fun, nvar, lb, ub, options)
```

Optimization ended: relative change in the objective value over the last `OPTIONS.MaxStallIterations` iterations is less than `OPTIONS.FunctionTolerance`.

```
x =
```

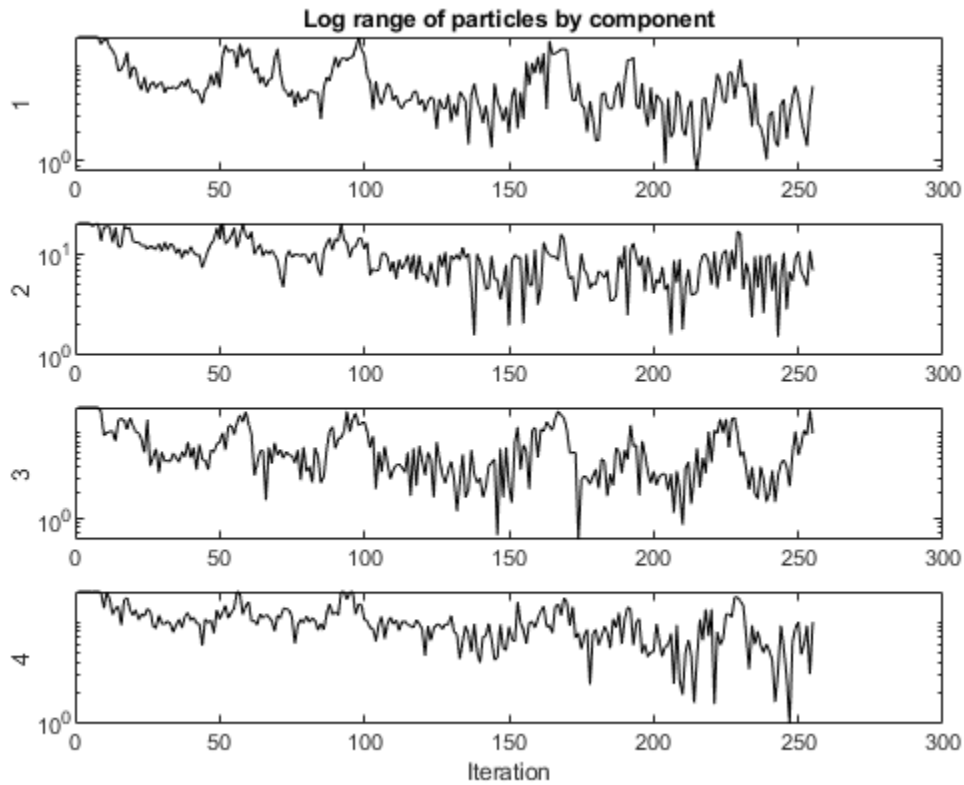
```
    0.9964    0.9930    0.9835    0.9681
```

```
fval =
```

```
    3.4935e-04
```

```
eflag =
```

```
1
```



Results

The solver returned a point near the optimum $[1, 1, 1, 1]$. But the span of the swarm did not converge to zero.

See Also

More About

- “Output Function and Plot Function” on page 17-48

Particle Swarm Optimization Algorithm

In this section...

“Algorithm Outline” on page 10-11

“Initialization” on page 10-11

“Iteration Steps” on page 10-12

“Stopping Criteria” on page 10-13

Algorithm Outline

`particleswarm` is based on the algorithm described in Kennedy and Eberhart [1], using modifications suggested in Mezura-Montes and Coello Coello [2] and in Pedersen [3].

The particle swarm algorithm begins by creating the initial particles, and assigning them initial velocities.

It evaluates the objective function at each particle location, and determines the best (lowest) function value and the best location.

It chooses new velocities, based on the current velocity, the particles' individual best locations, and the best locations of their neighbors.

It then iteratively updates the particle locations (the new location is the old one plus the velocity, modified to keep particles within bounds), velocities, and neighbors.

Iterations proceed until the algorithm reaches a stopping criterion.

Here are the details of the steps.

Initialization

By default, `particleswarm` creates particles at random uniformly within bounds. If there is an unbounded component, `particleswarm` creates particles with a random uniform distribution from -1000 to 1000. If you have only one bound, `particleswarm` shifts the creation to have the bound as an endpoint, and a creation interval 2000 wide. Particle i has position $x(i)$, which is a row vector with `nvars` elements. Control the span of the initial swarm using the `InitialSwarmSpan` option.

Similarly, `particleswarm` creates initial particle velocities v at random uniformly within the range $[-r, r]$, where r is the vector of initial ranges. The range of component k is $\min(\text{ub}(k) - \text{lb}(k), \text{InitialSwarmSpan}(k))$.

`particleswarm` evaluates the objective function at all particles. It records the current position $p(i)$ of each particle i . In subsequent iterations, $p(i)$ will be the location of the best objective function that particle i has found. And b is the best over all particles: $b = \min(\text{fun}(p(i)))$. d is the location such that $b = \text{fun}(d)$.

`particleswarm` initializes the neighborhood size N to $\text{minNeighborhoodSize} = \max(2, \text{floor}(\text{SwarmSize} * \text{MinNeighborsFraction}))$.

`particleswarm` initializes the inertia $W = \max(\text{InertiaRange})$, or if `InertiaRange` is negative, it sets $W = \min(\text{InertiaRange})$.

particleswarm initializes the stall counter $c = 0$.

For convenience of notation, set the variable $y1 = \text{SelfAdjustmentWeight}$, and $y2 = \text{SocialAdjustmentWeight}$, where $\text{SelfAdjustmentWeight}$ and $\text{SocialAdjustmentWeight}$ are options.

Iteration Steps

The algorithm updates the swarm as follows. For particle i , which is at position $x(i)$:

- 1 Choose a random subset S of N particles other than i .
- 2 Find $f_{\text{best}}(S)$, the best objective function among the neighbors, and $g(S)$, the position of the neighbor with the best objective function.
- 3 For $u1$ and $u2$ uniformly $(0,1)$ distributed random vectors of length n_{vars} , update the velocity

$$v = W*v + y1*u1.*(p-x) + y2*u2.*(g-x).$$

This update uses a weighted sum of:

- The previous velocity v
- The difference between the current position and the best position the particle has seen $p - x$
- The difference between the current position and the best position in the current neighborhood $g - x$

- 4 Update the position $x = x + v$.
- 5 Enforce the bounds. If any component of x is outside a bound, set it equal to that bound. For those components that were just set to a bound, if the velocity v of that component points outside the bound, set that velocity component to zero.
- 6 Evaluate the objective function $f = \text{fun}(x)$.
- 7 If $f < \text{fun}(p)$, then set $p = x$. This step ensures p has the best position the particle has seen.
- 8 The next steps of the algorithm apply to parameters of the entire swarm, not the individual particles. Consider the smallest $f = \min(f(j))$ among the particles j in the swarm.

If $f < b$, then set $b = f$ and $d = x$. This step ensures b has the best objective function in the swarm, and d has the best location.

- 9 If, in the previous step, the best function value was lowered, then set $\text{flag} = \text{true}$. Otherwise, $\text{flag} = \text{false}$. The value of flag is used in the next step.
- 10 Update the neighborhood. If $\text{flag} = \text{true}$:
 - a Set $c = \max(0, c-1)$.
 - b Set N to $\text{minNeighborhoodSize}$.
 - c If $c < 2$, then set $W = 2*W$.
 - d If $c > 5$, then set $W = W/2$.
 - e Ensure that W is in the bounds of the InertiaRange option.

If $\text{flag} = \text{false}$:

- a Set $c = c+1$.
- b Set $N = \min(N + \text{minNeighborhoodSize}, \text{SwarmSize})$.

Stopping Criteria

particleswarm iterates until it reaches a stopping criterion.

Stopping Option	Stopping Test	Exit Flag
MaxStallIterations and FunctionTolerance	Relative change in the best objective function value g over the last MaxStallIterations iterations is less than FunctionTolerance.	1
MaxIterations	Number of iterations reaches MaxIterations.	0
OutputFcn or PlotFcn	OutputFcn or PlotFcn can halt the iterations.	-1
ObjectiveLimit	Best objective function value g is less than ObjectiveLimit.	-3
MaxStallTime	Best objective function value g did not change in the last MaxStallTime seconds.	-4
MaxTime	Function run time exceeds MaxTime seconds.	-5

If particleswarm stops with exit flag 1, it optionally calls a hybrid function after it exits.

References

- [1] Kennedy, J., and R. Eberhart. "Particle Swarm Optimization." *Proceedings of the IEEE International Conference on Neural Networks*. Perth, Australia, 1995, pp. 1942-1945.
- [2] Mezura-Montes, E., and C. A. Coello Coello. "Constraint-handling in nature-inspired numerical optimization: Past, present and future." *Swarm and Evolutionary Computation*. 2011, pp. 173-194.
- [3] Pedersen, M. E. "Good Parameters for Particle Swarm Optimization." Luxembourg: Hvas Laboratories, 2010.

See Also

More About

- "What Is Particle Swarm Optimization?" on page 10-2
- "Optimize Using Particle Swarm" on page 10-5

Tune Particle Swarm Optimization Process

This example shows how to optimize using the `particleswarm` solver. The particle swarm algorithm moves a population of particles called a swarm toward a minimum of an objective function. The velocity of each particle in the swarm changes according to three factors:

- The effect of inertia (`InertiaRange` option)
- An attraction to the best location the particle has visited (`SelfAdjustmentWeight` option)
- An attraction to the best location among neighboring particles (`SocialAdjustmentWeight` option)

This example shows some effects of changing particle swarm options.

When to Modify Options

Often, `particleswarm` finds a good solution when using its default options. For example, it optimizes `rastriginsfcn` well with the default options. This function has many local minima, and a global minimum of 0 at the point $[0, 0]$.

```
rng default % for reproducibility
[x,fval,exitflag,output] = particleswarm(@rastriginsfcn,2);
```

```
Optimization ended: relative change in the objective value
over the last OPTIONS.MaxStallIterations iterations is less than OPTIONS.FunctionTolerance.
```

```
formatstring = 'particleswarm reached the value %f using %d function evaluations.\n';
fprintf(formatstring,fval,output.funccount)
```

```
particleswarm reached the value 0.000000 using 2560 function evaluations.
```

For this function, you know the optimal objective value, so you know that the solver found it. But what if you do not know the solution? One way to evaluate the solution quality is to rerun the solver.

```
[x,fval,exitflag,output] = particleswarm(@rastriginsfcn,2);
```

```
Optimization ended: relative change in the objective value
over the last OPTIONS.MaxStallIterations iterations is less than OPTIONS.FunctionTolerance.
```

```
fprintf(formatstring,fval,output.funccount)
```

```
particleswarm reached the value 0.000000 using 1480 function evaluations.
```

Both the solution and the number of function evaluations are similar to the previous run. This suggests that the solver is not having difficulty arriving at a solution.

Difficult Objective Function Using Default Parameters

The Rosenbrock function is well known to be a difficult function to optimize. This example uses a multidimensional version of the Rosenbrock function. The function has a minimum value of 0 at the point $[1, 1, 1, \dots]$.

```
rng default % for reproducibility
nvars = 6; % choose any even value for nvars
fun = @multirosenbrock;
[x,fval,exitflag,output] = particleswarm(fun,nvars);
```

Optimization ended: relative change in the objective value over the last `OPTIONS.MaxStallIterations` iterations is less than `OPTIONS.FunctionTolerance`.

```
fprintf(formatstring, fval, output.funccount)
```

particleswarm reached the value 3106.436648 using 12960 function evaluations.

The solver did not find a very good solution.

Bound the Search Space

Try bounding the space to help the solver locate a good point.

```
lb = -10*ones(1,nvars);
```

```
ub = -lb;
```

```
[xbounded, fvalbounded, exitflagbounded, outputbounded] = particleswarm(fun, nvars, lb, ub);
```

Optimization ended: relative change in the objective value over the last `OPTIONS.MaxStallIterations` iterations is less than `OPTIONS.FunctionTolerance`.

```
fprintf(formatstring, fvalbounded, outputbounded.funccount)
```

particleswarm reached the value 0.000006 using 71160 function evaluations.

The solver found a much better solution. But it took a very large number of function evaluations to do so.

Change Options

Perhaps the solver would converge faster if it paid more attention to the best neighbor in the entire space, rather than some smaller neighborhood.

```
options = optimoptions('particleswarm', 'MinNeighborsFraction', 1);
```

```
[xn, fvaln, exitflagn, outputn] = particleswarm(fun, nvars, lb, ub, options);
```

Optimization ended: relative change in the objective value over the last `OPTIONS.MaxStallIterations` iterations is less than `OPTIONS.FunctionTolerance`.

```
fprintf(formatstring, fvaln, outputn.funccount)
```

particleswarm reached the value 0.000462 using 30180 function evaluations.

While the solver took fewer function evaluations, it is unclear if this was due to randomness or to a better option setting.

Perhaps you should raise the `SelfAdjustmentWeight` option.

```
options.SelfAdjustmentWeight = 1.9;
```

```
[xn2, fvaln2, exitflagn2, outputn2] = particleswarm(fun, nvars, lb, ub, options);
```

Optimization ended: relative change in the objective value over the last `OPTIONS.MaxStallIterations` iterations is less than `OPTIONS.FunctionTolerance`.

```
fprintf(formatstring, fvaln2, outputn2.funccount)
```

particleswarm reached the value 0.000074 using 18780 function evaluations.

This time particleswarm took even fewer function evaluations. Is this improvement due to randomness, or are the option settings really worthwhile? Rerun the solver and look at the number of function evaluations.

```
[xn3,fvaln3,exitflagn3,outputn3] = particleswarm(fun,nvars,lb,ub,options);
```

```
Optimization ended: relative change in the objective value  
over the last OPTIONS.MaxStallIterations iterations is less than OPTIONS.FunctionTolerance.
```

```
fprintf(formatstring,fvaln3,outputn3.funccount)
```

```
particleswarm reached the value 0.157026 using 53040 function evaluations.
```

This time the number of function evaluations increased. Apparently, this `SelfAdjustmentWeight` setting does not necessarily improve performance.

Provide an Initial Point

Perhaps `particleswarm` would do better if it started from a known point that is not too far from the solution. Try the origin. Give a few individuals at the same initial point. Their random velocities ensure that they do not remain together.

```
x0 = zeros(20,6); % set 20 individuals as row vectors  
options.InitialSwarmMatrix = x0; % the rest of the swarm is random  
[xn3,fvaln3,exitflagn3,outputn3] = particleswarm(fun,nvars,lb,ub,options);
```

```
Optimization ended: relative change in the objective value  
over the last OPTIONS.MaxStallIterations iterations is less than OPTIONS.FunctionTolerance.
```

```
fprintf(formatstring,fvaln3,outputn3.funccount)
```

```
particleswarm reached the value 0.039015 using 32100 function evaluations.
```

The number of function evaluations decreased again.

Vectorize for Speed

The `multirosenbrock` function allows for vectorized function evaluation. This means that it can simultaneously evaluate the objective function for all particles in the swarm. This usually speeds up the solver considerably.

```
rng default % do a fair comparison  
options.UseVectorized = true;  
tic  
[xv,fvalv,exitflagv,outputv] = particleswarm(fun,nvars,lb,ub,options);
```

```
Optimization ended: relative change in the objective value  
over the last OPTIONS.MaxStallIterations iterations is less than OPTIONS.FunctionTolerance.
```

```
toc
```

```
Elapsed time is 0.306237 seconds.
```

```
options.UseVectorized = false;  
rng default  
tic  
[xnv,fvalnv,exitflagnv,outputnv] = particleswarm(fun,nvars,lb,ub,options);
```

```
Optimization ended: relative change in the objective value  
over the last OPTIONS.MaxStallIterations iterations is less than OPTIONS.FunctionTolerance.
```

```
toc
```

```
Elapsed time is 0.781767 seconds.
```

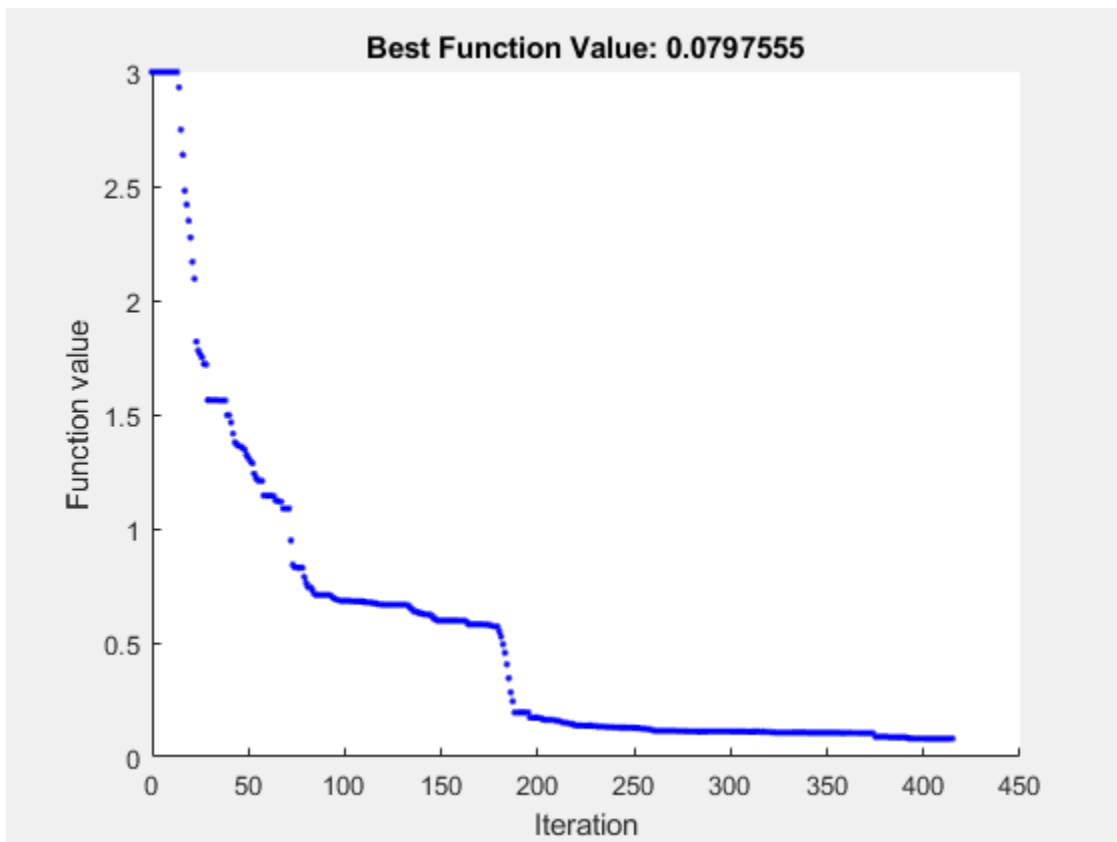
The vectorized calculation took about half the time of the serial calculation.

Plot Function

You can view the progress of the solver using a plot function.

```
options = optimoptions(options, 'PlotFcn', @pswplotbestf);
rng default
[x, fval, exitflag, output] = particleswarm(fun, nvars, lb, ub, options);
```

Optimization ended: relative change in the objective value over the last OPTIONS.MaxStallIterations iterations is less than OPTIONS.FunctionTolerance.



```
fprintf(formatstring, fval, output.funccount)
```

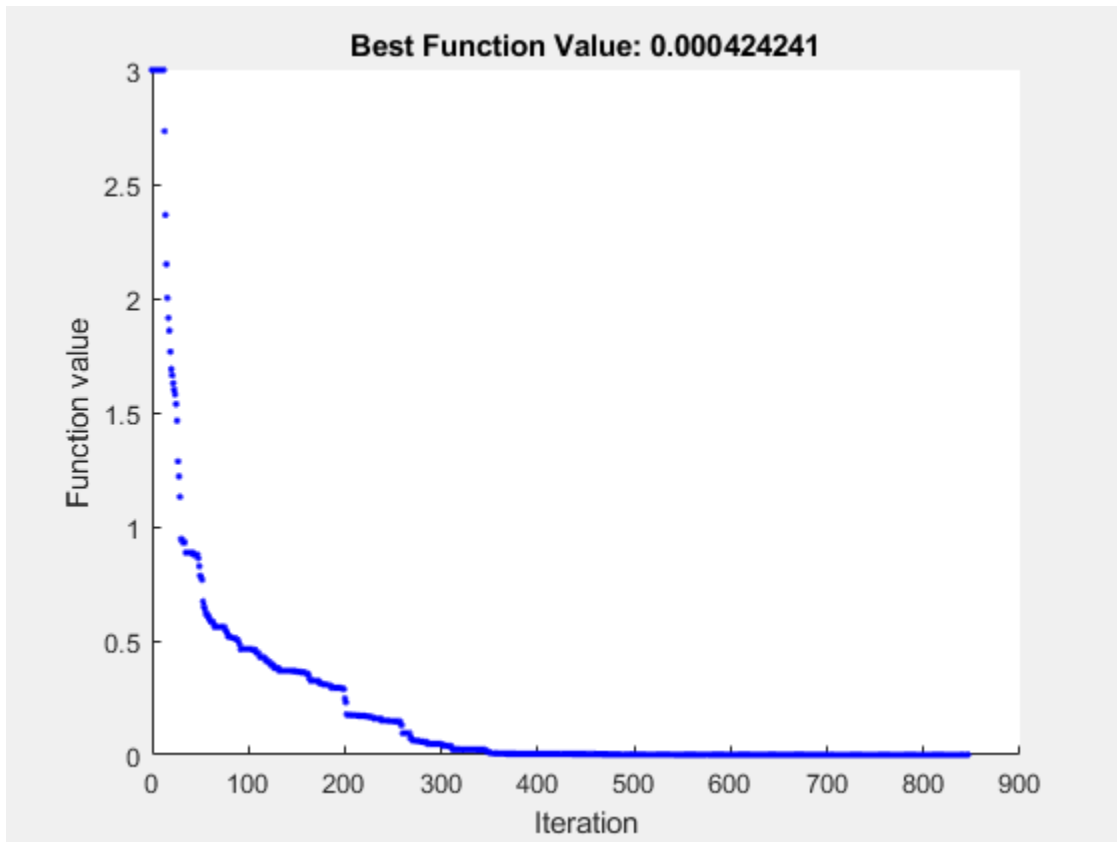
particleswarm reached the value 0.079755 using 24960 function evaluations.

Use More Particles

Frequently, using more particles obtains a more accurate solution.

```
rng default
options.SwarmSize = 200;
[x, fval, exitflag, output] = particleswarm(fun, nvars, lb, ub, options);
```

Optimization ended: relative change in the objective value over the last OPTIONS.MaxStallIterations iterations is less than OPTIONS.FunctionTolerance.



```
fprintf(formatstring,fval,output.funccount)
```

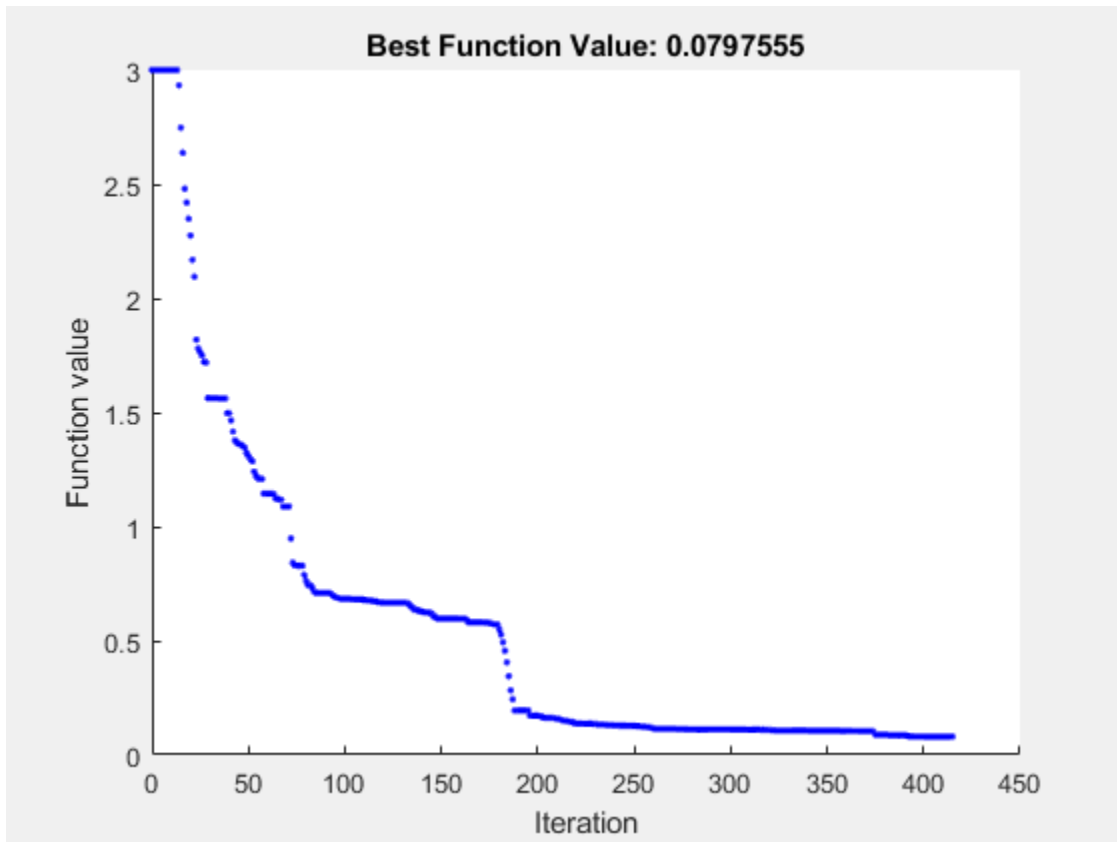
particleswarm reached the value 0.000424 using 169400 function evaluations.

Hybrid Function

particleswarm can search through several basins of attraction to arrive at a good local solution. Sometimes, though, it does not arrive at a sufficiently accurate local minimum. Try improving the final answer by specifying a hybrid function that runs after the particle swarm algorithm stops. Reset the number of particles to their original value, 60, to see the difference the hybrid function makes.

```
rng default
options.HybridFcn = @fmincon;
options.SwarmSize = 60;
[x,fval,exitflag,output] = particleswarm(fun,nvars,lb,ub,options);
```

Optimization ended: relative change in the objective value over the last OPTIONS.MaxStallIterations iterations is less than OPTIONS.FunctionTolerance.



```
fprintf(formatstring, fval, output.funccount)
```

```
particleswarm reached the value 0.000000 using 25191 function evaluations.
```

```
disp(output.hybridflag)
```

```
1
```

While the hybrid function improved the result, the plot function shows the same final value as before. This is because the plot function shows only the particle swarm algorithm iterations, and not the hybrid function calculations. The hybrid function caused the final function value to be very close to the true minimum value of 0. The `output.hybridflag` field shows that `fmincon` stops with exit flag 1, indicating that `x` is a true local minimum.

See Also

More About

- “Particle Swarm Options” on page 17-45

Surrogate Optimization

What Is Surrogate Optimization?

A surrogate is a function that approximates another function. The surrogate is useful because it takes little time to evaluate. So, for example, to search for a point that minimizes an objective function, simply evaluate its surrogate on thousands of points, and take the best value as an approximation to the minimizer of the objective function.

Surrogate optimization is best suited to time-consuming objective functions. The objective function need not be smooth, but the algorithm works best when the objective function is continuous.

Surrogate optimization attempts to find a global minimum of an objective function using few objective function evaluations. To do so, the algorithm tries to balance the optimization process between two goals: exploration and speed.

- Exploration to search for a global minimum.
- Speed to obtain a good solution in few objective function evaluations.

The algorithm has been proven to converge to a global solution for continuous objective functions on bounded domains. See Gutmann [1]. However, this convergence is not fast.

In general, there is no useful stopping criterion that stops the solver when it is near a global solution. Typically, you set a stopping criterion of a number of function evaluations or an amount of time, and take the best solution found within this computational budget.

For details of the `surrogateopt` algorithm, see “Surrogate Optimization Algorithm” on page 11-3.

References

[1] Gutmann, H.-M. *A radial basis function method for global optimization*. Journal of Global Optimization 19, Issue 3, 2001, pp. 201-227. <https://doi.org/10.1023/A:1011255519438>

See Also

`surrogateopt`

More About

- “Surrogate Optimization Algorithm” on page 11-3
- “Compare Surrogate Optimization with Other Solvers” on page 11-31

Surrogate Optimization Algorithm

In this section...

- “Serial surrogateopt Algorithm” on page 11-3
- “Mixed-Integer surrogateopt Algorithm” on page 11-8
- “Linear Constraint Handling” on page 11-9
- “surrogateopt Algorithm with Nonlinear Constraints” on page 11-9
- “Parallel surrogateopt Algorithm” on page 11-10
- “Parallel Mixed-Integer surrogateopt Algorithm” on page 11-10

Serial surrogateopt Algorithm

- “Serial surrogateopt Algorithm Overview” on page 11-3
- “Definitions for Surrogate Optimization” on page 11-3
- “Construct Surrogate Details” on page 11-4
- “Search for Minimum Details” on page 11-5
- “Merit Function Definition” on page 11-7

Serial surrogateopt Algorithm Overview

The surrogate optimization algorithm alternates between two phases.

- **Construct Surrogate** — Create `options.MinSurrogatePoints` random points within the bounds. Evaluate the (expensive) objective function at these points. Construct a surrogate of the objective function by interpolating a radial basis function through these points.
- **Search for Minimum** — Search for a minimum of the objective function by sampling several thousand random points within the bounds. Evaluate a merit function based on the surrogate value at these points and on the distances between them and points where the (expensive) objective function has been evaluated. Choose the best point as a candidate, as measured by the merit function. Evaluate the objective function at the best candidate point. This point is called an adaptive point. Update the surrogate using this value and search again.

During the Construct Surrogate phase, the algorithm constructs sample points from a quasirandom sequence. Constructing an interpolating radial basis function takes at least `nvars + 1` sample points, where `nvars` is the number of problem variables. The default value of `options.MinSurrogatePoints` is $2 * nvars$ or 20, whichever is larger.

The algorithm stops the Search for Minimum phase when all the search points are too close (less than the option `MinSampleDistance`) to points where the objective function was previously evaluated. See “Search for Minimum Details” on page 11-5. This switch from the Search for Minimum phase is called surrogate reset.

Definitions for Surrogate Optimization

The surrogate optimization algorithm description uses the following definitions.

- **Current** — The point where the objective function was evaluated most recently.
- **Incumbent** — The point with the smallest objective function value among all evaluated since the most recent surrogate reset.

- **Best** — The point with the smallest objective function value among all evaluated so far.
- **Initial** — The points, if any, that you pass to the solver in the `InitialPoints` option.
- **Random points** — Points in the Construct Surrogate phase where the solver evaluates the objective function. Generally, the solver takes these points from a quasirandom sequence, scaled and shifted to remain within the bounds. A quasirandom sequence is similar to a pseudorandom sequence such as `rand` returns, but is more evenly spaced. See https://en.wikipedia.org/wiki/Low-discrepancy_sequence. However, when the number of variables is above 500, the solver takes points from a Latin hypercube sequence. See https://en.wikipedia.org/wiki/Latin_hypercube_sampling.
- **Adaptive points** — Points in the Search for Minimum phase where the solver evaluates the objective function.
- **Merit function** — See “Merit Function Definition” on page 11-7.
- **Evaluated points** — All points at which the objective function value is known. These points include initial points, Construct Surrogate points, and Search for Minimum points at which the solver evaluates the objective function.
- **Sample points**. Pseudorandom points where the solver evaluates the merit function during the Search for Minimum phase. These points are not points at which the solver evaluates the objective function, except as described in “Search for Minimum Details” on page 11-5.

Construct Surrogate Details

To construct the surrogate, the algorithm chooses quasirandom points within the bounds. If you pass an initial set of points in the `InitialPoints` option, the algorithm uses those points and new quasirandom points (if necessary) to reach a total of `options.MinSurrogatePoints`.

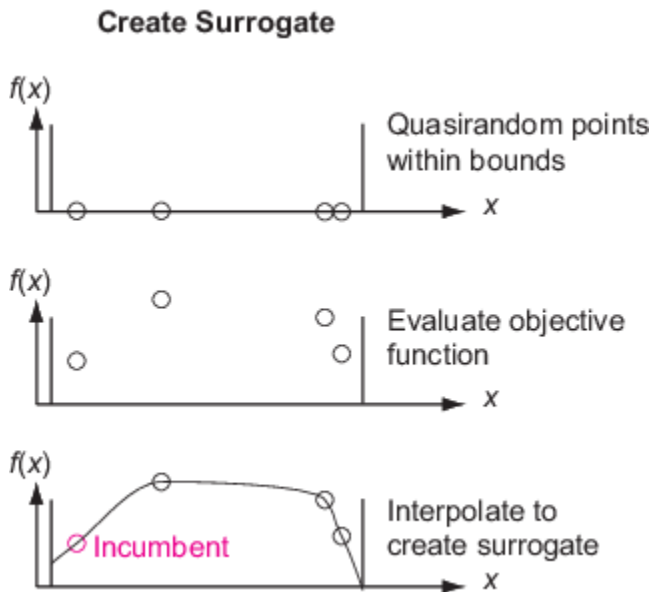
When `BatchUpdateInterval > 1`, the minimum number of random sample points used to create a surrogate is the larger of `MinSurrogatePoints` and `BatchUpdateInterval`.

Note If some upper bounds and lower bounds are equal, `surrogateopt` removes those “fixed” variables from the problem before constructing a surrogate. `surrogateopt` manages the variables seamlessly. So, for example, if you pass initial points, pass the full set, including any fixed variables.

On subsequent Construct Surrogate phases, the algorithm uses `options.MinSurrogatePoints` quasirandom points. The algorithm evaluates the objective function at these points.

The algorithm constructs a surrogate as an interpolation of the objective function by using a radial basis function (RBF) interpolator. RBF interpolation has several convenient properties that make it suitable for constructing a surrogate:

- An RBF interpolator is defined using the same formula in any number of dimensions and with any number of points.
- An RBF interpolator takes the prescribed values at the evaluated points.
- Evaluating an RBF interpolator takes little time.
- Adding a point to an existing interpolation takes relatively little time.
- Constructing an RBF interpolator involves solving an N-by-N linear system of equations, where N is the number of surrogate points. As Powell [1] showed, this system has a unique solution for many RBFs.
- `surrogateopt` uses a cubic RBF with a linear tail. This RBF minimizes a measure of bumpiness. See Gutmann [4].



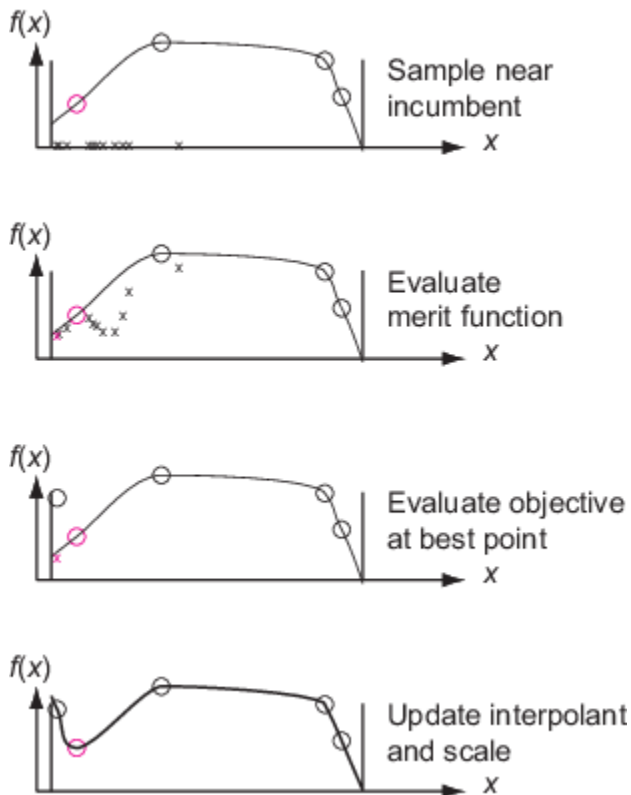
The algorithm uses only initial points and random points in the first Construct Surrogate phase, and uses only random points in subsequent Construct Surrogate phases. In particular, the algorithm does not use any adaptive points from the Search for Minimum phase in this surrogate.

Search for Minimum Details

The solver searches for a minimum of the objective function by following a procedure that is related to local search. The solver initializes a scale for the search with the value 0.2. The scale is like a search region radius or the mesh size in a pattern search. The solver starts from the incumbent point, which is the point with the smallest objective function value since the last surrogate reset. The solver searches for a minimum of a merit function that relates to both the surrogate and to a distance from existing search points, to try to balance minimizing the surrogate and searching the space. See "Merit Function Definition" on page 11-7.

The solver adds hundreds or thousands of pseudorandom vectors with scaled length to the incumbent point to obtain sample points. For details, see the Sampler Cycle table and surrounding discussion. These vectors are shifted and scaled by the bounds in each dimension, and multiplied by the scale. If necessary, the solver alters the sample points so that they stay within the bounds. The solver evaluates the merit function at the sample points, but not at any point within `options.MinSampleDistance` of a previously evaluated point. The point with the lowest merit function value is called the adaptive point. The solver evaluates the objective function value at the adaptive point, and updates the surrogate with this value. If the objective function value at the adaptive point is sufficiently lower than the incumbent value, then the solver deems the search successful and sets the adaptive point as the incumbent. Otherwise, the solver deems the search unsuccessful and does not change the incumbent.

Search for Minimum



The solver changes the scale when the first of these conditions is met:

- Three successful searches occur since the last scale change. In this case, the scale is doubled, up to a maximum scale length of 0.8 times the size of the box specified by the bounds.
- $\max(5, nvar)$ unsuccessful searches occur since the last scale change, where $nvar$ is the number of problem variables. In this case, the scale is halved, down to a minimum scale length of $1e-5$ times the size of the box specified by the bounds.

In this way, the random search eventually concentrates near an incumbent point that has a small objective function value. Then the solver geometrically reduces the scale toward the minimum scale length.

`surrogateopt` uses three different methods of sampling random points to locate a minimum of the merit function. `surrogateopt` chooses the sampler in a cycle associated with the weights according to the following table.

Sampler Cycle

Weight	0.3	0.5	0.8	0.95
Sampler	Random	Random	OrthoMADS	GPS

- **Scale** — Each sampler samples points within a scaled region around the incumbent. Any integer points have a scale that starts at $\frac{1}{2}$ times the width of the bounds, and adjusts exactly as the non-integer points, except that the width is increased to 1 if it would ever fall below 1.

- **Random** — The sampler generates integer points uniformly at random within a scale, centered at the incumbent. The sampler generates continuous points according to a Gaussian with mean zero from the incumbent. The width of the samples of any integer points is multiplied by the scale, as is the standard deviation of the continuous points.
- **OrthoMADS** — The sampler chooses an orthogonal coordinate system uniformly at random. The algorithm then creates sample points around the incumbent, adding and subtracting the current scale times each unit vector in the coordinate system. This creates $2N$ samples (unless some integer points are rounded to the incumbent), where N is the number of problem dimensions. OrthoMADS also uses two more points than the $2N$ fixed directions, one at $[+1, +1, \dots, +1]$, and the other at $[-1, -1, \dots, -1]$, for a total of $2N+2$ points. Then the sampler repeatedly halves the $2N + 2$ steps, creating a finer and finer set of points around the incumbent. This process ends when either there are enough samples. The points that should be integer-valued are then rounded to the nearest feasible integer points. The OrthoMADS algorithm is based on the paper [6]. That paper uses a quasirandom set of numbers for generating the coordinate system. In contrast, `surrogateopt` uses standard MATLAB pseudorandom numbers, with the orthogonal coordinates generated by `qr`.
- **GPS (Generalized Pattern Search)** — The sampler is like OrthoMADS, except instead of choosing a random set of directions, GPS uses the non-rotated coordinate system. The GPS algorithm is based on the paper [5]. This algorithm is described in detail in “How Pattern Search Polling Works” on page 6-27.

The solver does not evaluate the merit function at points within `options.MinSampleDistance` of an evaluated point (see “Definitions for Surrogate Optimization” on page 11-3). The solver switches from the Search for Minimum phase to a Construct Surrogate phase (in other words, performs a surrogate reset) when all sample points are within `MinSampleDistance` of evaluated points. Generally, this reset occurs after the solver reduces the scale so that all sample points are tightly clustered around the incumbent.

When the `BatchUpdateInterval` option is larger than 1, the solver generates `BatchUpdateInterval` adaptive points before updating the surrogate model or changing the incumbent. The update includes all of the adaptive points. Effectively, the algorithm does not use any new information until it generates `BatchUpdateInterval` adaptive points, and then the algorithm uses all the information to update the surrogate.

Merit Function Definition

The merit function $f_{\text{merit}}(x)$ is a weighted combination of two terms:

- **Scaled surrogate.** Define s_{\min} as the minimum surrogate value among the sample points, s_{\max} as the maximum, and $s(x)$ as the surrogate value at the point x . Then the scaled surrogate $S(x)$ is

$$S(x) = \frac{s(x) - s_{\min}}{s_{\max} - s_{\min}}.$$

$s(x)$ is nonnegative and is zero at points x that have minimal surrogate value among sample points.

- **Scaled distance.** Define $x_j, j = 1, \dots, k$ as the k evaluated points. Define d_{ij} as the distance from sample point i to evaluated point k . Set $d_{\min} = \min(d_{ij})$ and $d_{\max} = \max(d_{ij})$, where the minimum and maximum are taken over all i and j . The scaled distance $D(x)$ is

$$D(x) = \frac{d_{\max} - d(x)}{d_{\max} - d_{\min}},$$

where $d(x)$ is the minimum distance of the point x to an evaluated point. $D(x)$ is nonnegative and is zero at points x that are maximally far from evaluated points. So, minimizing $D(x)$ leads the algorithm to points that are far from evaluated points.

The merit function is a convex combination of the scaled surrogate and scaled distance. For a weight w with $0 < w < 1$, the merit function is

$$f_{\text{merit}}(x) = wS(x) + (1 - w)D(x).$$

A large value of w gives importance to the surrogate values, causing the search to minimize the surrogate. A small value of w gives importance to points that are far from evaluated points, leading the search to new regions. During the Search for Minimum phase, the weight w cycles through these four values, as suggested by Regis and Shoemaker [2]: 0.3, 0.5, 0.8, and 0.95.

Mixed-Integer surrogateopt Algorithm

- “Mixed-Integer surrogateopt Overview” on page 11-8
- “Algorithm Start” on page 11-8
- “Integer Search for Minimum” on page 11-8
- “Tree Search” on page 11-8

Mixed-Integer surrogateopt Overview

When some or all of the variables are integer, as specified in the `intcon` argument, `surrogateopt` changes some aspects of the “Serial surrogateopt Algorithm” on page 11-3. This description is mainly about the changes, rather than the entire algorithm.

Algorithm Start

If necessary, `surrogateopt` moves the specified bounds for `intcon` points so that they are integers. Also, `surrogateopt` ensures that a supplied initial point is integer feasible and feasible with respect to bounds. The algorithm then generates quasirandom points as in the non-integer algorithm, rounding integer points to integer values. The algorithm generates a surrogate exactly as in the non-integer algorithm.

Integer Search for Minimum

This portion of the algorithm is the same as in “Search for Minimum Details” on page 11-5. The modifications for integer constraints appear in that section.

Tree Search

After sampling hundreds or thousands of values of the merit function, `surrogateopt` usually chooses the minimal point, and evaluates the objective function. However, under two circumstances, `surrogateopt` performs another search called a Tree Search before evaluating the objective:

- There have been $2N$ steps since the last Tree Search, called Case A.
- `surrogateopt` is about to perform a surrogate reset, called Case B.

The Tree Search proceeds as follows, similar to a procedure in `intlinprog`, as described in “Branch and Bound”. The algorithm makes a tree by finding an integer value and creating a new problem that has a bound on this value either one higher or one lower, and solving the subproblem with this new

bound. After solving the subproblem, the algorithm chooses a different integer to be bounded either above or below by one.

- Case A: Use the scaled sampling radius as the overall bounds, and run for up to 1000 steps of the search.
- Case B: Use the original problem bounds as the overall bounds, and run for up to 5000 steps of the search.

In this case, solving the subproblem means running the `fmincon 'sqp'` algorithm on the continuous variables, starting from the incumbent with the specified integer values, so search for a local minimum of the merit function.

Tree Search can be time-consuming. So `surrogateopt` has an internal iteration limit to avoid excessive time in this step, limiting both the number of Case A and Case B steps.

At the end of the Tree search, the algorithm takes the better of the point found by Tree Search and the point found by the preceding search for a minimum, as measured by the merit function. The algorithm evaluates the objective function at this point. The remainder of the integer algorithm is exactly the same as the continuous algorithm.

Linear Constraint Handling

When a problem has linear constraints, the algorithm modifies its search procedure in a way that keeps all points feasible with respect to these constraints and with respect to bounds at every iteration. During the construction and search phases, the algorithm creates only linearly feasible points by a procedure similar to the `patternsearch 'GSSPositiveBasis2N'` poll algorithm.

When a problem has integer constraints and linear constraints, the algorithm first creates linearly feasible points. Then the algorithm tries to satisfy integer constraints by a process of rounding linearly feasible points to integers using a heuristic that attempts to keep the points linearly feasible. When this process is unsuccessful in obtaining enough feasible points for constructing a surrogate, the algorithm calls `intlinprog` to attempt to find more points that are feasible with respect to bounds, linear constraints, and integer constraints.

surrogateopt Algorithm with Nonlinear Constraints

When the problem has nonlinear constraints, `surrogateopt` modifies the previously described algorithm in several ways.

Initially and after each surrogate reset, the algorithm creates surrogates of the objective and nonlinear constraint functions. Subsequently, the algorithm differs depending on whether or not the Construct Surrogate phase found any feasible points; finding a feasible point is equivalent to the incumbent point being feasible when the surrogate is constructed.

- Incumbent is infeasible — This case, called Phase 1, involves a search for a feasible point. In the Search for Minimum phase before encountering a feasible point, `surrogateopt` changes the definition of the merit function. The algorithm counts the number of constraints that are violated at each point, and considers only those points with the fewest number of violated constraints. Among those points, the algorithm chooses the point that minimizes the maximum constraint violation as the best (lowest merit function) point. This best point is the incumbent. Subsequently, until the algorithm encounters a feasible point, it uses this modification of the merit function. When `surrogateopt` evaluates a point and finds that it is feasible, the feasible point becomes the incumbent and the algorithm is in Phase 2.

- Incumbent is feasible — This case, called Phase 2, proceeds in the same way as the standard algorithm. The algorithm ignores infeasible points for the purpose of computing the merit function.

The algorithm proceeds according to the “Mixed-Integer surrogateopt Algorithm” on page 11-8 with the following changes. After every $2 \cdot nvars$ points where the algorithm evaluates the objective and nonlinear constraint functions, `surrogateopt` calls the `fmincon` function to minimize the surrogate, subject to the surrogate nonlinear constraints and bounds, where the bounds are scaled by the current scale factor. (If the incumbent is infeasible, `fmincon` ignores the objective and attempts to find a point satisfying the constraints.) If the problem has both integer and nonlinear constraints, then `surrogateopt` calls “Tree Search” on page 11-8 instead of `fmincon`.

If the problem is a feasibility problem, meaning it has no objective function, then `surrogateopt` performs a surrogate reset immediately after it finds a feasible point.

Parallel surrogateopt Algorithm

The parallel `surrogateopt` algorithm differs from the serial algorithm as follows:

- The parallel algorithm maintains a queue of points on which to evaluate the objective function. This queue is 30% larger than the number of parallel workers, rounded up. The queue is larger than the number of workers to minimize the chance that a worker is idle because no point is available to evaluate.
- The scheduler takes points from the queue in a FIFO fashion and assigns them to workers as they become idle, asynchronously.
- When the algorithm switches between phases (Search for Minimum and Construct Surrogate), the existing points being evaluated remain in service, and any other points in the queue are flushed (discarded from the queue). So, generally, the number of random points that the algorithm creates for the Construct Surrogate phase is at most `options.MinSurrogatePoints + PoolSize`, where `PoolSize` is the number of parallel workers. Similarly, after a surrogate reset, the algorithm still has `PoolSize - 1` adaptive points that its workers are evaluating.

Note Currently, parallel surrogate optimization does not necessarily give reproducible results, due to the nonreproducibility of parallel timing, which can lead to different execution paths.

Parallel Mixed-Integer surrogateopt Algorithm

When run in parallel on a mixed-integer problem, `surrogateopt` performs the Tree Search procedure on the host, not on the parallel workers. Using the latest surrogate, `surrogateopt` searches for a smaller value of the surrogate after each worker returns with an adaptive point.

If the objective function is not expensive (time-consuming), then this Tree Search procedure can be a bottleneck on the host. In contrast, if the objective function is expensive, then the Tree Search procedure takes a small fraction of the overall computational time, and is not a bottleneck.

References

- [1] Powell, M. J. D. *The Theory of Radial Basis Function Approximation in 1990*. In Light, W. A. (editor), *Advances in Numerical Analysis, Volume 2: Wavelets, Subdivision Algorithms, and Radial Basis Functions*. Clarendon Press, 1992, pp. 105–210.

- [2] Regis, R. G., and C. A. Shoemaker. *A Stochastic Radial Basis Function Method for the Global Optimization of Expensive Functions*. INFORMS J. Computing 19, 2007, pp. 497-509.
- [3] Wang, Y., and C. A. Shoemaker. *A General Stochastic Algorithm Framework for Minimizing Expensive Black Box Objective Functions Based on Surrogate Models and Sensitivity Analysis*. arXiv:1410.6271v1 (2014). Available at <https://arxiv.org/pdf/1410.6271>.
- [4] Gutmann, H.-M. *A Radial Basis Function Method for Global Optimization*. Journal of Global Optimization 19, March 2001, pp. 201-227.
- [5] Audet, Charles, and J. E. Dennis Jr. "Analysis of Generalized Pattern Searches." *SIAM Journal on Optimization*. Volume 13, Number 3, 2003, pp. 889-903.
- [6] Abramson, Mark A., Charles Audet, J. E. Dennis, Jr., and Sebastien Le Digabel. "ORTHOMADS: A deterministic MADS instance with orthogonal directions." *SIAM Journal on Optimization*. Volume 20, Number 2, 2009, pp. 948-966.

See Also

surrogateopt

More About

- "Interpret surrogateoptplot" on page 11-25
- "Surrogate Optimization"

External Websites

- https://en.wikipedia.org/wiki/Radial_basis_function

Surrogate Optimization of Multidimensional Function

This example shows the behavior of three recommended solvers on a minimization problem. The objective function is the `multirosenbrock` function:

```
type multirosenbrock

function F = multirosenbrock(x)
% This function is a multidimensional generalization of Rosenbrock's
% function. It operates in a vectorized manner, assuming that x is a matrix
% whose rows are the individuals.

% Copyright 2014 by The MathWorks, Inc.

N = size(x,2); % assumes x is a row vector or 2-D matrix
if mod(N,2) % if N is odd
    error('Input rows must have an even number of elements')
end

odds = 1:2:N-1;
evens = 2:2:N;
F = zeros(size(x));
F(:,odds) = 1-x(:,odds);
F(:,evens) = 10*(x(:,evens)-x(:,odds).^2);
F = sum(F.^2,2);
```

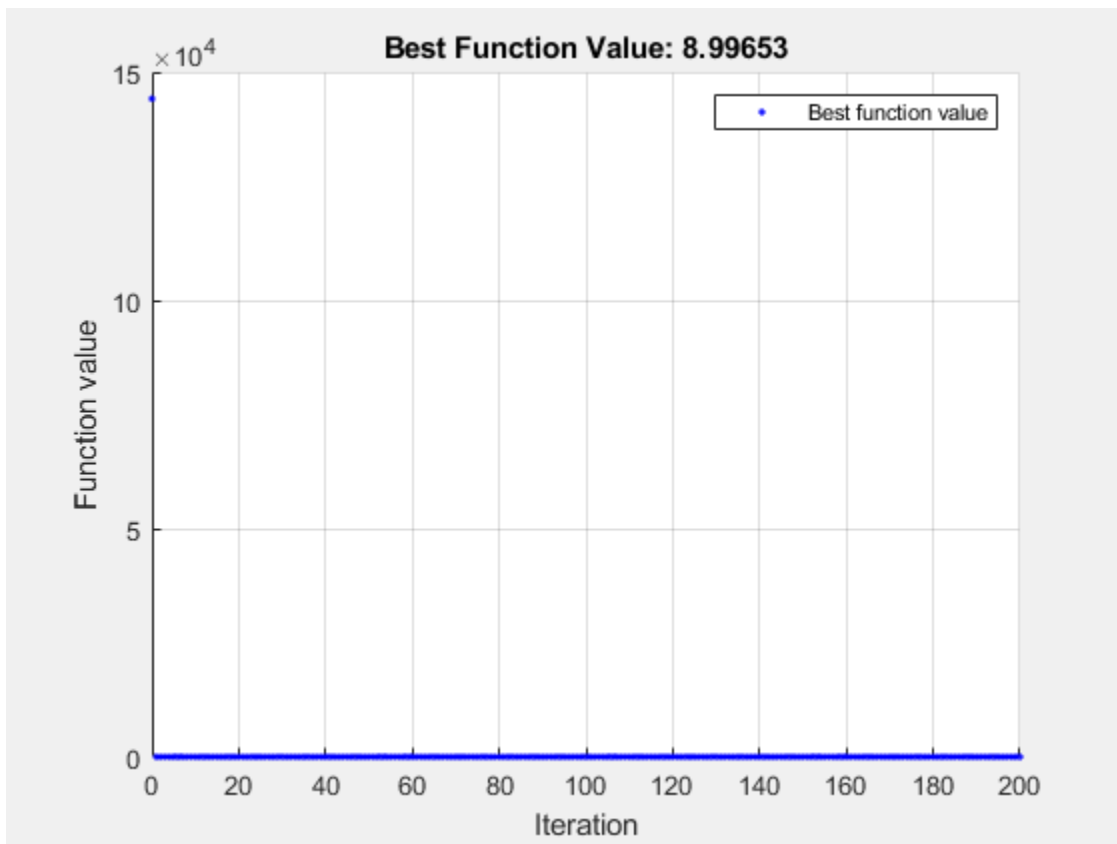
The `multirosenbrock` function has a single local minimum of 0 at the point $[1, 1, \dots, 1]$. See how well the three best solvers for general nonlinear problems work on this function in 20 dimensions with a challenging maximum function count of only 200.

Set up the problem.

```
N = 20; % any even number
mf = 200; % max fun evals
fun = @multirosenbrock;
lb = -3*ones(1,N);
ub = -lb;
rng default
x0 = -3*rand(1,N);
```

Set options for `surrogateopt` to use only 200 function evaluations, and then run the solver.

```
options = optimoptions('surrogateopt','MaxFunctionEvaluations',mf);
[xm,fvalm,~,~,pop] = surrogateopt(fun,lb,ub,options);
```

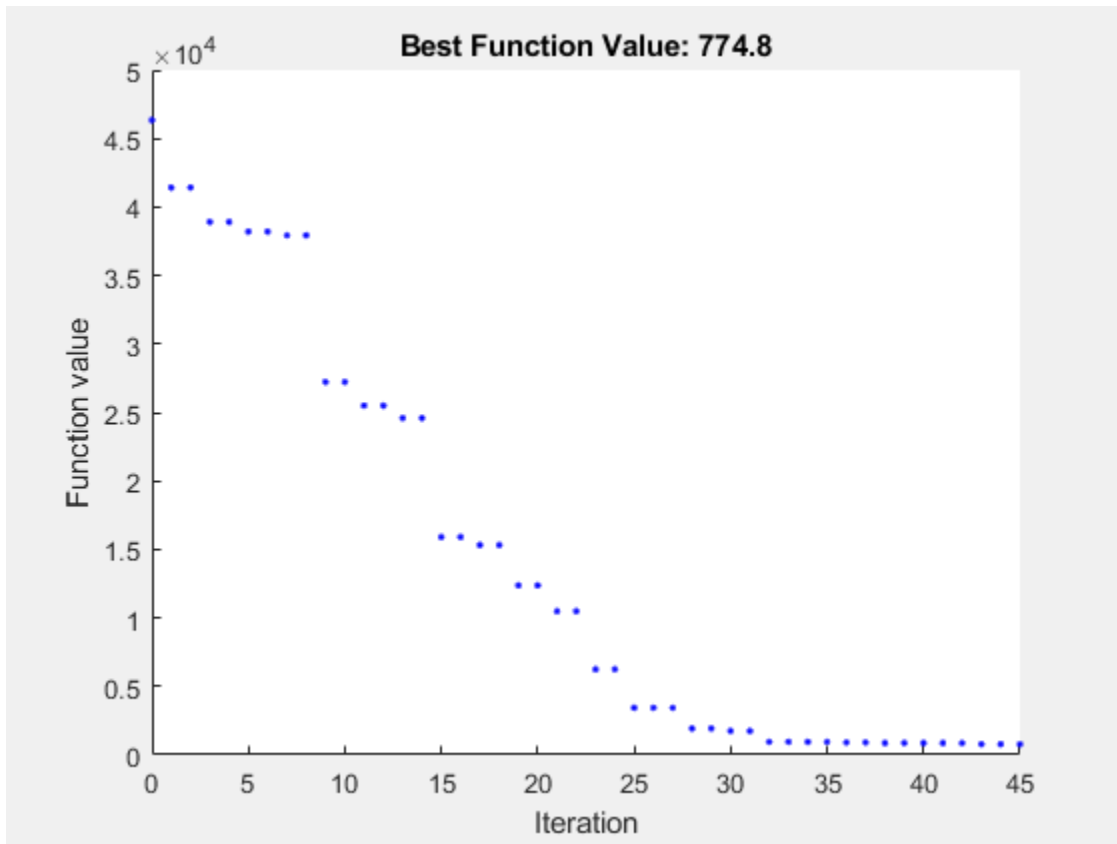


surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

Set similar options for patternsearch, including a plot function to monitor the optimization.

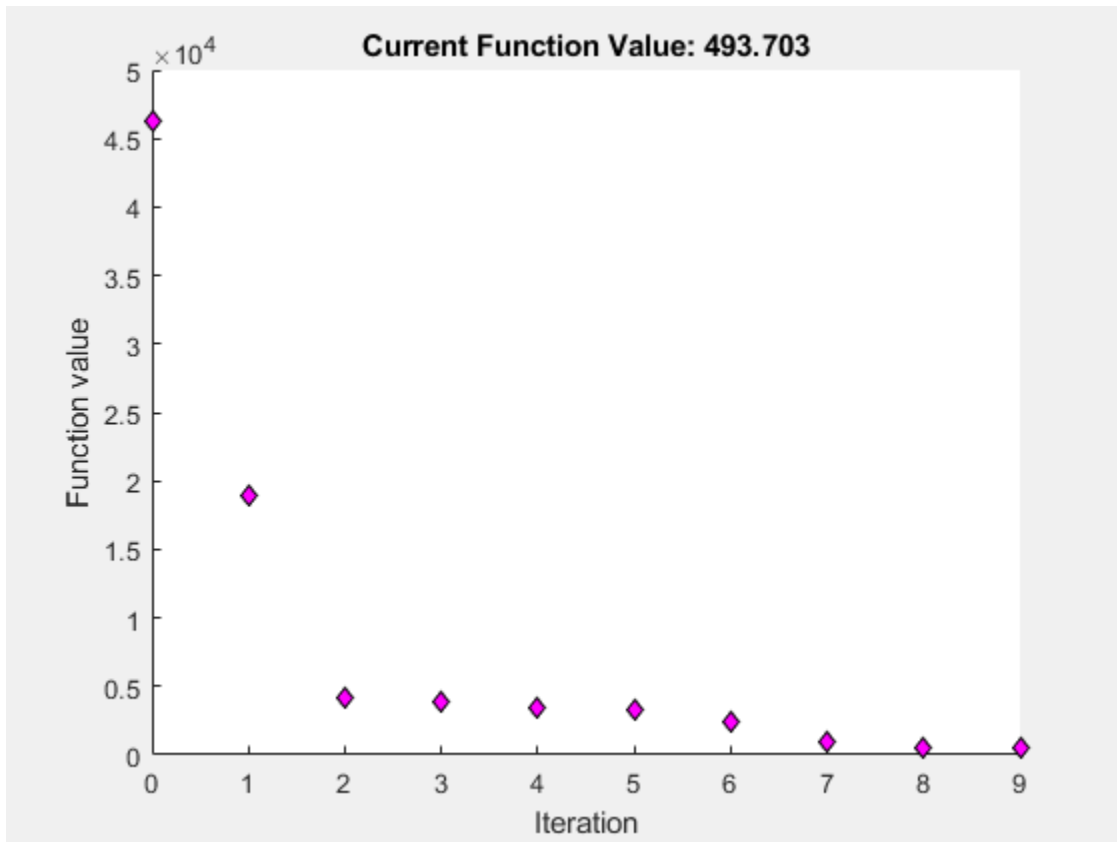
```
psopts = optimoptions('patternsearch','PlotFcn','psplotbestf','MaxFunctionEvaluations',mf);
[psol,pfval] = patternsearch(fun,x0,[],[],[],[],lb,ub,[],psopts);
```

Maximum number of function evaluations exceeded: increase options.MaxFunctionEvaluations.



Set similar options for `fmincon`.

```
opts = optimoptions('fmincon','PlotFcn','optimplotfval','MaxFunctionEvaluations',mf);  
[fmsol, fmfval, eflag, foutput] = fmincon(fun, x0, [], [], [], [], lb, ub, [], opts);
```



Solver stopped prematurely.

fmincon stopped because it exceeded the function evaluation limit,
`options.MaxFunctionEvaluations = 2.000000e+02`.

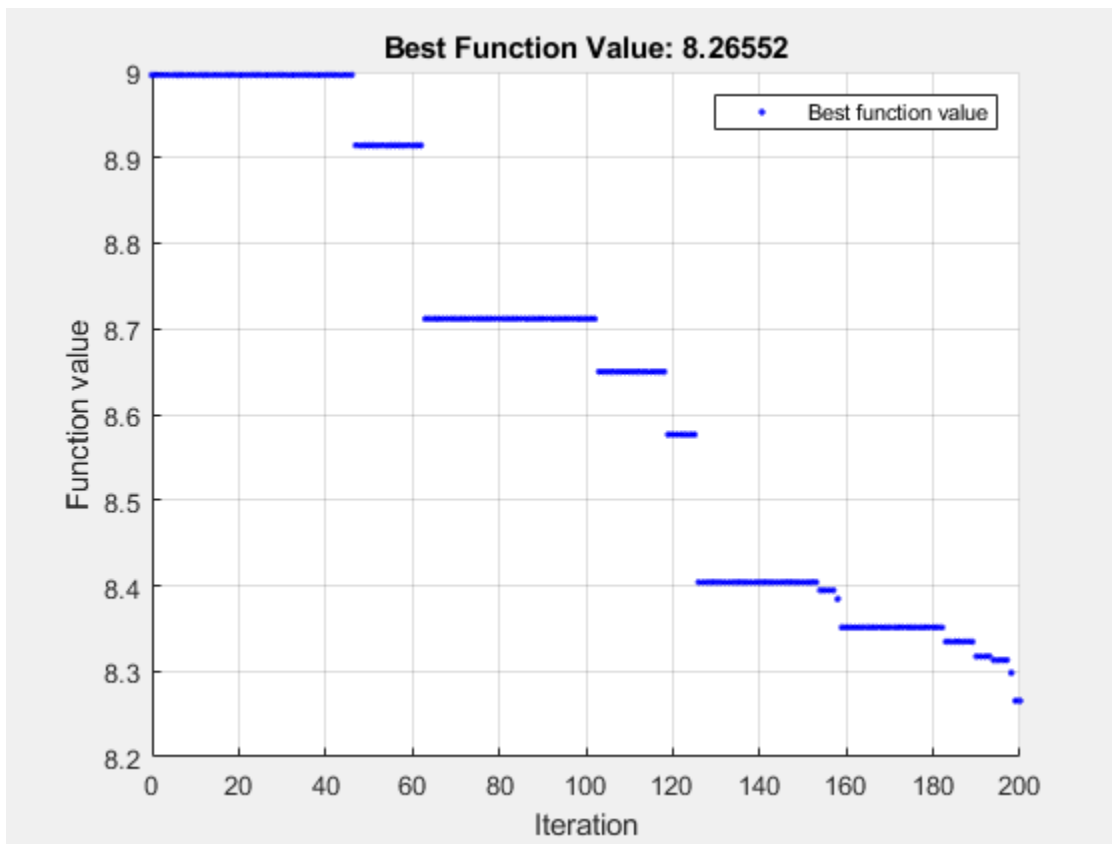
For this extremely restricted number of function evaluations, the `surrogateopt` solution is closest to the true minimum value of 0.

```
table(fvalm,pfval,fmfval,'VariableNames',{'surrogateopt','patternsearch','fmincon'})
```

```
ans=1x3 table
    surrogateopt    patternsearch    fmincon
    _____    _____    _____
           8.9965           774.8           493.7
```

Allowing another 200 function evaluations shows that the other solvers rapidly approach the true solution, while `surrogateopt` does not improve significantly. Restart the solvers from their previous solutions, which adds 200 function evaluations to each optimization.

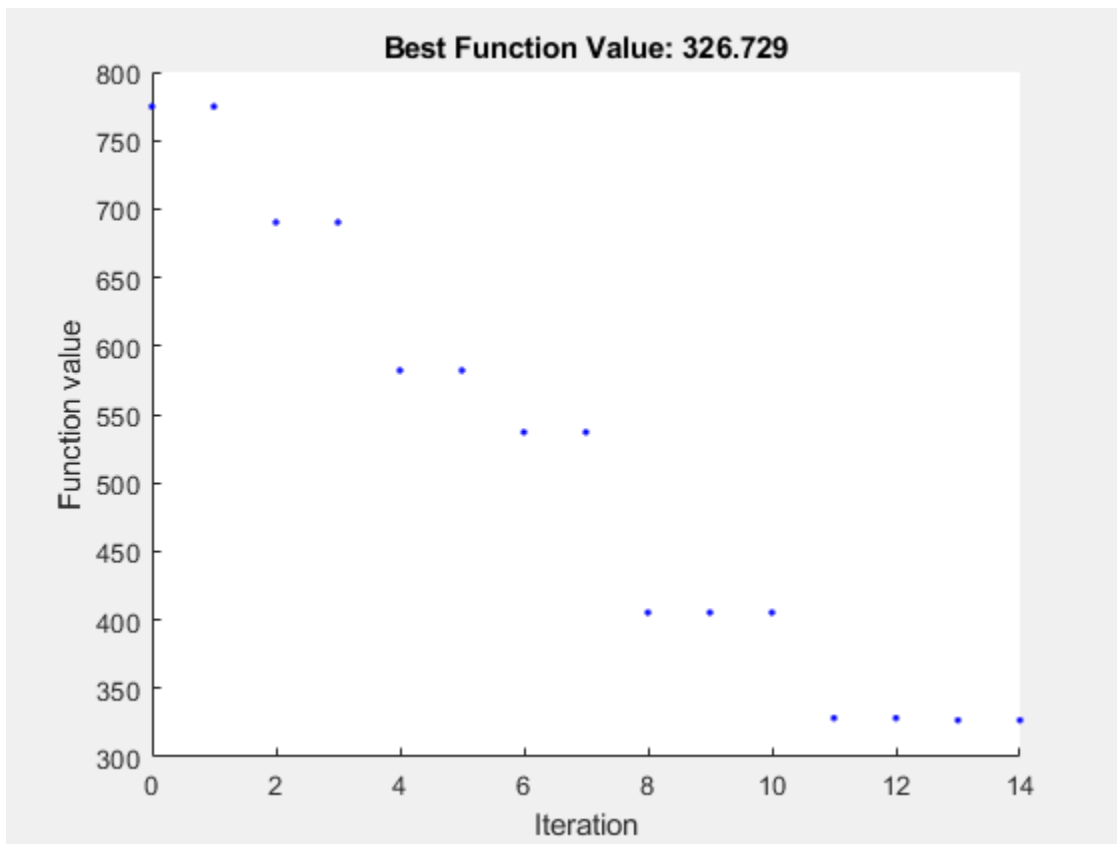
```
options = optimoptions(options,'InitialPoints',pop);
[xm,fvalm,~,~,pop] = surrogateopt(fun,lb,ub,options);
```



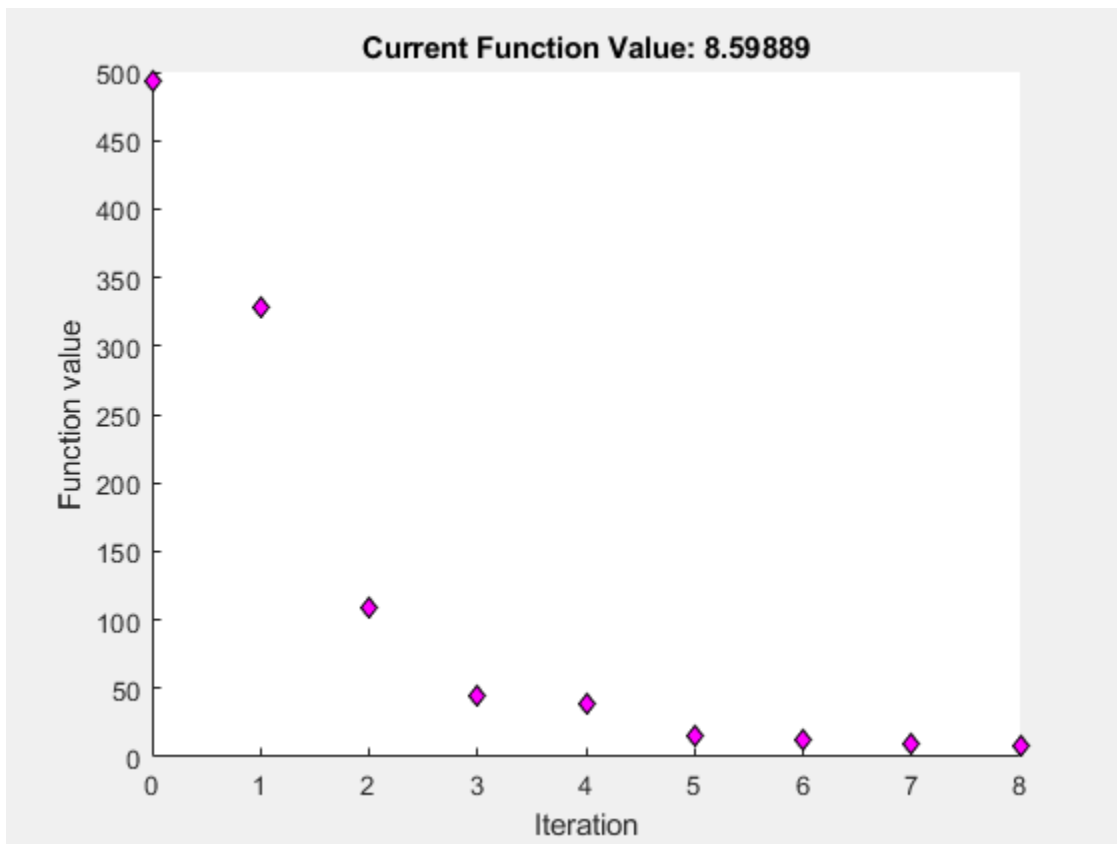
surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
[psol,pfval] = patternsearch(fun,psol,[],[],[],[],lb,ub,[],psopts);
```

Maximum number of function evaluations exceeded: increase options.MaxFunctionEvaluations.



```
[fmsol,fmval,eflag,fmoutput] = fmincon(fun,fmsol,[],[],[],[],lb,ub,[],opts);
```



Solver stopped prematurely.

fmincon stopped because it exceeded the function evaluation limit,
`options.MaxFunctionEvaluations = 2.000000e+02`.

```
table(fvalm,pfval,fmfval,'VariableNames',{'surrogateopt','patternsearch','fmincon'})
```

```
ans=1x3 table
   surrogateopt   patternsearch   fmincon
   _____   _____   _____
           8.2655           326.73           8.5989
```

See Also

surrogateopt

More About

- “Surrogate Optimization”

Modify surrogateopt Options

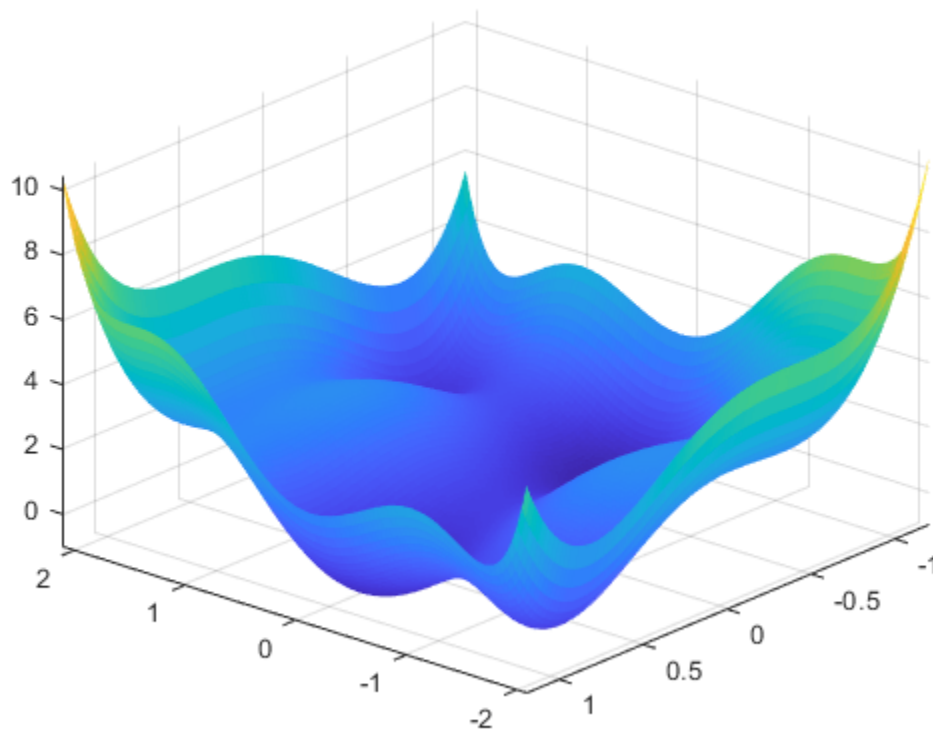
This example shows how to search for a global minimum by running `surrogateopt` on a two-dimensional problem that has six local minima. The example then shows how to modify some options to search more effectively.

Define the objective function `sixmin` as follows.

```
sixmin = @(x)(4*x(:,1).^2 - 2.1*x(:,1).^4 + x(:,1).^6/3 ...
    + x(:,1).*x(:,2) - 4*x(:,2).^2 + 4*x(:,2).^4);
```

Plot the function.

```
[X,Y] = meshgrid(linspace(-2.1,2.1),linspace(-1.2,1.2));
Z = sixmin([X(:),Y(:)]);
Z = reshape(Z,size(X));
surf(X,Y,Z, 'EdgeColor', 'none')
view(-139,31)
```



The function has six local minima and two global minima.

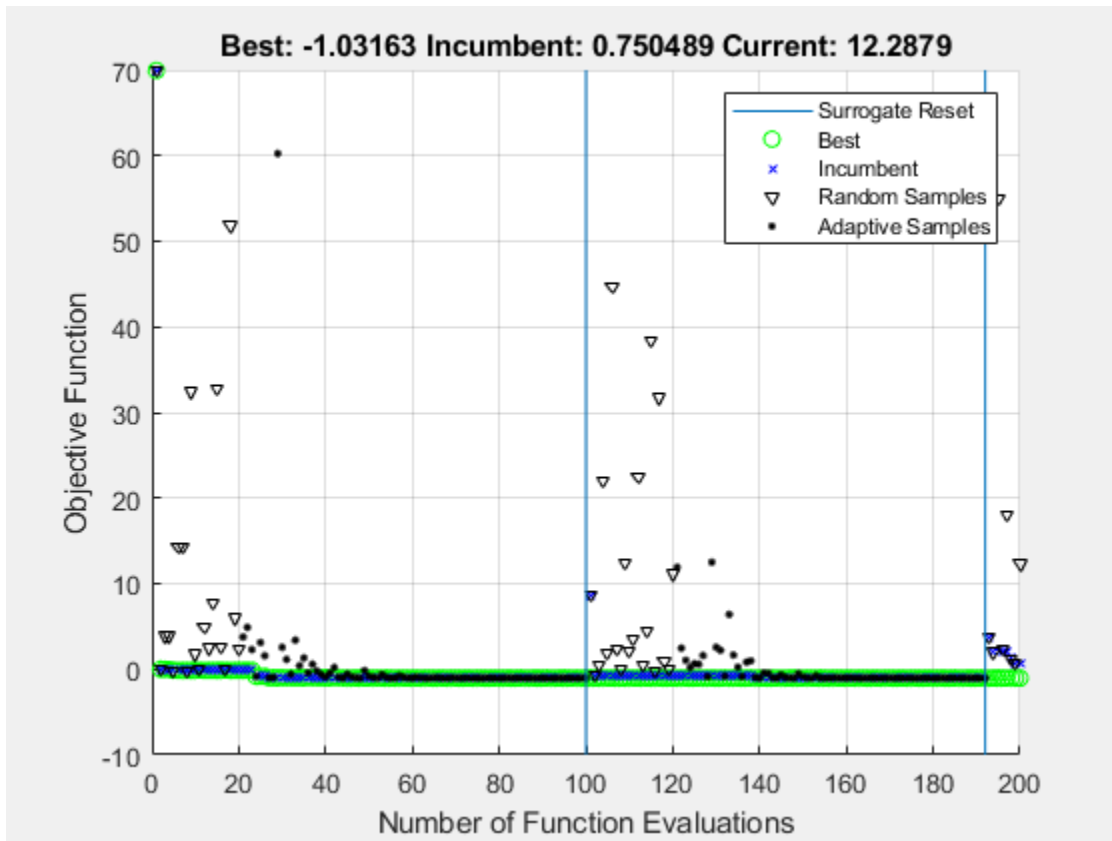
Run `surrogateopt` on the problem using the `'surrogateoptplot'` plot function in the region bounded in each direction by `[-2.1,2.1]`. To understand the `'surrogateoptplot'` plot, see “Interpret surrogateoptplot” on page 11-25.

```
rng default
lb = [-2.1,-2.1];
```

```

ub = -lb;
opts = optimoptions('surrogateopt','PlotFcn','surrogateoptplot');
[xs,fvals,eflags,outputs] = surrogateopt(sixmin,lb,ub,opts);

```



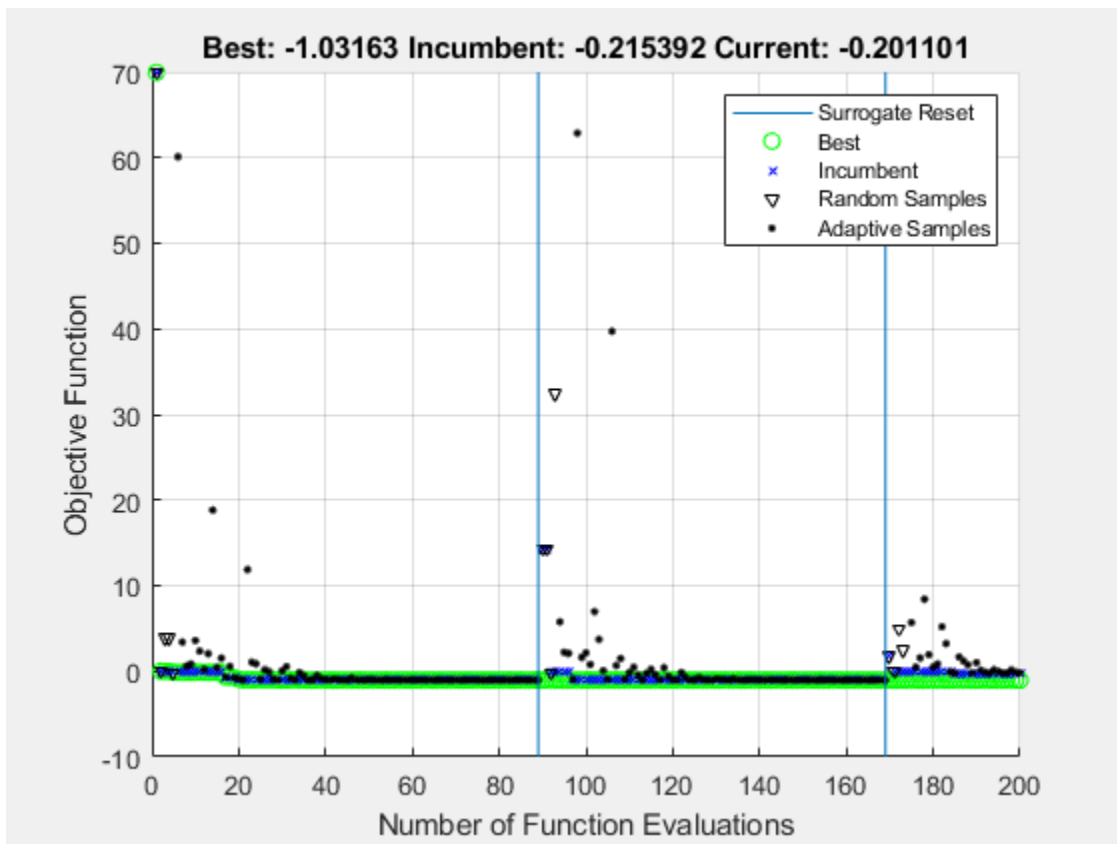
Surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

Set a smaller value for the MinSurrogatePoints option to see whether the change helps the solver reach the global minimum faster.

```

opts.MinSurrogatePoints = 4;
[xs2,fvals2,eflags2,outputs2] = surrogateopt(sixmin,lb,ub,opts);

```

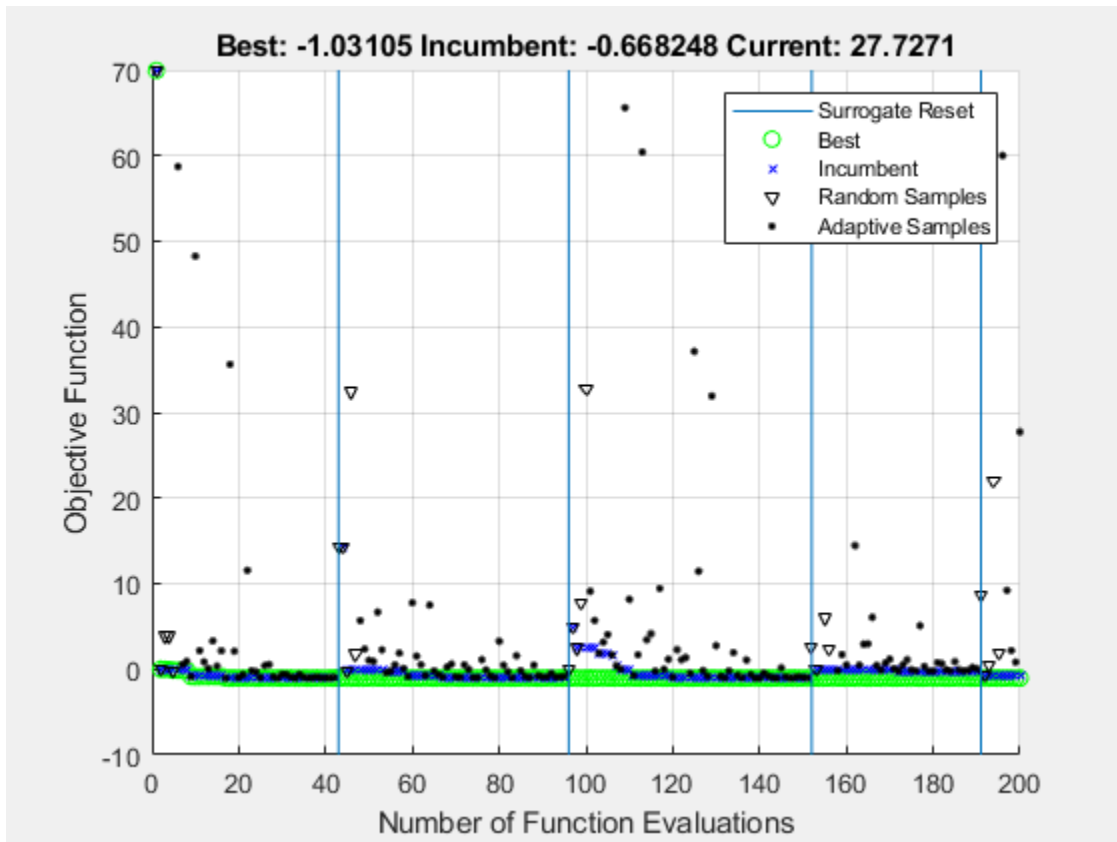


Surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

The smaller `MinSurrogatePoints` option does not noticeably change the solver behavior.

Try setting a larger value of the `MinSampleDistance` option.

```
opts.MinSampleDistance = 0.05;
[xs3,fvals3,eflags3,outputs3] = surrogateopt(sixmin,lb,ub,opts);
```



Surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

Changing the MinSampleDistance option has a small effect on the solver. This setting causes the surrogate to reset more often, and causes the best objective function to be slightly higher (worse) than before.

Try using parallel processing. Time the execution both with and without parallel processing on the camelback function, which is a variant of the sixmin function. To simulate a time-consuming function, the camelback function has an added pause of one second for each function evaluation.

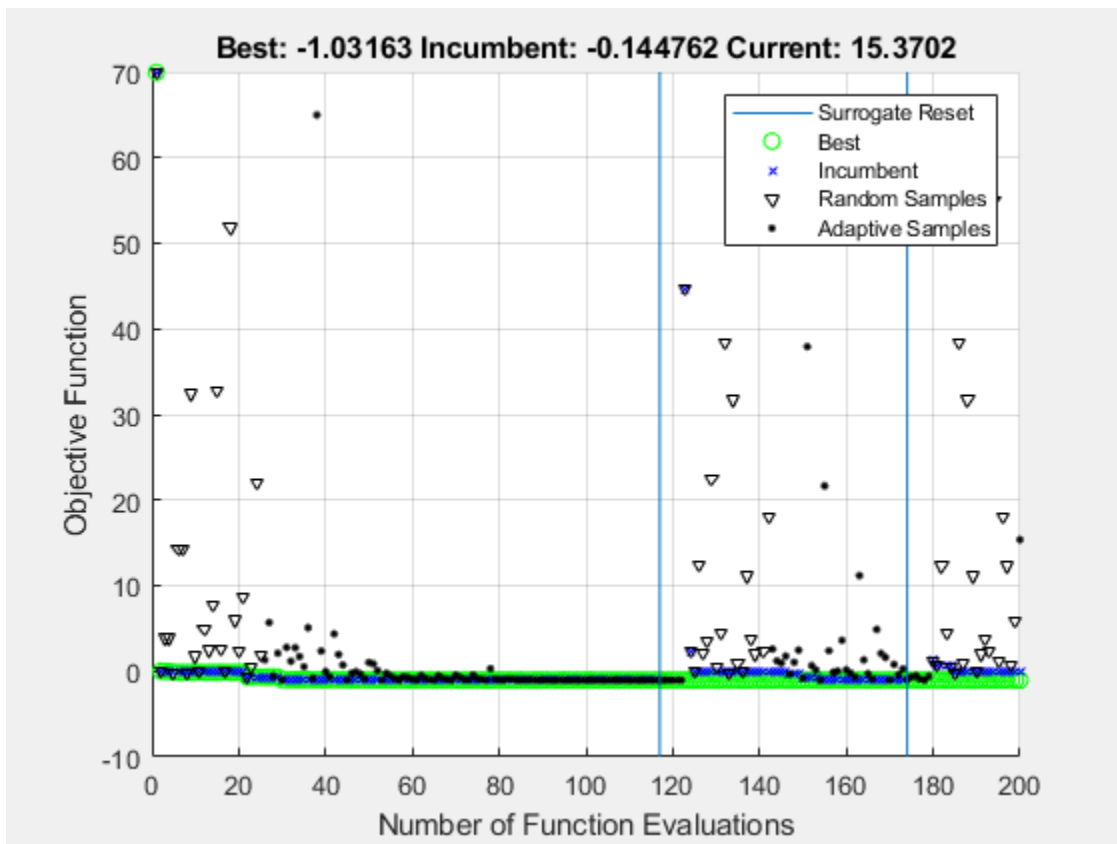
```
type camelback
```

```
function y = camelback(x)
```

```
y = (4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
     + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
pause(1)
```

```
tic
```

```
opts = optimoptions('surrogateopt','UseParallel',true,'PlotFcn','surrogateoptplot');
[xs4,fvals4,eflags4,outputs4] = surrogateopt(@camelback,lb,ub,opts);
```



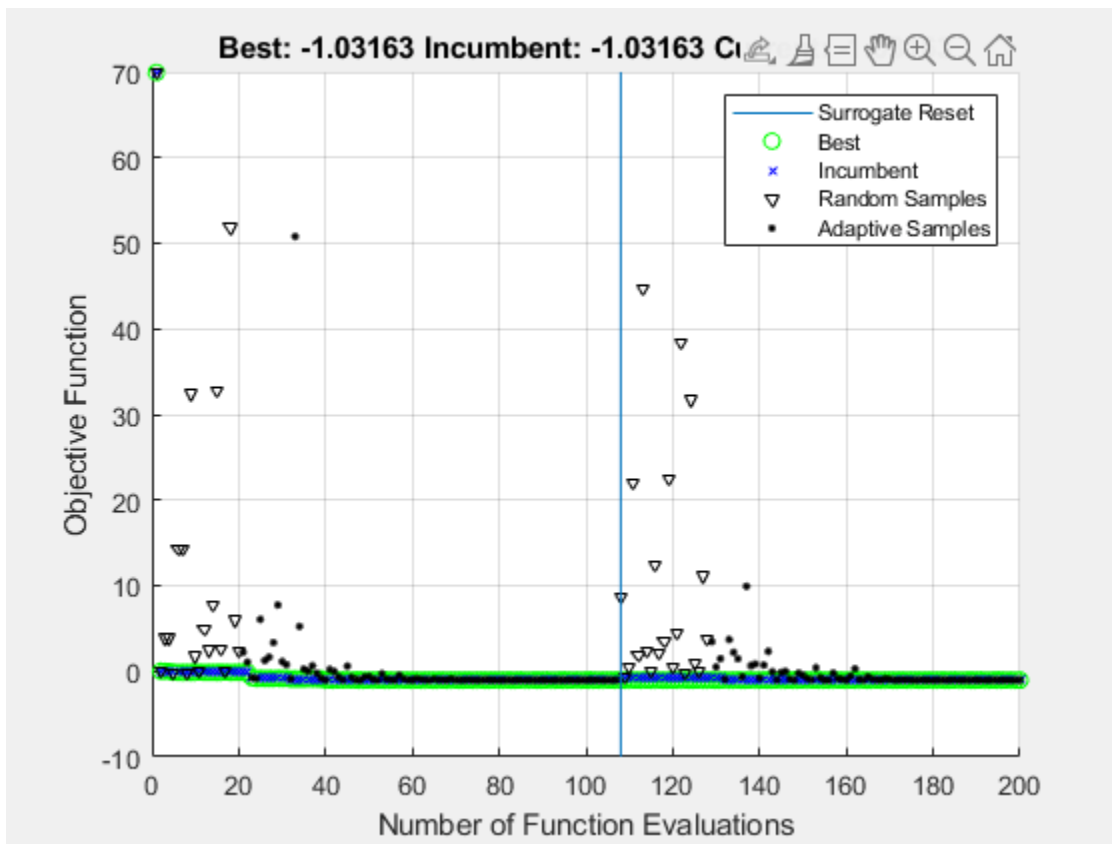
Surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
toc
```

```
Elapsed time is 43.142697 seconds.
```

Time the solver when run on the same problem in serial.

```
opts.UseParallel = false;
tic
[xs5,fvals5,eflags5,outputs5]= surrogateopt(@camelback,lb,ub,opts);
```



Surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
toc
```

```
Elapsed time is 227.968689 seconds.
```

For time-consuming objective functions, parallel processing significantly improves the speed, without overly affecting the results.

See Also

surrogateopt

More About

- “Surrogate Optimization”

Interpret surrogateoptplot

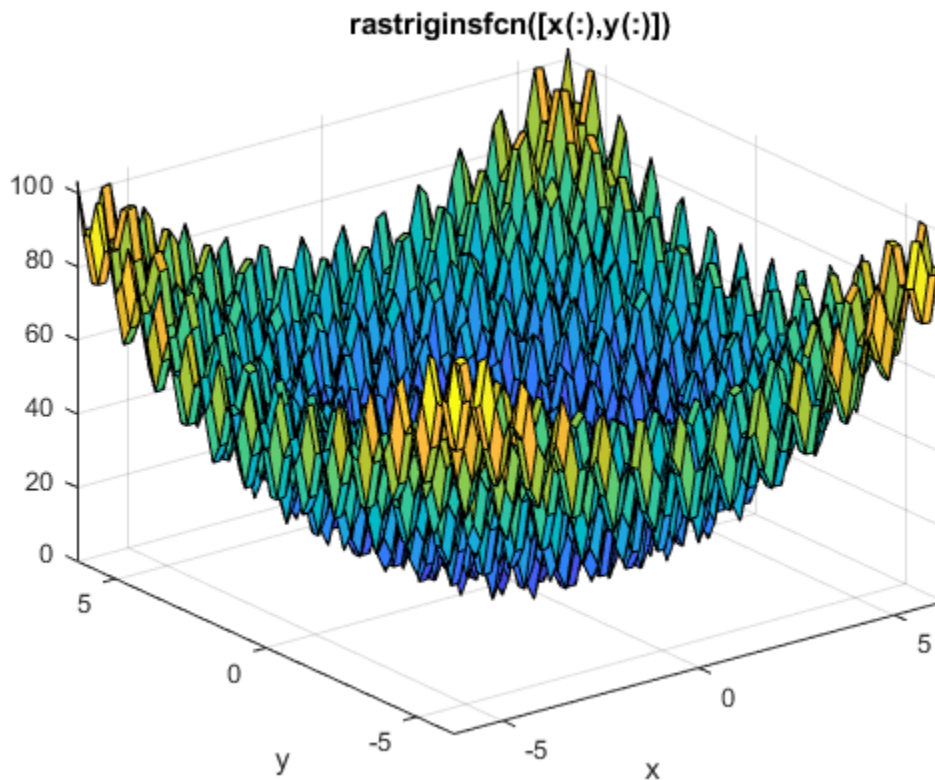
The `surrogateoptplot` plot function provides a good deal of information about the surrogate optimization steps.

Minimize Bounded Function

For example, consider the plot of the steps `surrogateopt` takes on the test function `rastriginsfcn`, which is available when you run this example. This function has a global minimum value of 0 at the point [0,0].

Create a surface plot of `rastriginsfcn`.

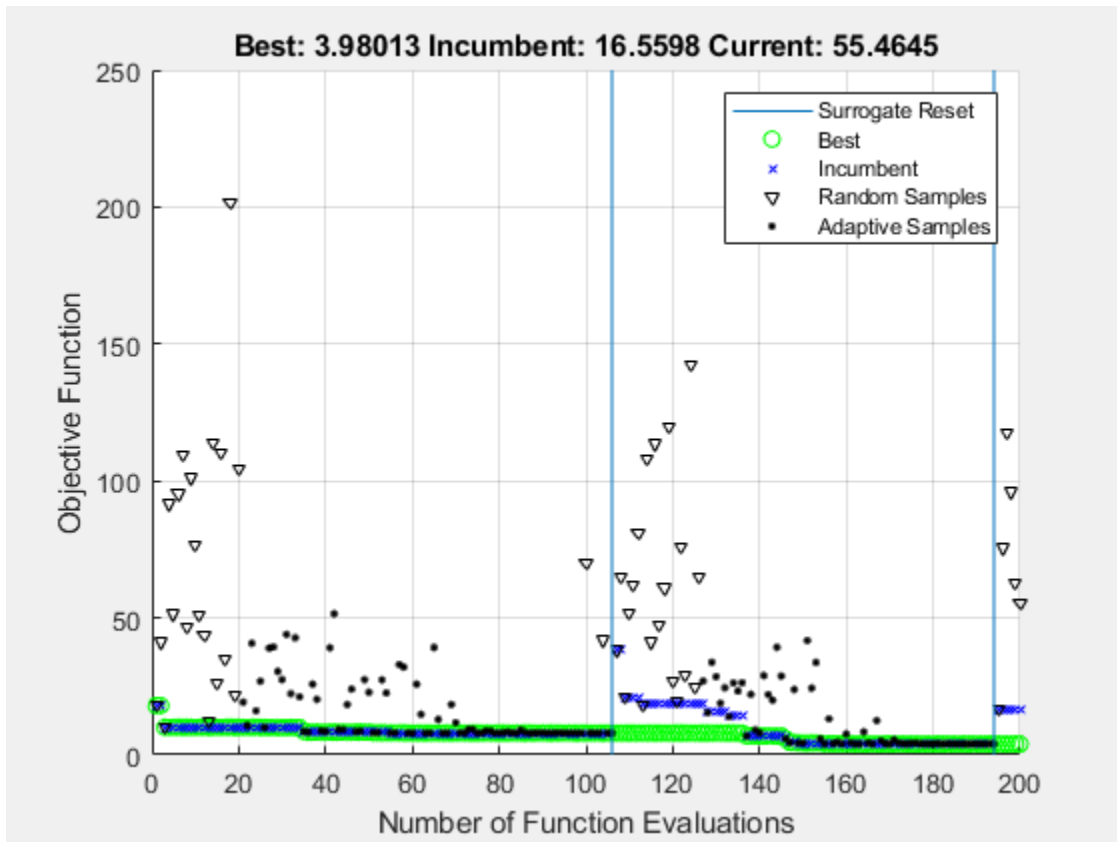
```
ezsurf(@(x,y)rastriginsfcn([x(:),y(:)]));
```



Plot Minimization Process

By giving asymmetric bounds, you encourage `surrogateopt` to search away from the global minimum. Set asymmetric bounds of [-3, -3] and [9, 10]. Set options to use the `surrogateoptplot` plot function, and then call `surrogateopt`.

```
lb = [-3, -3];
ub = [9, 10];
options = optimoptions('surrogateopt', 'PlotFcn', 'surrogateoptplot');
rng(100)
[x, fval] = surrogateopt(@rastriginsfcn, lb, ub, options);
```



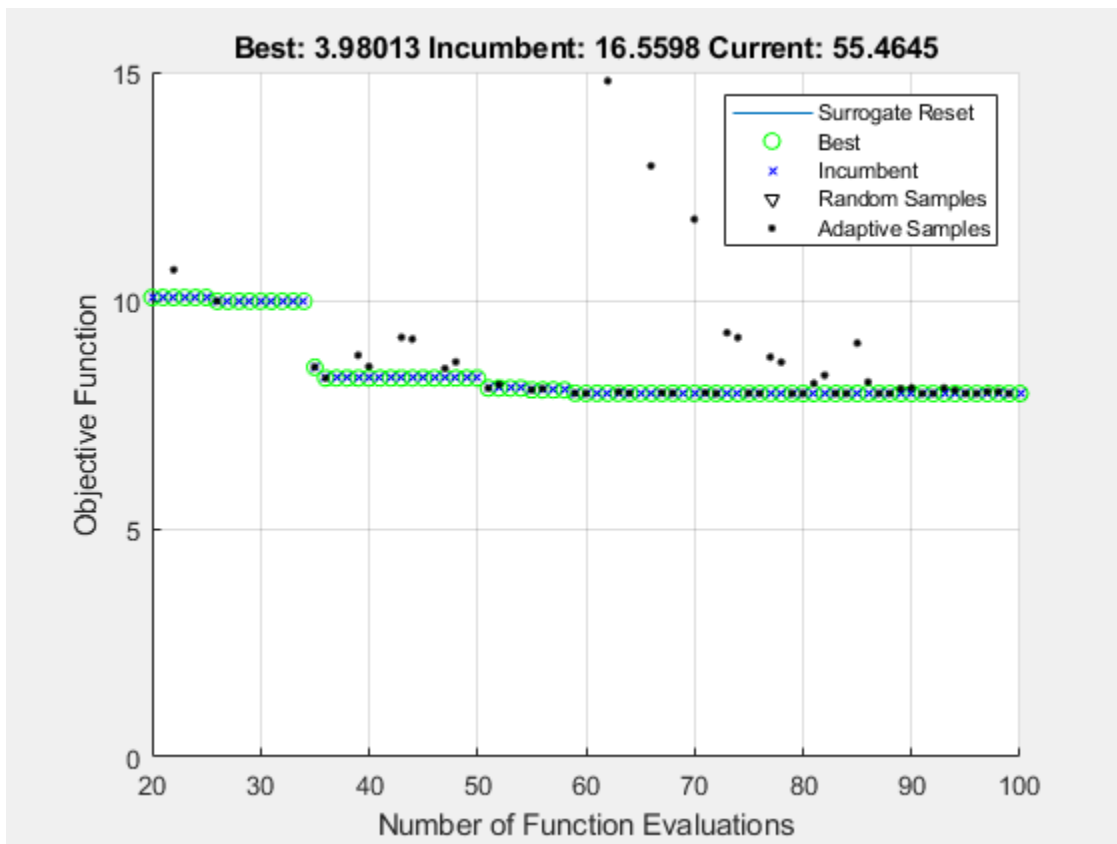
surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

Interpret Plot

Begin interpreting the plot from its left side. For details of the algorithm steps, see “Surrogate Optimization Algorithm” on page 11-3.

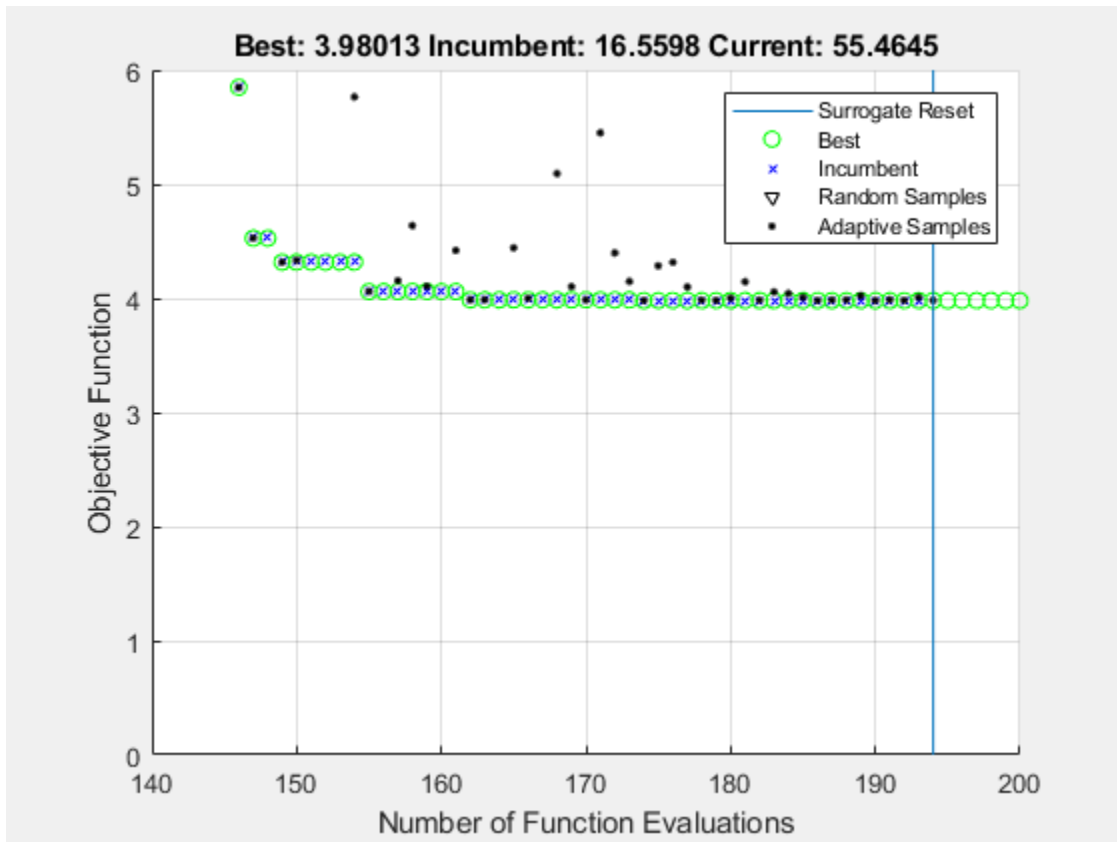
- The first points are black triangles, indicating quasirandom samples of the function within the problem bounds. These points come from the Construct Surrogate phase.
- Next are black dots indicating the adaptive points, the points created in the Search for Minimum phase.
- The thick green line represents the best (lowest) objective function value found. Shortly after evaluation number 30, surrogateopt is stuck in a local minimum with an objective function value near 8. Zoom in to see this behavior more clearly.

```
xlim([20 100])
ylim([0 15])
```



- Near evaluation number 120, a vertical line indicates a surrogate reset. At this point, the algorithm returns to the Construct Surrogate phase.
- The dark blue x points represent the incumbent, which is the best point found since the previous surrogate reset.
- Near evaluation number 150, the incumbent improves on the previous best point by attaining a value of about 4. Zoom in to see this behavior more clearly.

```
xlim([140 200])
ylim([0 6])
```



- The solver has another surrogate reset after evaluation 190.
- The optimization halts at evaluation number 200 because it is the default function evaluation limit for a 2-D problem.

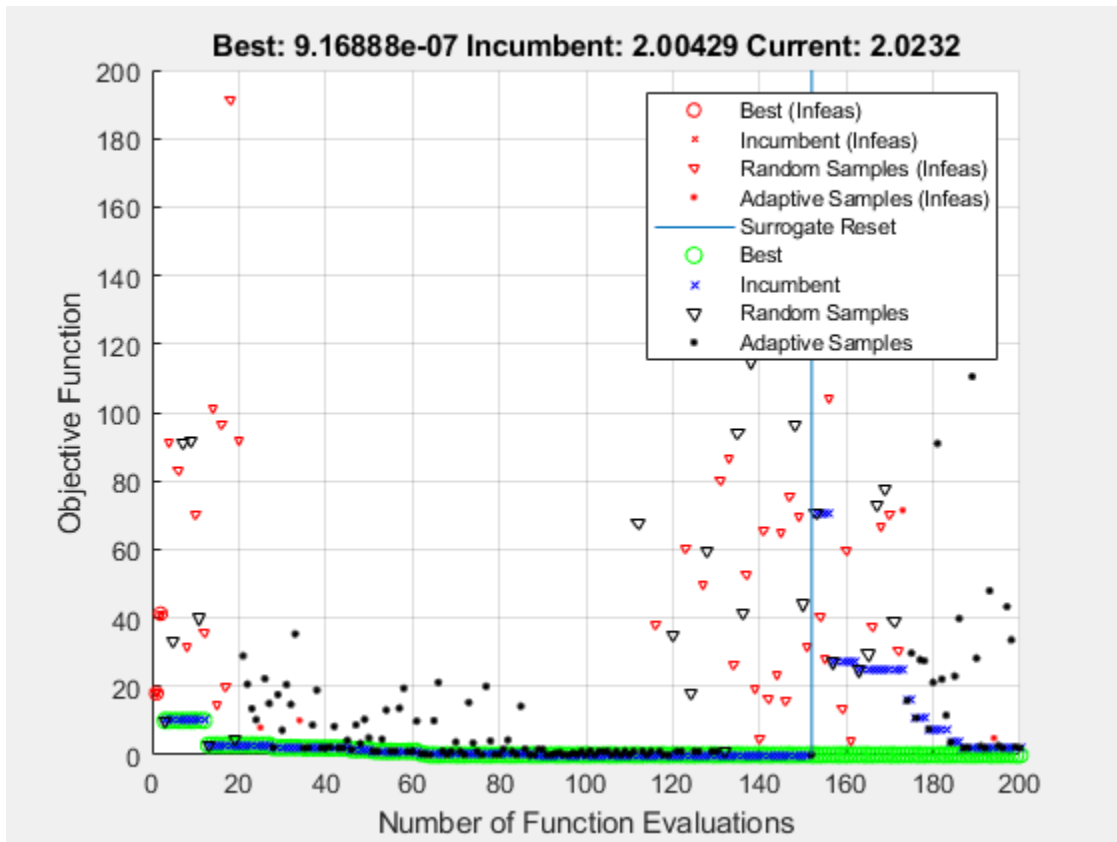
Problem with Nonlinear and Integer Constraints

The `surrogateoptplot` display changes when you have nonlinear constraints. Impose the constraint that $x(1)$ is integer-valued, and the nonlinear constraint that $x_2 \geq x_1^2 - 2$. For the function that implements this constraint, see `rasfcn` on page 11-29 at the end of this example.

```
fun = @rasfcn;
```

Set integer constraints by setting `intcon = 1`, and run the minimization.

```
intcon = 1;
[x,fval] = surrogateopt(fun,lb,ub,intcon,options);
```



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

The plot now shows red markers where surrogateopt evaluates an infeasible point. The final point is close to the true minimum point of [0,0].

```
disp(x)
```

```
1.0e-04 *
    0    0.6798
```

The integer constraint likely helps surrogateopt find the true minimum by reducing the search space.

```
function F = rasfcn(x)
F.Fval = rastriginsfcn(x);
F.Ineq = x(1)^2 - 2 - x(2);
end
```

See Also
surrogateopt

More About

- “Surrogate Optimization Algorithm” on page 11-3
- “Surrogate Optimization”

Compare Surrogate Optimization with Other Solvers

This example compares `surrogateopt` to two other solvers: `fmincon`, the recommended solver for smooth problems, and `patternsearch`, the recommended solver for nonsmooth problems. The example uses a nonsmooth function on a two-dimensional region.

```

type nonSmoothFcn

function [f, g] = nonSmoothFcn(x)
%NONSMOOTHFCN is a non-smooth objective function

% Copyright 2005 The MathWorks, Inc.

for i = 1:size(x,1)
    if x(i,1) < -7
        f(i) = (x(i,1))^2 + (x(i,2))^2 ;
    elseif x(i,1) < -3
        f(i) = -2*sin(x(i,1)) - (x(i,1)*x(i,2)^2)/10 + 15 ;
    elseif x(i,1) < 0
        f(i) = 0.5*x(i,1)^2 + 20 + abs(x(i,2))+ patho(x(i,:));
    elseif x(i,1) >= 0
        f(i) = .3*sqrt(x(i,1)) + 25 +abs(x(i,2)) + patho(x(i,:));
    end
end

%Calculate gradient
g = NaN;
if x(i,1) < -7
    g = 2*[x(i,1); x(i,2)];
elseif x(i,1) < -3
    g = [-2*cos(x(i,1))-(x(i,2)^2)/10; -x(i,1)*x(i,2)/5];
elseif x(i,1) < 0
    [fp,gp] = patho(x(i,:));
    if x(i,2) > 0
        g = [x(i,1)+gp(1); 1+gp(2)];
    elseif x(i,2) < 0
        g = [x(i,1)+gp(1); -1+gp(2)];
    end
elseif x(i,1) >0
    [fp,gp] = patho(x(i,:));
    if x(i,2) > 0
        g = [.15/sqrt(x(i,1))+gp(1); 1+ gp(2)];
    elseif x(i,2) < 0
        g = [.15/sqrt(x(i,1))+gp(1); -1+ gp(2)];
    end
end

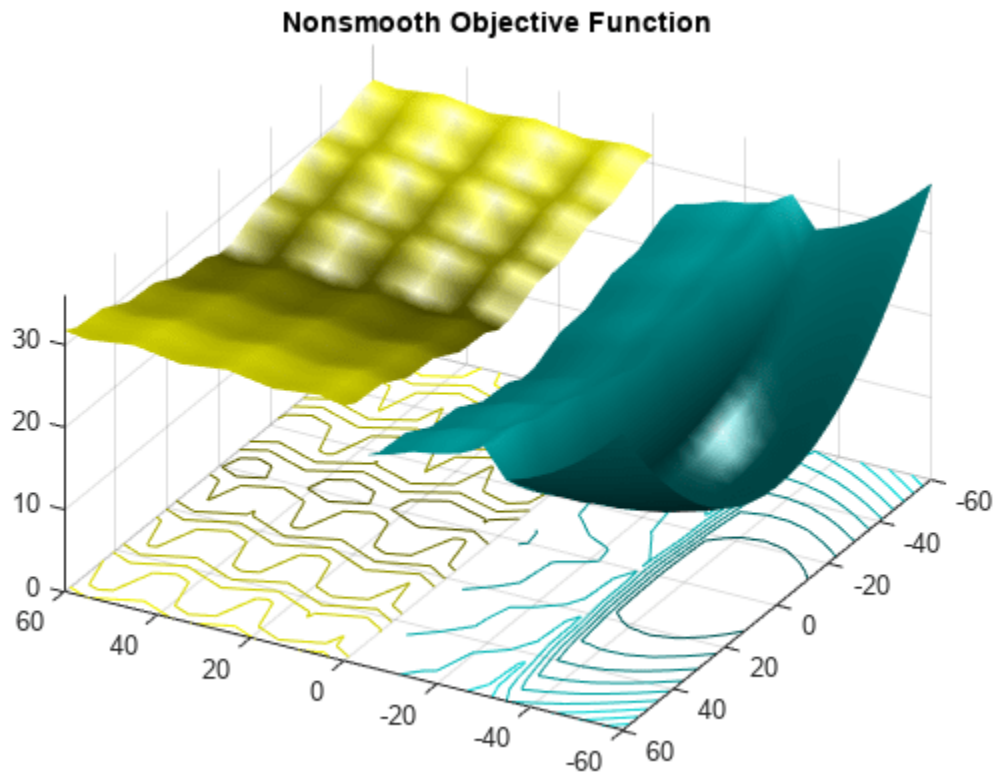
function [f,g] = patho(x)
Max = 500;
f = zeros(size(x,1),1);
g = zeros(size(x));
for k = 1:Max %k
    arg = sin(pi*k^2*x)/(pi*k^2);
    f = f + sum(arg,2);
    g = g + cos(pi*k^2*x);
end

```

```

mplier = 0.1; % Scale the control variable
Objfcn = @(x)nonSmoothFcn(mplier*x); % Handle to the objective function
range = [-6 6;-6 6]/mplier; % Range used to plot the objective function
rng default % Reset the global random number generator
showNonSmoothFcn(Objfcn,range);
title('Nonsmooth Objective Function')
view(-151,44)

```



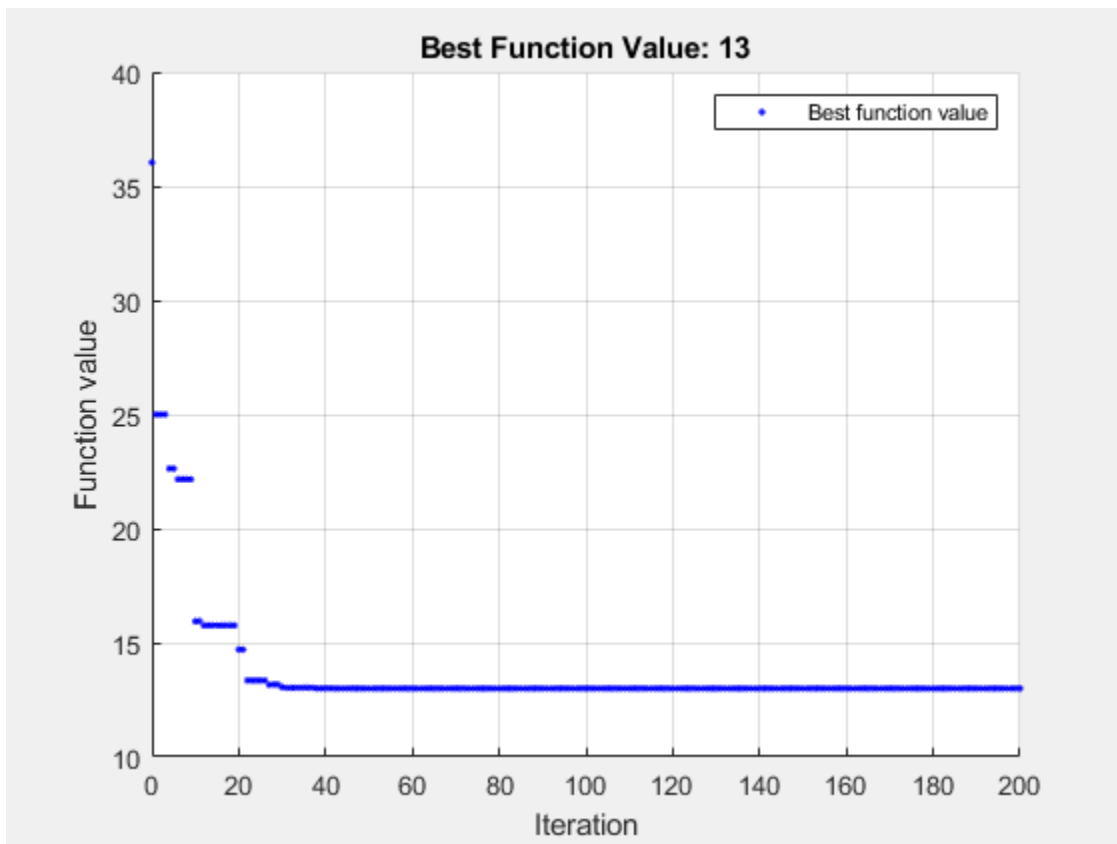
```
drawnow
```

See how well surrogateopt does in locating the global minimum within the default number of iterations.

```

lb = -6*ones(1,2)/mplier;
ub = -lb;
[xs,fvals,eflags,outputs] = surrogateopt(Objfcn,lb,ub);

```

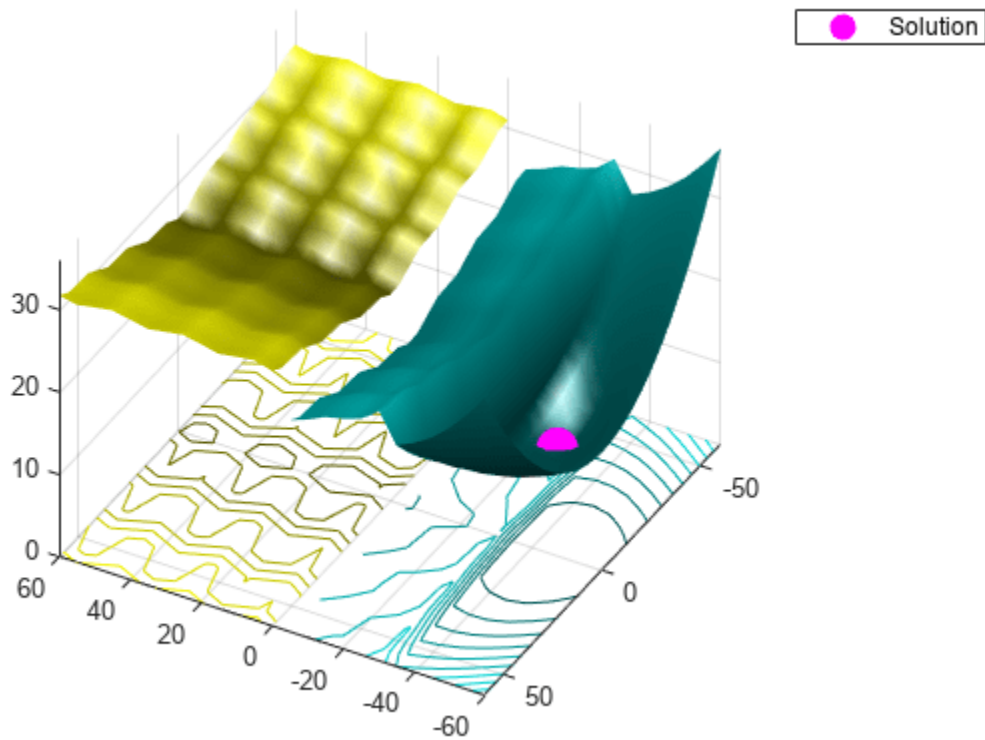



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
fprintf("Lowest found value = %g.\r",fvals)
```

```
Lowest found value = 13.
```

```
figure
showNonSmoothFcn(Objfcn,range);
view(-151,44)
hold on
p1 = plot3(xs(1),xs(2),fvals,'om','MarkerSize',15,'MarkerFaceColor','m');
legend(p1,{'Solution'})
hold off
```

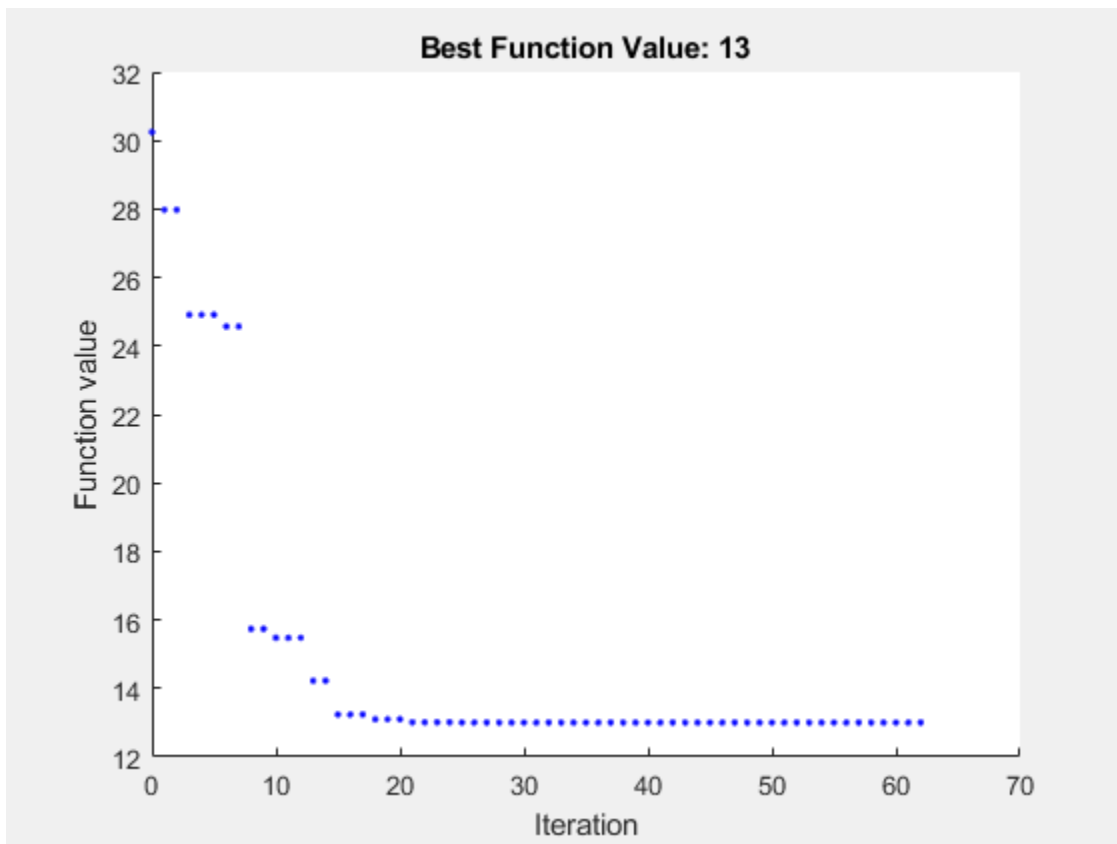


Compare with patternsearch

Set `patternsearch` options to use the same number of function evaluations, starting from a random point within the bounds.

```
rng default
x0 = lb + rand(size(lb)).*(ub - lb);
optsps = optimoptions('patternsearch','MaxFunctionEvaluations',200,'PlotFcn','psplotbestf');
[xps,fvalps,eflagps,outputps] = patternsearch(Objfcn,x0,[],[],[],[],lb,ub,[],optsps);
```

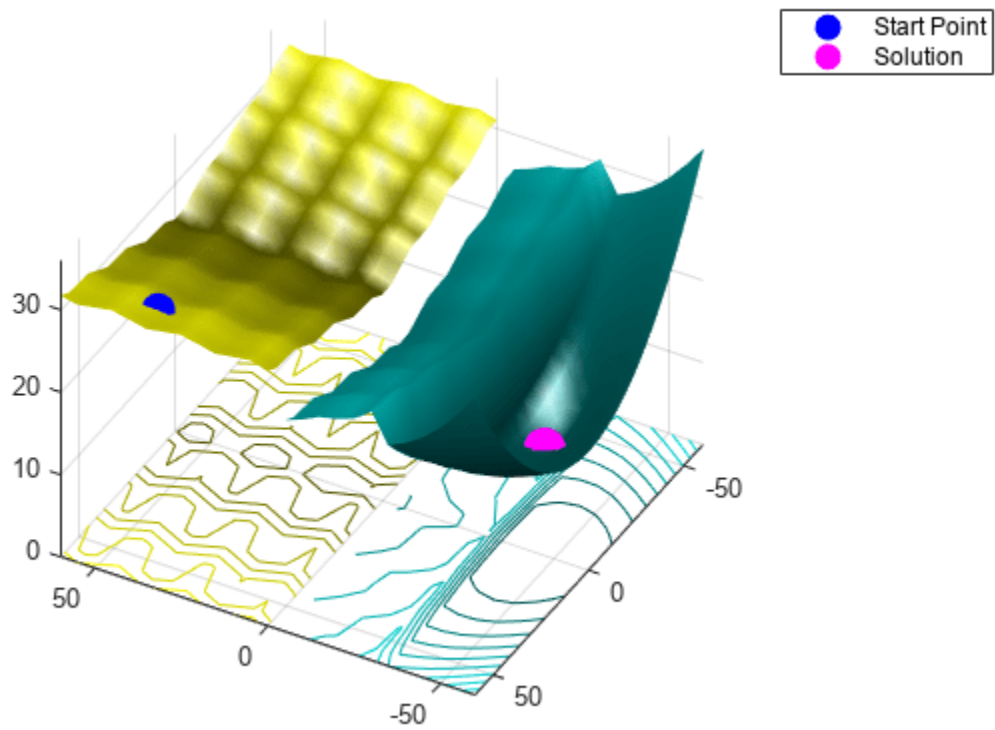
Optimization terminated: mesh size less than options.MeshTolerance.



```

figure
showNonSmoothFcn(Objfcn, range);
view(-151,44)
hold on
p1 = plot3(x0(1),x0(2),Objfcn(x0),'ob','MarkerSize',12,'MarkerFaceColor','b');
p2 = plot3(xps(1),xps(2),fvalps,'om','MarkerSize',15,'MarkerFaceColor','m');
legend([p1,p2],{'Start Point','Solution'})
hold off

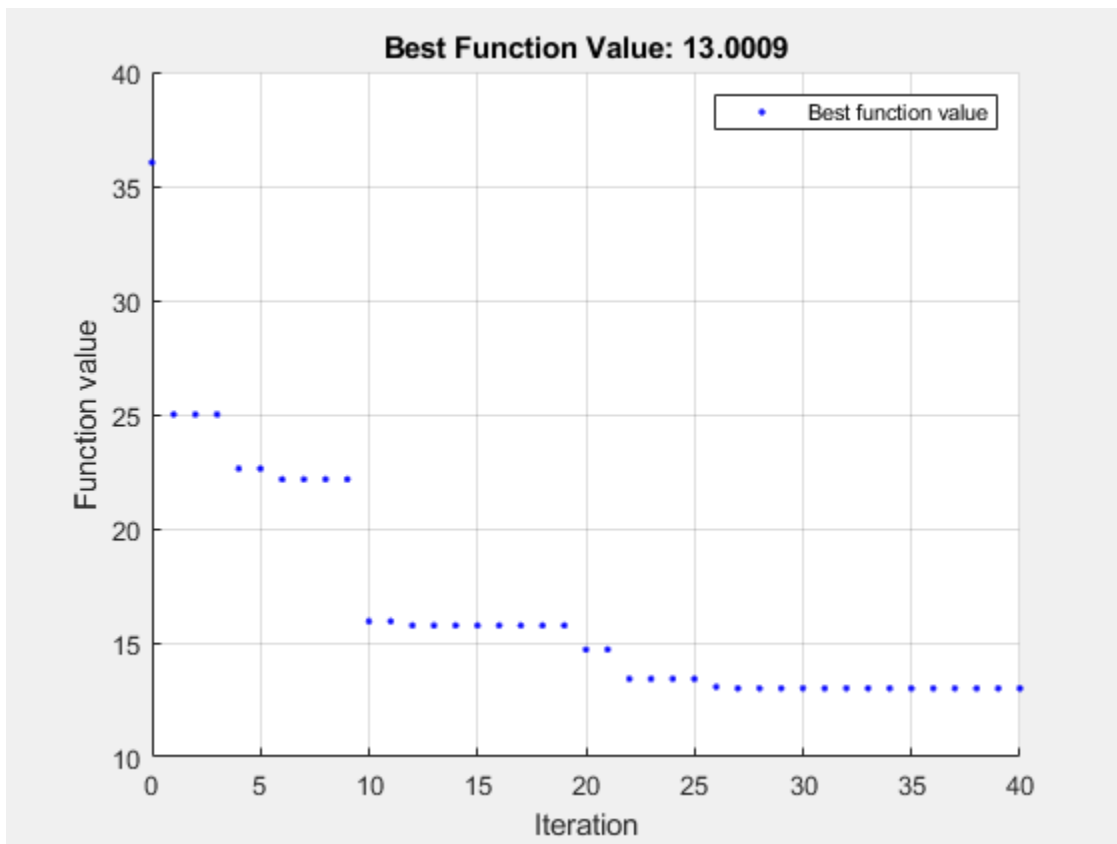
```



patternsearch found the same solution as surrogateopt.

Restrict the number of function evaluations and try again.

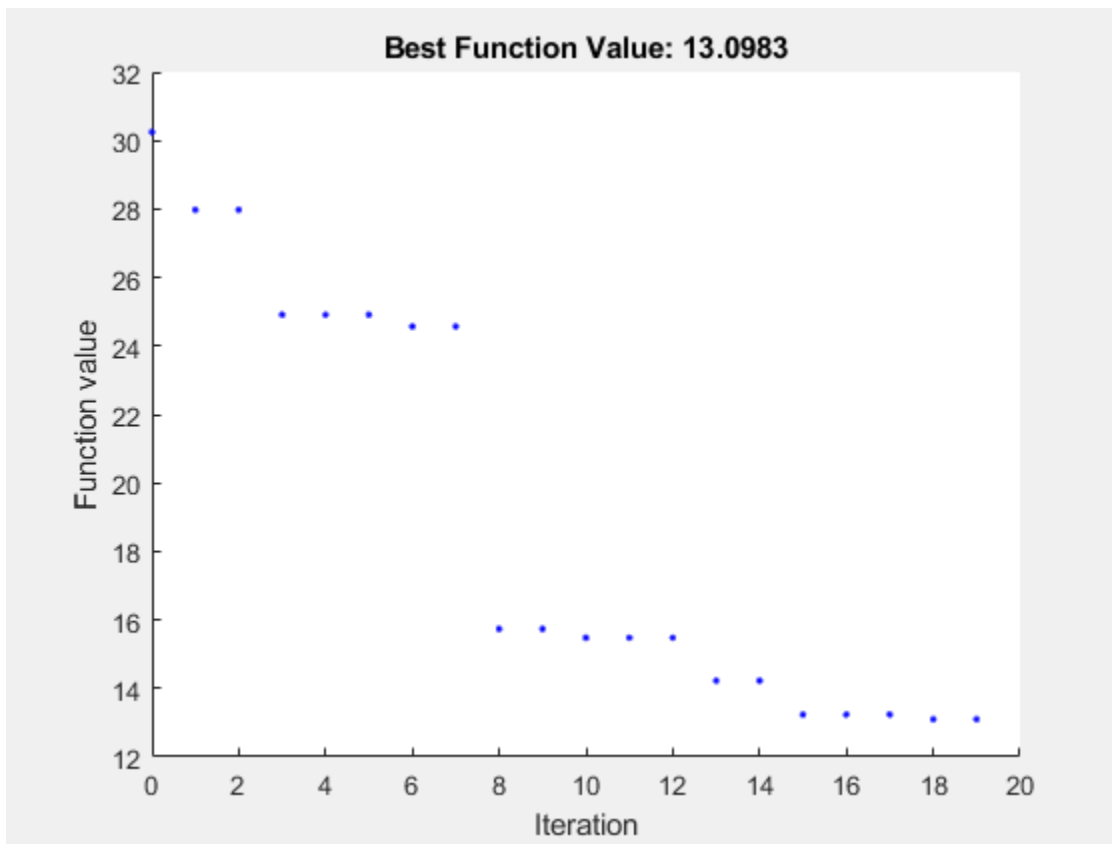
```
optsurr = optimoptions('surrogateopt','MaxFunctionEvaluations',40);  
[xs,fvals,eflags,outputs] = surrogateopt(Objfcn,lb,ub,optsurr);
```



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
optsps.MaxFunctionEvaluations = 40;  
[xps,fvalps,eflags,outputps] = patternsearch(Objfcn,x0,[],[],[],[],lb,ub,[],optsps);
```

Maximum number of function evaluations exceeded: increase options.MaxFunctionEvaluations.



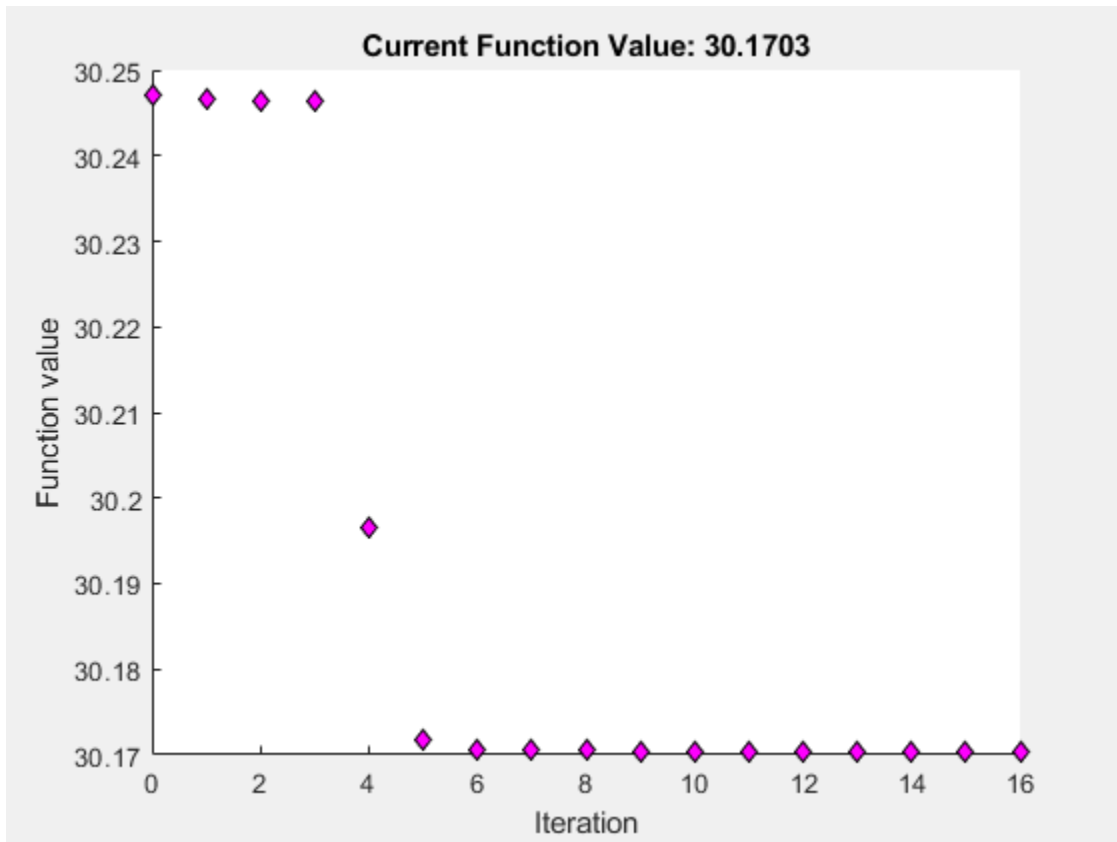
Again, both solvers found the global solution quickly.

Compare with `fmincon`

`fmincon` is efficient at finding a local solution near the start point. However, it can easily get stuck far from the global solution in a nonconvex or nonsmooth problem.

Set `fmincon` options to use a plot function, the same number of function evaluations as the previous solvers, and the same start point as `patternsearch`.

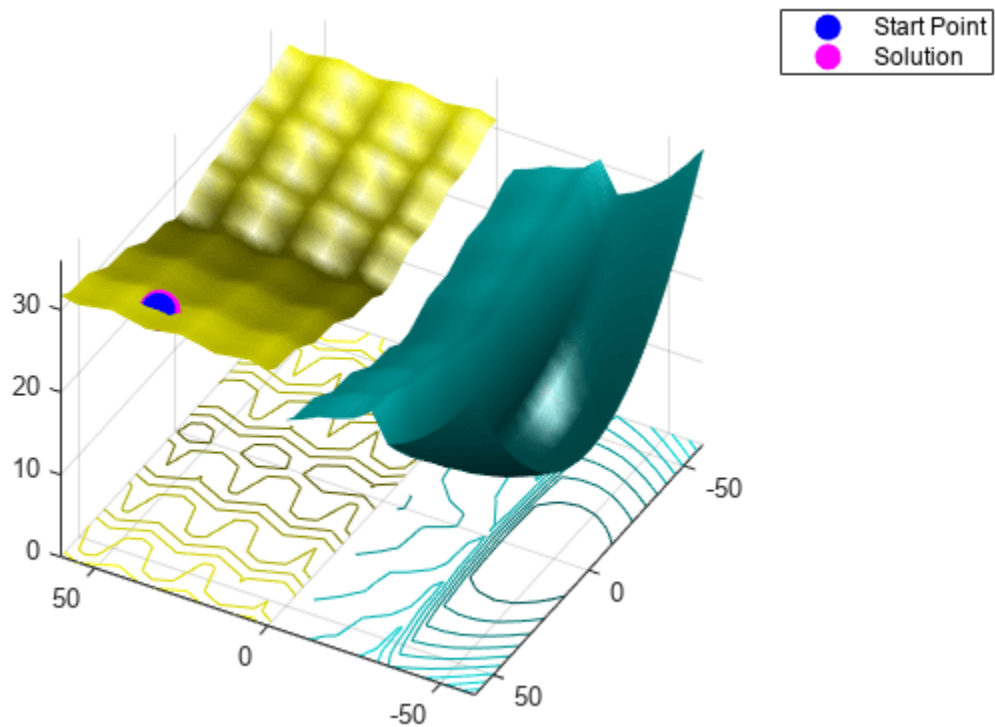
```
opts = optimoptions('fmincon','PlotFcn','optimplotfval','MaxFunctionEvaluations',200);  
[fmsol, fmfval, eflag, fmoutput] = fmincon(Objfcn, x0, [], [], [], [], lb, ub, [], opts);
```



Local minimum possible. Constraints satisfied.

fmincon stopped because the size of the current step is less than the value of the step size tolerance and constraints are satisfied to within the value of the constraint tolerance.

```
figure
showNonSmoothFcn(Objfcn,range);
view(-151,44)
hold on
p1 = plot3(x0(1),x0(2),Objfcn(x0),'ob','MarkerSize',12,'MarkerFaceColor','b');
p2 = plot3(fmsol(1),fmsol(2),fmfval,'om','MarkerSize',15,'MarkerFaceColor','m');
legend([p1,p2],{'Start Point','Solution'})
hold off
```



`fmincon` is stuck in a local minimum near the start point.

See Also

`fmincon` | `patternsearch` | `surrogateopt`

More About

- “Modify `surrogateopt` Options” on page 11-19
- “Surrogate Optimization of Multidimensional Function” on page 11-12

Surrogate Optimization with Nonlinear Constraint

This example shows how to include nonlinear inequality constraints in a surrogate optimization. The example solves an ODE with a nonlinear constraint. The example “Optimize ODEs in Parallel” on page 6-87 shows how to solve the same problem using other solvers that accept nonlinear constraints.

For a video overview of this example, see Surrogate Optimization.

Problem Description

The problem is to change the position and angle of a cannon to fire a projectile as far as possible beyond a wall. The cannon has a muzzle velocity of 300 m/s. The wall is 20 m high. If the cannon is too close to the wall, it fires at too steep an angle, and the projectile does not travel far enough. If the cannon is too far from the wall, the projectile does not travel far enough.

Nonlinear air resistance slows the projectile. The resisting force is proportional to the square of velocity, with the proportionality constant 0.02. Gravity acts on the projectile, accelerating it downward with constant 9.81 m/s². Therefore, the equations of motion for the trajectory $x(t)$ are

$$\frac{d^2x(t)}{dt^2} = -0.02\|v(t)\|v(t) - (0, 9.81),$$

where $v(t) = dx(t)/dt$.

The initial position x_0 and initial velocity x_{p0} are 2-D vectors. However, the initial height $x_0(2)$ is 0, so the initial position is given by the scalar $x_0(1)$. The initial velocity has magnitude 300 (the muzzle velocity) and, therefore, depends only on the initial angle, which is a scalar. For an initial angle θ , the initial velocity is $x_{p0} = 300 * (\cos(\theta), \sin(\theta))$. Therefore, the optimization problem depends only on two scalars, making it a 2-D problem. Use the horizontal distance and initial angle as the decision variables.

Formulate ODE Model

ODE solvers require you to formulate your model as a first-order system. Augment the trajectory vector $(x_1(t), x_2(t))$ with its time derivative $(x_1'(t), x_2'(t))$ to form a 4-D trajectory vector. In terms of this augmented vector, the differential equation becomes

$$\frac{d}{dt}x(t) = \begin{bmatrix} x_3(t) \\ x_4(t) \\ -0.02\|(x_3(t), x_4(t))\|x_3(t) \\ -0.02\|(x_3(t), x_4(t))\|x_4(t) - 9.81 \end{bmatrix}.$$

The `cannonshot` file implements this differential equation.

type `cannonshot`

```
function f = cannonshot(~,x)
```

```
f = [x(3);x(4);x(3);x(4)]; % initial, gets f(1) and f(2) correct
nrm = norm(x(3:4)) * .02; % norm of the velocity times constant
f(3) = -x(3)*nrm; % horizontal acceleration
f(4) = -x(4)*nrm - 9.81; % vertical acceleration
```

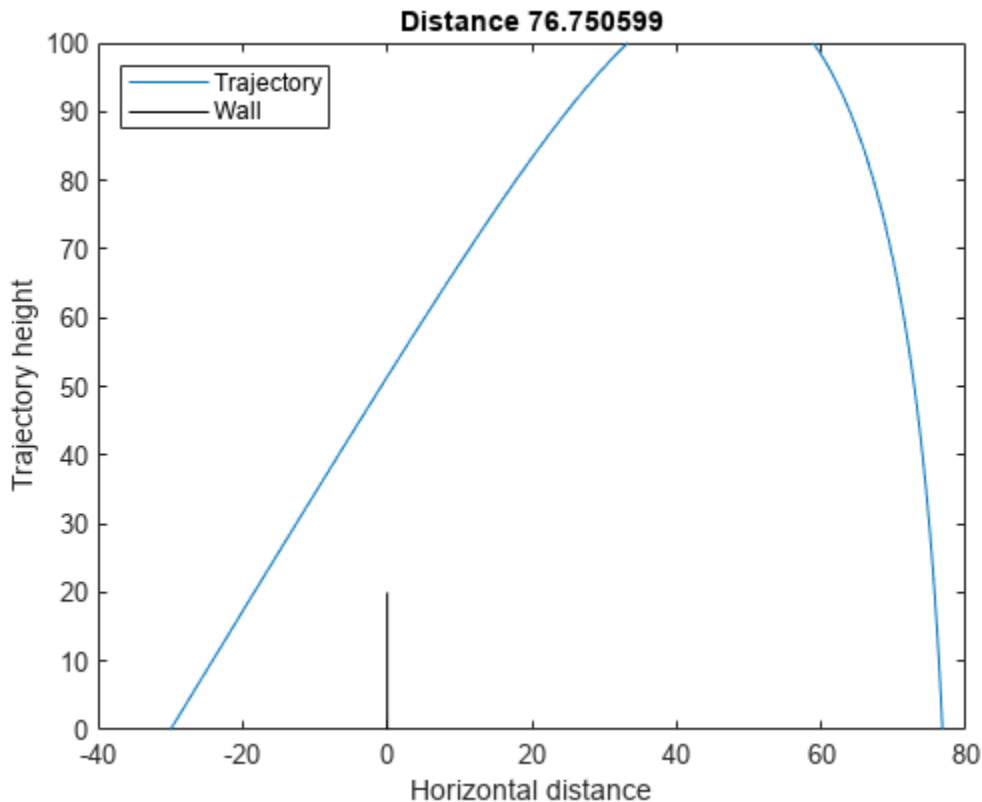
Visualize the solution of this ODE starting 30 m from the wall with an initial angle of $\pi/3$. The `plotcannonsolution` function uses `ode45` to solve the differential equation.

type `plotcannonsolution`

```
function dist = plotcannonsolution(x)
% Change initial 2-D point x to 4-D x0
x0 = [x(1);0;300*cos(x(2));300*sin(x(2))];
sol = ode45(@cannonshot,[0,15],x0);
% Find the time when the projectile lands
zerofnd = fzero(@(r)deval(sol,r,2),[sol.x(2),sol.x(end)]);
t = linspace(0,zerofnd); % equal times for plot
xs = deval(sol,t,1); % interpolated x values
ys = deval(sol,t,2); % interpolated y values
plot(xs,ys)
hold on
plot([0,0],[0,20],'k') % Draw the wall
xlabel('Horizontal distance')
ylabel('Trajectory height')
ylim([0 100])
legend('Trajectory','Wall','Location','NW')
dist = xs(end);
title(sprintf('Distance %f',dist))
hold off
```

`plotcannonsolution` uses `fzero` to find the time when the projectile lands, meaning its height is 0. The projectile lands before time 15 s, so `plotcannonsolution` uses 15 as the amount of time for the ODE solution.

```
x0 = [-30;pi/3];
dist = plotcannonsolution(x0);
```



Prepare Optimization

To optimize the initial position and angle, write a function similar to the previous plotting routine. Calculate the trajectory starting from an arbitrary horizontal position and initial angle.

Include sensible bound constraints. The horizontal position cannot be greater than 0. Set an upper bound of -1. Similarly, the horizontal position cannot be below -200, so set a lower bound of -200. The initial angle must be positive, so set its lower bound to 0.05. The initial angle should not exceed $\pi/2$; set its upper bound to $\pi/2 - 0.05$.

```
lb = [-200;0.05];
ub = [-1;pi/2-.05];
```

Write an objective function that returns the negative of the resulting distance from the wall, given an initial position and angle. If the trajectory crosses the wall at a height less than 20, the trajectory is infeasible; this constraint is a nonlinear constraint. The `cannonobjcon` function implements the objective function calculation. To implement the nonlinear constraint, the function calls `fzero` to find the time when the x-value of the projectile is zero. The function accounts for the possibility of failure in the `fzero` function by checking whether, after time 15, the x-value of the projectile is greater than zero. If not, then the function skips the step of finding the time when the projectile passes the wall.

```
type cannonobjcon

function f = cannonobjcon(x)
    % Change initial 2-D point x to 4-D x0
    x0 = [x(1);0;300*cos(x(2));300*sin(x(2))];
    % Solve for trajectory
```

```
sol = ode45(@cannonshot,[0,15],x0);
% Find time t when trajectory height = 0
zerofnd = fzero(@(r)deval(sol,r,2),[1e-2,15]);
% Find the horizontal position at that time
dist = deval(sol,zerofnd,1);
% What is the height when the projectile crosses the wall at x = 0?
if deval(sol,15,1) > 0
    wallfnd = fzero(@(r)deval(sol,r,1),[0,15]);
    height = deval(sol,wallfnd,2);
else
    height = deval(sol,15,2);
end
f.Ineq = 20 - height; % height must be above 20
% Take negative of distance for maximization
f.Fval = -dist;
end
```

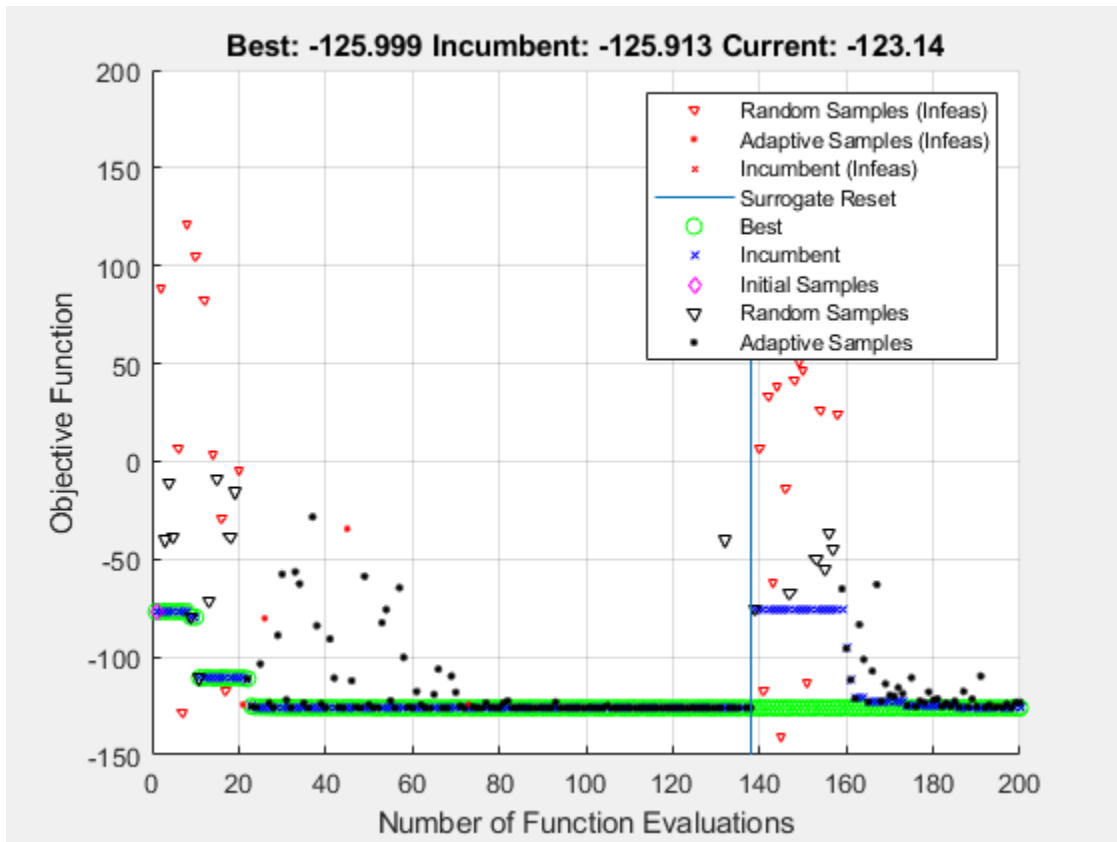
You already calculated one feasible initial trajectory. Use that value as an initial point.

```
fx0 = cannonobjcon(x0);
fx0.X = x0;
```

Solve Optimization Using surrogateopt

Set `surrogateopt` options to use the initial point. For reproducibility, set the random number generator to `default`. Use the `'surrogateoptplot'` plot function. Run the optimization. To understand the `'surrogateoptplot'` plot, see “Interpret surrogateoptplot” on page 11-25.

```
opts = optimoptions('surrogateopt','InitialPoints',x0,'PlotFcn','surrogateoptplot');
rng default
[xsolution,distance,exitflag,output] = surrogateopt(@cannonobjcon,lb,ub,opts)
```



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
xsolution = 1x2
```

```
-28.4012    0.6161
```

```
distance = -125.9987
```

```
exitflag = 0
```

```
output = struct with fields:
```

```
    elapsedtime: 64.6181
```

```
    funccount: 200
```

```
    constrviolation: 8.0630e-04
```

```
    ineq: 8.0630e-04
```

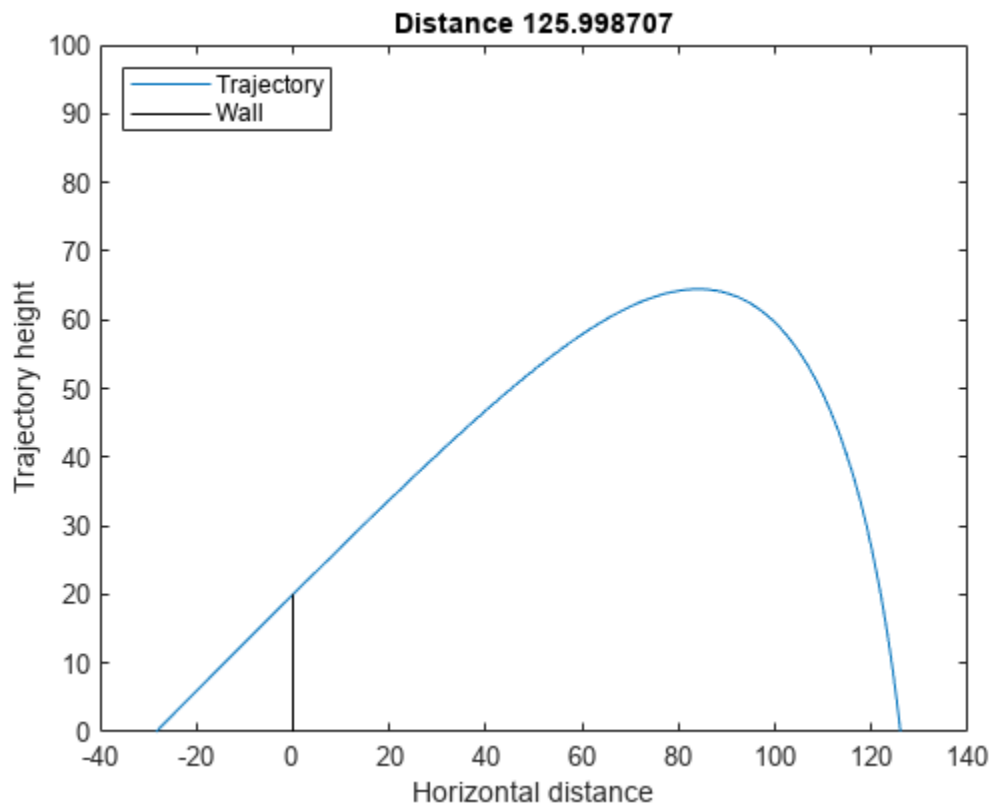
```
    rngstate: [1x1 struct]
```

```
    message: 'surrogateopt stopped because it exceeded the function evaluation limit set
```

Plot the final trajectory.

```
figure
```

```
dist = plotcannonsolution(xsolution);
```



The patternsearch solution in “Optimize ODEs in Parallel” on page 6-87 shows a final distance of 125.9880, which is almost the same as this surrogateopt solution.

See Also

surrogateopt

More About

- “Optimize ODEs in Parallel” on page 6-87
- Surrogate Optimization

Surrogate Optimization of Six-Element Yagi-Uda Antenna

This example shows how to optimize an antenna design using the surrogate optimization solver. The radiation patterns of antennas depend sensitively on the parameters that define the antenna shapes. Typically, the features of a radiation pattern have multiple local optima. To calculate a radiation pattern, this example uses Antenna Toolbox™ functions.

A Yagi-Uda antenna is a widely used radiating structure for a variety of applications in commercial and military sectors. This antenna can receive TV signals in the VHF-UHF range of frequencies [1]. The Yagi-Uda is a directional traveling-wave antenna with a single driven element, usually a folded dipole or a standard dipole, which is surrounded by several passive dipoles. The passive elements form the *reflector* and *director*. These names identify the positions relative to the driven element. The reflector dipole is behind the driven element, in the direction of the back lobe of the antenna radiation. The director dipole is in front of the driven element, in the direction where a main beam forms.

Design Parameters

Specify the initial design parameters in the center of the VHF band [2].

```
freq = 165e6;
wirediameter = 19e-3;
c = physconst('lightspeed');
lambda = c/freq;
```

Create Yagi-Uda Antenna

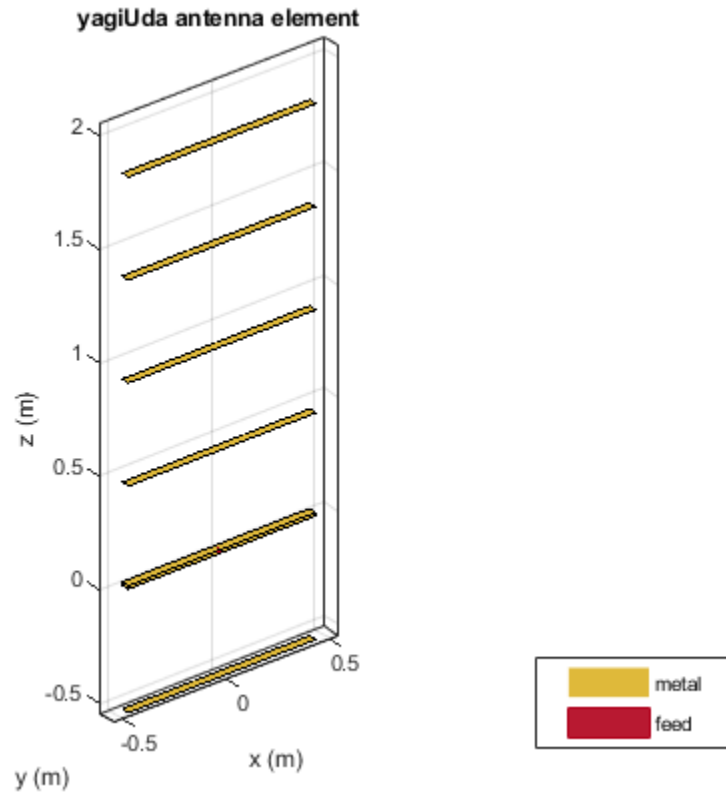
The driven element for the Yagi-Uda antenna is a folded dipole, a standard exciter for this type of antenna. Adjust the length and width parameters of the folded dipole. Because cylindrical structures are modeled as equivalent metal strips, calculate the width using the `cylinder2strip` utility function available in the Antenna Toolbox™. The length is $\lambda/2$ at the design frequency.

```
d = dipoleFolded;
d.Length = lambda/2;
d.Width = cylinder2strip(wirediameter/2);
d.Spacing = d.Length/60;
```

Create a Yagi-Uda antenna with the exciter as the folded dipole. Set the lengths of the reflector and director elements to be $\lambda/2$. Set the number of directors to four. Specify the reflector and director spacing as 0.3λ and 0.25λ , respectively. These settings provide an initial guess and serve as a starting point for the optimization procedure. Show the initial design.

```
Numdirs = 4;
refLength = 0.5;
dirLength = 0.5*ones(1,Numdirs);
refSpacing = 0.3;
dirSpacing = 0.25*ones(1,Numdirs);
initialdesign = [refLength dirLength refSpacing dirSpacing].*lambda;
yagidesign = yagiUda;
yagidesign.Exciter = d;
yagidesign.NumDirectors = Numdirs;
yagidesign.ReflectorLength = refLength*lambda;
yagidesign.DirectorLength = dirLength.*lambda;
yagidesign.ReflectorSpacing = refSpacing*lambda;
```

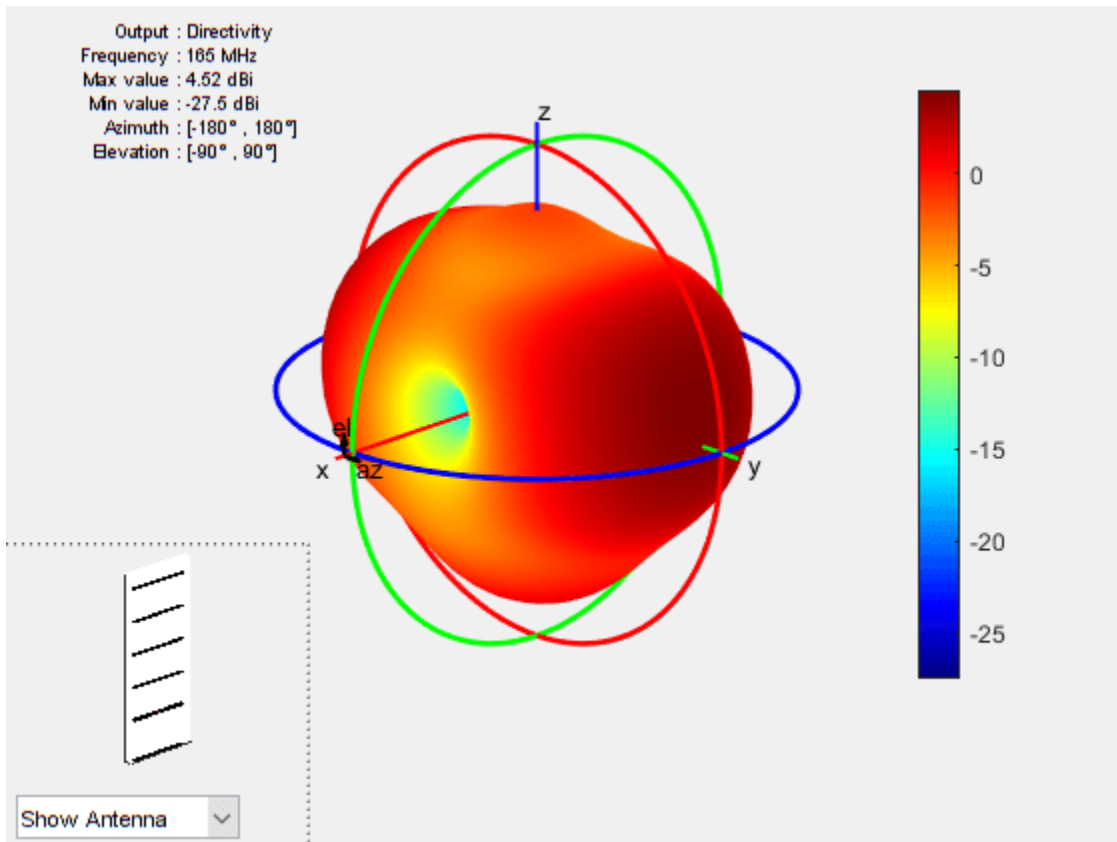
```
yagidesign.DirectorSpacing = dirSpacing*lambda;  
show(yagidesign)
```



Plot Radiation Pattern at Design Frequency

Prior to executing the optimization process, plot the radiation pattern for the initial guess in 3-D.

```
fig1 = figure;  
pattern(yagidesign, freq);
```

This antenna does not have a higher directivity in the preferred direction, at zenith (elevation = 90 deg). This initial Yagi-Uda antenna design is a poorly designed radiator.

Set Up Optimization

Use the following variables as control variables for the optimization:

- Reflector length (1 variable)
- Director lengths (4 variables)
- Reflector spacing (1 variable)
- Director spacings (4 variables)

In terms of a single vector parameter `parasiticVals`, use these settings:

- Reflector length = `parasiticVals(1)`
- Director lengths = `parasiticVals(2:5)`
- Reflector spacing = `parasiticVals(6)`
- Director spacings = `parasiticVals(7:10)`

In terms of `parasiticVals`, set an objective function that aims to have a large value in the 90-degree direction, a small value in the 270-degree direction, and a large value of maximum power between the elevation beamwidth angle bounds.

type `yagi_objective_function2.m`

```

function objectivevalue = yagi_objective_function2(y,parasiticVals,freq,elang)
% yagi_objective_function2 returns the objective for a 6-element Yagi
% objective_value = yagi_objective_function(y,parasiticvals,freq,elang)
% assigns the appropriate parasitic dimensions, parasiticvals, to the Yagi
% antenna y, and uses the frequency freq and angle pair elang to calculate
% the objective function value.

% The yagi_objective_function2 function is used for an internal example.
% Its behavior might change in subsequent releases, so it should not be
% relied upon for programming purposes.

```

```

% Copyright 2014-2018 The MathWorks, Inc.

```

```

bw1 = elang(1);
bw2 = elang(2);
y.ReflectorLength = parasiticVals(1);
y.DirectorLength = parasiticVals(2:y.NumDirectors+1);
y.ReflectorSpacing = parasiticVals(y.NumDirectors+2);
y.DirectorSpacing = parasiticVals(y.NumDirectors+3:end);
output = calculate_objectives(y,freq,bw1,bw2);
output = output.MaxDirectivity + output.FB;
objectivevalue = -output; % To maximize
end

```

```

function output = calculate_objectives(y,freq,bw1,bw2)
%calculate_objectives calculate the objective function
% output = calculate_objectives(y,freq,bw1,bw2) Calculate the directivity
% in az = 90 plane that covers the main beam, sidelobe and backlobe.
% Calculate the maximum directivity, sidelobe level and backlobe and store
% in fields of the output variable structure.
[es,~,el] = pattern(y,freq,90,0:1:270);
el1 = el < bw1;
el2 = el > bw2;
el3 = el > bw1 & el < bw2;
emainlobe = es(el3);
esidelobes = ([es(el1);es(el2)]);
Dmax = max(emainlobe);
SLLmax = max(esidelobes);
Backlobe = es(end);
F = es(91);
B = es(end);
F_by_B = F-B;
output.MaxDirectivity = Dmax;
output.MaxSLL = SLLmax;
output.BackLobeLevel = Backlobe;
output.FB = F_by_B;
end

```

Set bounds on the control variables.

```

refLengthBounds = [0.4;
                   0.6];
dirLengthBounds = [0.35 0.35 0.35 0.35; % lower bound on director length
                   0.495 0.495 0.495 0.495]; % upper bound on director length
refSpacingBounds = [0.05; % lower bound on reflector spacing
                   0.30]; % upper bound on reflector spacing
dirSpacingBounds = [0.05 0.05 0.05 0.05; % lower bound on director spacing
                   0.23 0.23 0.23 0.23]; % upper bound on director spacing

```

```
LB = [refLengthBounds(1) dirLengthBounds(1,:) refSpacingBounds(1) dirSpacingBounds(1,:) ].*lambda;
UB = [refLengthBounds(2) dirLengthBounds(2,:) refSpacingBounds(2) dirSpacingBounds(2,:) ].*lambda;
```

Set the initial point for the optimization, and set the elevation beamwidth angle bounds.

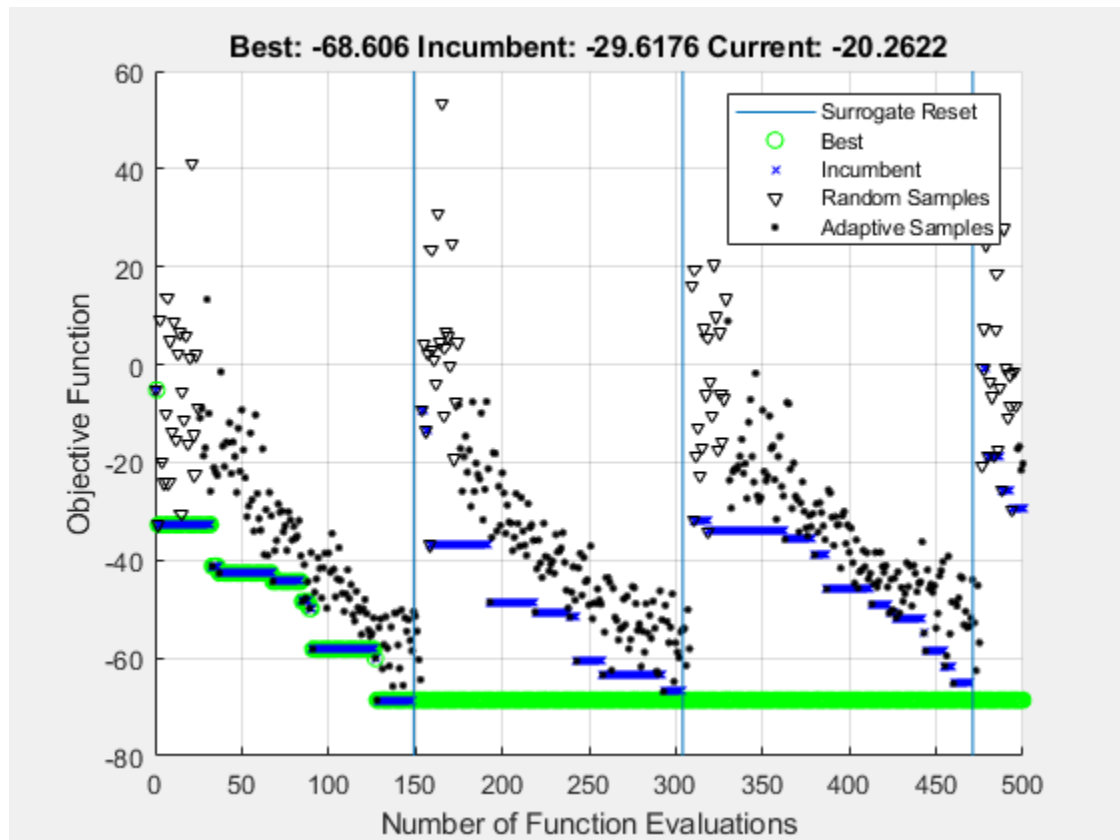
```
parasitic_values = [ yagidesign.ReflectorLength,           ...
                    yagidesign.DirectorLength,           ...
                    yagidesign.ReflectorSpacing,         ...
                    yagidesign.DirectorSpacing];
```

```
elang = [60 120]; % elevation beamwidth angles at az = 90
```

Surrogate Optimization

To search for a global optimum of the objective function, use `surrogateopt` as the solver. Set options to allow 500 function evaluations, include the initial point, use parallel computation, and use the `'surrogateoptplot'` plot function. To understand the `'surrogateoptplot'` plot, see "Interpret surrogateoptplot" on page 11-25..

```
surrogateoptions = optimoptions('surrogateopt','MaxFunctionEvaluations',500,...
    'InitialPoints',parasitic_values,'UseParallel',true,'PlotFcn','surrogateoptplot');
rng(4) % For reproducibility
optimdesign = surrogateopt(@(x) yagi_objective_function2(yagidesign,x,freq,elang),...
    LB,UB,surrogateoptions);
```



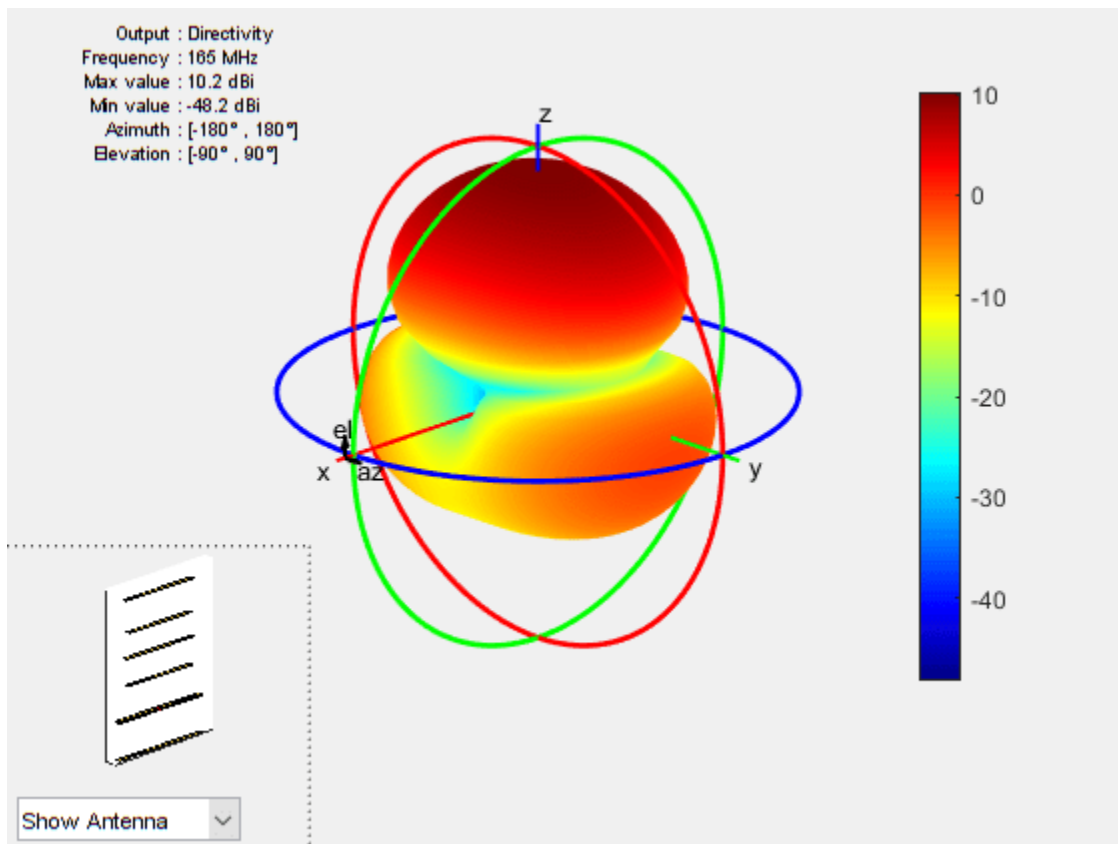
Surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

surrogateopt found a point giving an objective function value of -70. Investigate the effect of the optimized parameters on the radiation pattern of the antenna.

Plot Optimized Pattern

Plot the optimized antenna pattern at the design frequency.

```
yagidesign.ReflectorLength = optimdesign(1);
yagidesign.DirectorLength = optimdesign(2:5);
yagidesign.ReflectorSpacing = optimdesign(6);
yagidesign.DirectorSpacing = optimdesign(7:10);
fig2 = figure;
pattern(yagidesign,freq)
```

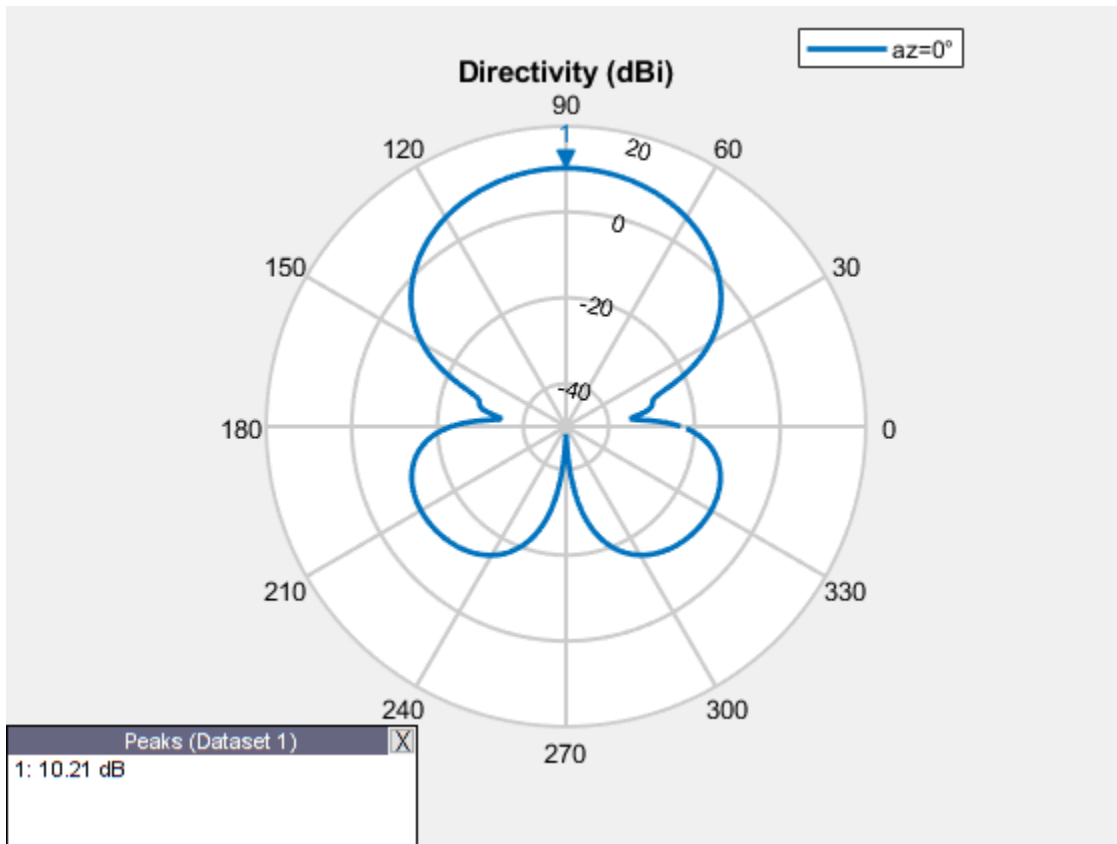


Apparently, the antenna now radiates significantly more power at zenith.

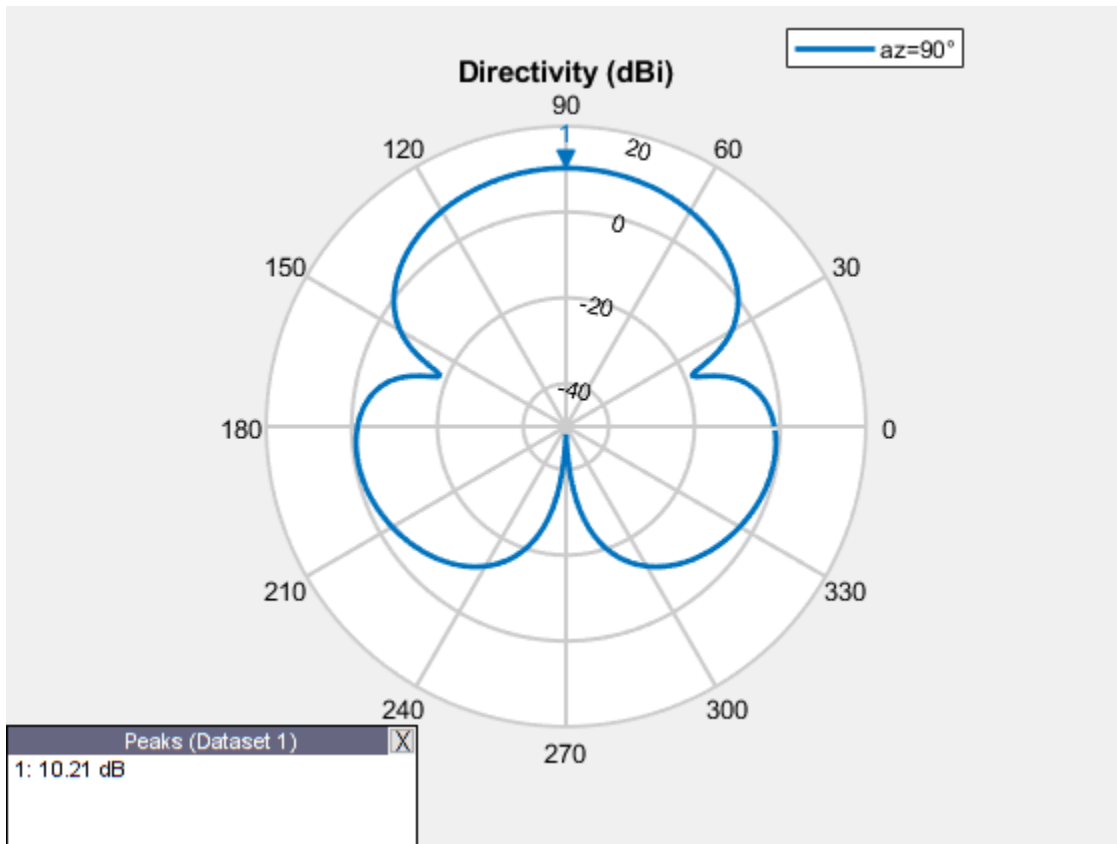
E-Plane and H-Plane Cuts of Pattern

To obtain a better insight into the behavior in two orthogonal planes, plot the normalized magnitude of the electric field in the E-plane and H-plane, that is, azimuth = 0 and 90 deg, respectively.

```
fig3 = figure;
pattern(yagidesign,freq,0,0:1:359);
```



```
fig4 = figure;  
pattern(yagidesign,freq,90,0:1:359);
```



The optimized design shows a significant improvement in the radiation pattern. Higher directivity is achieved in the desired direction toward zenith. The back lobe is small, resulting in a good front-to-back ratio for this antenna. Calculate the directivity at zenith, front-to-back ratio, and beamwidth in the E-plane and H-plane.

```
D_max = pattern(yagidesign,freq,0,90)
D_max = 10.2145
D_back = pattern(yagidesign,freq,0,-90)
D_back = -48.1770
F_B_ratio = D_max - D_back
F_B_ratio = 58.3915
Eplane_beamwidth = beamwidth(yagidesign,freq,0,1:1:360)
Eplane_beamwidth = 54
Hplane_beamwidth = beamwidth(yagidesign,freq,90,1:1:360)
Hplane_beamwidth = 68
```

Comparison with Manufacturer Datasheet

The optimized Yagi-Uda antenna achieves a forward directivity of 10.2 dBi, which translates to 8.1 dBd (relative to a dipole). This result is a bit less than the gain value reported by the datasheet in

reference [2] (8.5 dBd). The front-to-back ratio is 60 dB; this is part of the quantity that the optimizer maximizes. The optimized Yagi-Uda antenna has an E-plane beamwidth of 54 deg, whereas the datasheet lists the E-plane beamwidth as 56 deg. The H-plane beamwidth of the optimized Yagi-Uda antenna is 68 deg, whereas the value on the datasheet is 63 deg. The example does not address impedance matching over the band.

Tabulating Initial and Optimized Design

Tabulate the initial design guesses and the final optimized design values.

```
yagiparam= {'Reflector Length';
            'Director Length - 1'; 'Director Length - 2';
            'Director Length - 3'; 'Director Length - 4';
            'Reflector Spacing';  'Director Spacing - 1';
            'Director Spacing - 2'; 'Director Spacing - 3';
            'Director Spacing - 4'};
initialdesign = initialdesign';
optimdesign = optimdesign';
T = table(initialdesign,optimdesign,'RowNames',yagiparam)
```

T=10×2 table

	initialdesign	optimdesign
Reflector Length	0.90846	0.92703
Director Length - 1	0.90846	0.71601
Director Length - 2	0.90846	0.7426
Director Length - 3	0.90846	0.68847
Director Length - 4	0.90846	0.75779
Reflector Spacing	0.54508	0.3117
Director Spacing - 1	0.45423	0.28684
Director Spacing - 2	0.45423	0.23237
Director Spacing - 3	0.45423	0.21154
Director Spacing - 4	0.45423	0.27903

Reference

[1] Balanis, C. A. *Antenna Theory: Analysis and Design*. 3rd ed. New York: Wiley, 2005, p. 514.

[2] Online at: <https://amphenolprocom.com/products/base-station-antennas/2450-s-6y-165>

See Also

surrogateopt

More About

- “Surrogate Optimization”

Work with Checkpoint Files

In this section...

“Checkpoint for Restarting” on page 11-56

“Change Options to Extend or Monitor Optimization” on page 11-58

“Code for Robust Surrogate Optimization” on page 11-60

Checkpoint for Restarting

A checkpoint file contains data about the optimization process. To obtain a checkpoint file, use the `CheckpointFile` option.

One basic use of a checkpoint file is to resume an optimization when it stops prematurely. The cause of the premature stopping can be events such as a power failure or a crash, or when you press the **Stop** button in a plot function window.

Whatever the reason for the premature stopping, the restart procedure is simply to call `surrogateopt` with the checkpoint file name.

For example, suppose that you run an optimization with the 'check1' checkpoint file, and then click the **Stop** button soon after the optimization starts.

```
options = optimoptions('surrogateopt', 'CheckpointFile', 'check1.mat');  
lb = [-6, -8];  
ub = -lb;  
fun = @(x)100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;  
[x,fval,exitflag,output] = surrogateopt(fun,lb,ub,options)
```

Optimization stopped by a plot function or output function.

x =

```
    0    0
```

fval =

```
    1
```

exitflag =

```
   -1
```

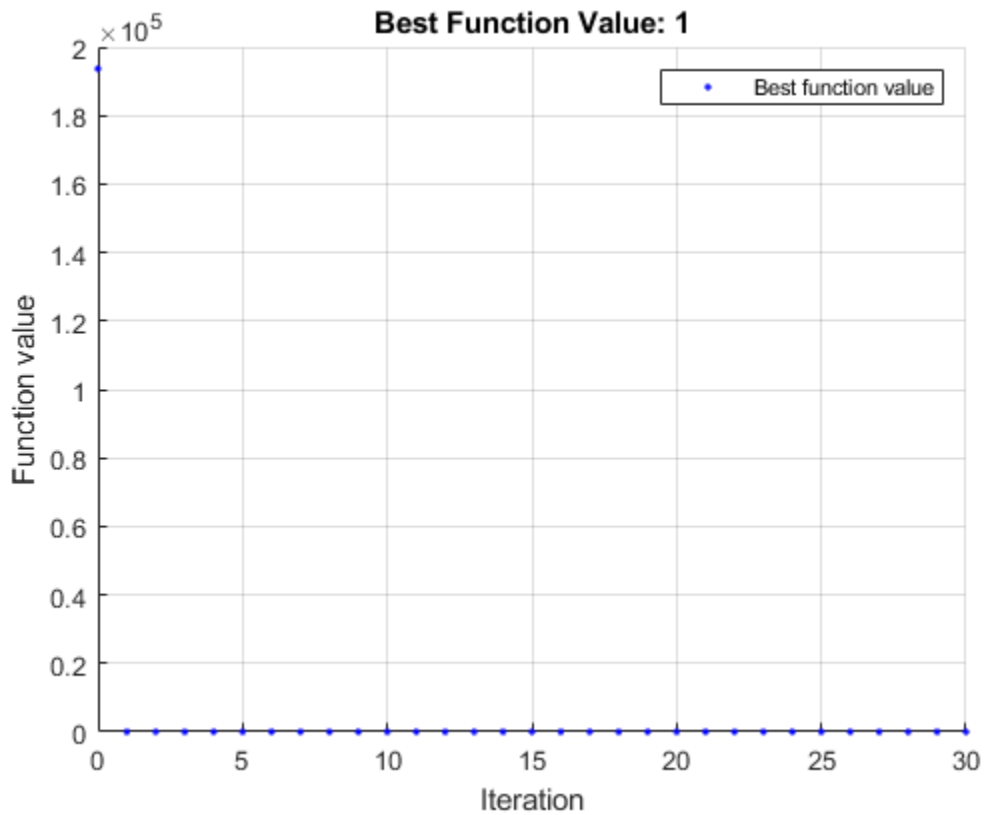
output =

struct with fields:

```
    elapsedtime: 15.3330  
    funccount: 30  
    constrviolation: 0  
    ineq: [1×0 double]
```



```
rngstate: [1x1 struct]
message: 'Optimization stopped by a plot function or output function.'
```



Note Checkpointing takes time. This overhead is especially noticeable for functions that otherwise take little time to evaluate.

To resume the optimization, call `surrogateopt` with the `'check1.mat'` argument.

```
[x,fval,exitflag,output] = surrogateopt('check1.mat')
```

```
Surrogateopt stopped because it exceeded the function evaluation limit set by
'options.MaxFunctionEvaluations'.
```

```
x =
```

```
    1.0186    1.0377
```

```
fval =
```

```
    3.4902e-04
```

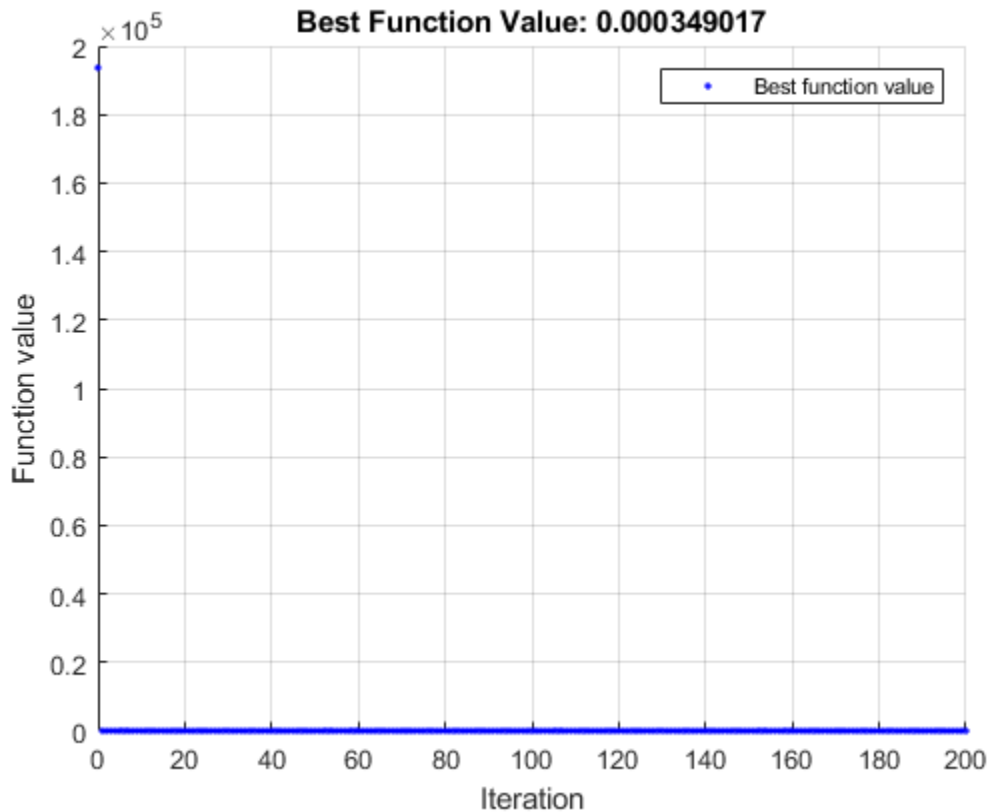
```
exitflag =
```

```
    0
```

```
output =
```

```
struct with fields:
```

```
    elapsedtime: 181.5824
    funccount: 200
    constrviolation: 0
        ineq: [1×0 double]
    rngstate: [1×1 struct]
    message: 'Surrogateopt stopped because it exceeded the function evaluation limit set
```



Change Options to Extend or Monitor Optimization

You can extend an optimization, whether it stops due to an unforeseen event or not, by changing the stopping criteria in the options. You can also monitor the optimization by displaying information at each iteration.

Note `surrogateopt` allows you to change only a limited set of options. For reliability, update the original options structure instead of creating new options.

For a list of the options you can change when restarting, see `opts`.

For example, suppose that you want to extend the previous optimization to run for a total of 400 function evaluations. Additionally, you want to monitor the optimization using the 'surrogateoptplot' plot function.

```
opts = optimoptions(options,'MaxFunctionEvaluations',400,...  
    'PlotFcn','surrogateoptplot');  
[x,fval,exitflag,output] = surrogateopt('check1.mat',opts)
```

Surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

x =

```
    1.0186    1.0377
```

fval =

```
    3.4902e-04
```

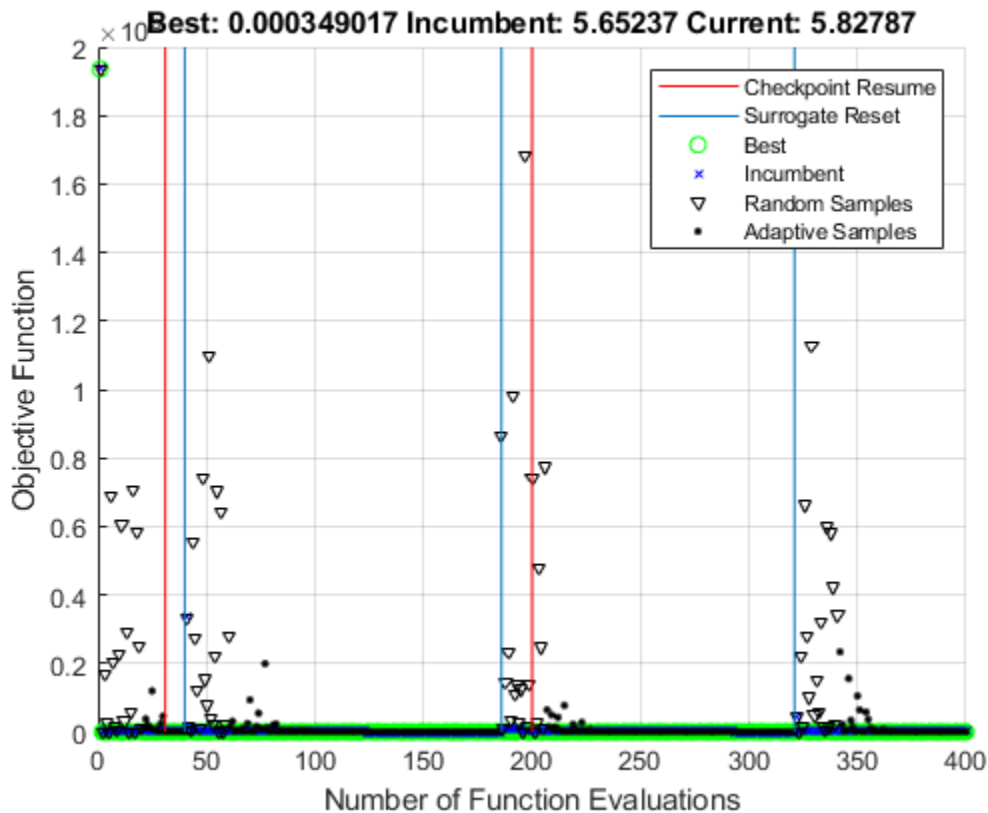
exitflag =

```
    0
```

output =

struct with fields:

```
    elapsedtime: 959.7619  
    funccount: 400  
    constrviolation: 0  
        ineq: [1×0 double]  
    rngstate: [1×1 struct]  
    message: 'Surrogateopt stopped because it exceeded the function evaluation limit set
```



The new plot function plots from the beginning of the optimization, even though you started the plot function only after the solver stopped at function evaluation number 200. The 'surrogateoptplot' plot function also shows the evaluation numbers where the optimization stopped and where it restarted from the checkpoint file.

Code for Robust Surrogate Optimization

To restart a surrogate optimization from a checkpoint file only if the file exists, use the following code logic. In this way, you can write scripts to keep an optimization going, even after a crash or other unexpected event.

```
% Assume that myfun, lb, and ub exist
if isfile('saveddata.mat')
    [x,fval,exitflag,output] = surrogateopt('saveddata.mat');
else
    options = optimoptions("surrogateopt","CheckpointFile",'saveddata.mat');
    [x,fval,exitflag,output] = surrogateopt(myfun,lb,ub,options);
end
```

See Also

surrogateopt

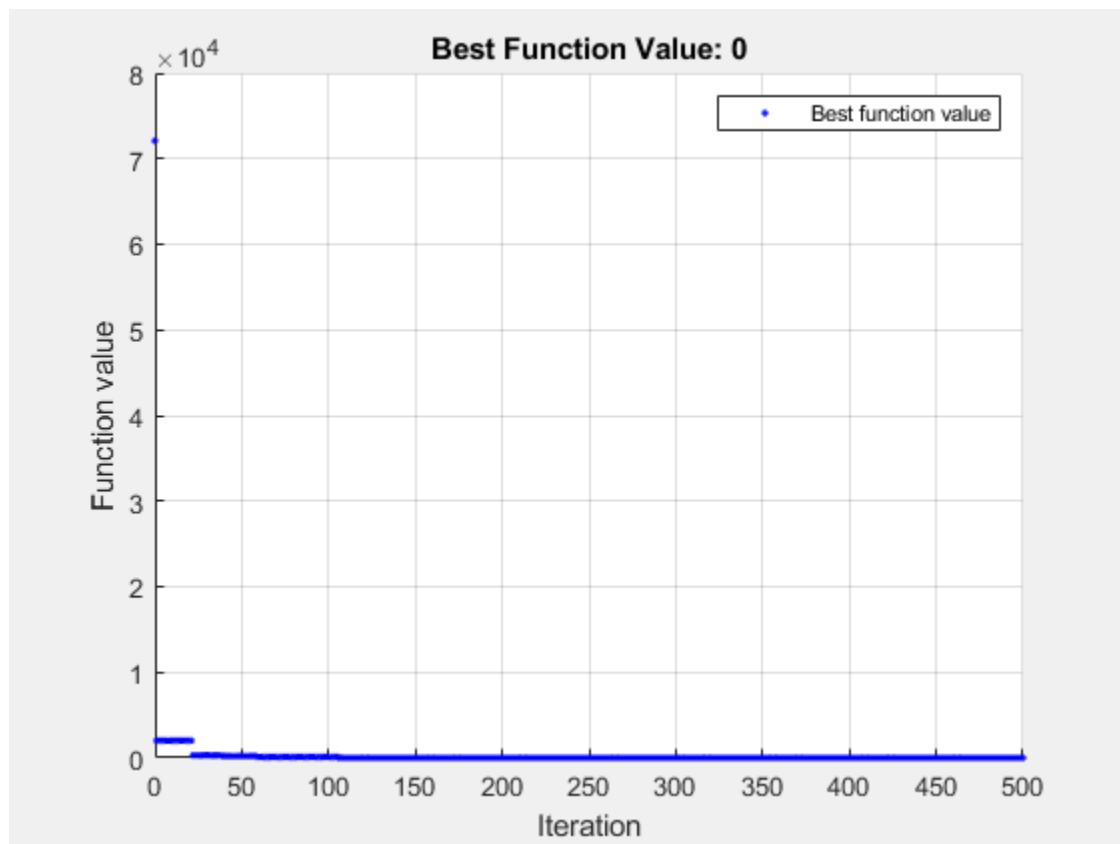
More About

- “Surrogate Optimization”
- “Surrogate Optimization Options” on page 17-51

Mixed-Integer Surrogate Optimization

This example shows how to solve an optimization problem that involves integer variables. Beginning in R2019b, `surrogateopt` accepts integer constraints. In this example, find the point x that minimizes the `multirosenbrock` function over integer-valued arguments ranging from -3 to 6 in ten dimensions. The `multirosenbrock` function is a poorly scaled function that is difficult to optimize. Its minimum value is 0 , which is attained at the point $[1, 1, \dots, 1]$. Code for the `multirosenbrock` function appears at the end of this example on page 11-63.

```
rng(1, 'twister') % For reproducibility
nvar = 10; % Any even number
lb = -3*ones(1,nvar);
ub = 6*ones(1,nvar);
fun = @multirosenbrock;
intcon = 1:nvar; % All integer variables
[sol,fval] = surrogateopt(fun,lb,ub,intcon)
```



`surrogateopt` stopped because it exceeded the function evaluation limit set by `'options.MaxFunctionEvaluations'`.

```
sol = 1x10
```

```
    1    1    1    1    1    1    1    1    1    1
```

```
fval = 0
```

In this case, `surrogateopt` finds the solution.

Helper Function

This code creates the `multirosenbrock` helper function.

```
function F = multirosenbrock(x)
% This function is a multidimensional generalization of Rosenbrock's
% function. It operates in a vectorized manner, assuming that x is a matrix
% whose rows are the individuals.
% Copyright 2014 by The MathWorks, Inc.
N = size(x,2); % assumes x is a row vector or 2-D matrix
if mod(N,2) % if N is odd
    error('Input rows must have an even number of elements')
end
odds = 1:2:N-1;
evens = 2:2:N;
F = zeros(size(x));
F(:,odds) = 1-x(:,odds);
F(:,evens) = 10*(x(:,evens)-x(:,odds).^2);
F = sum(F.^2,2);
end
```

See Also

`surrogateopt`

More About

- “Surrogate Optimization”
- “Mixed Integer ga Optimization” on page 8-40

Fix Variables in surrogateopt

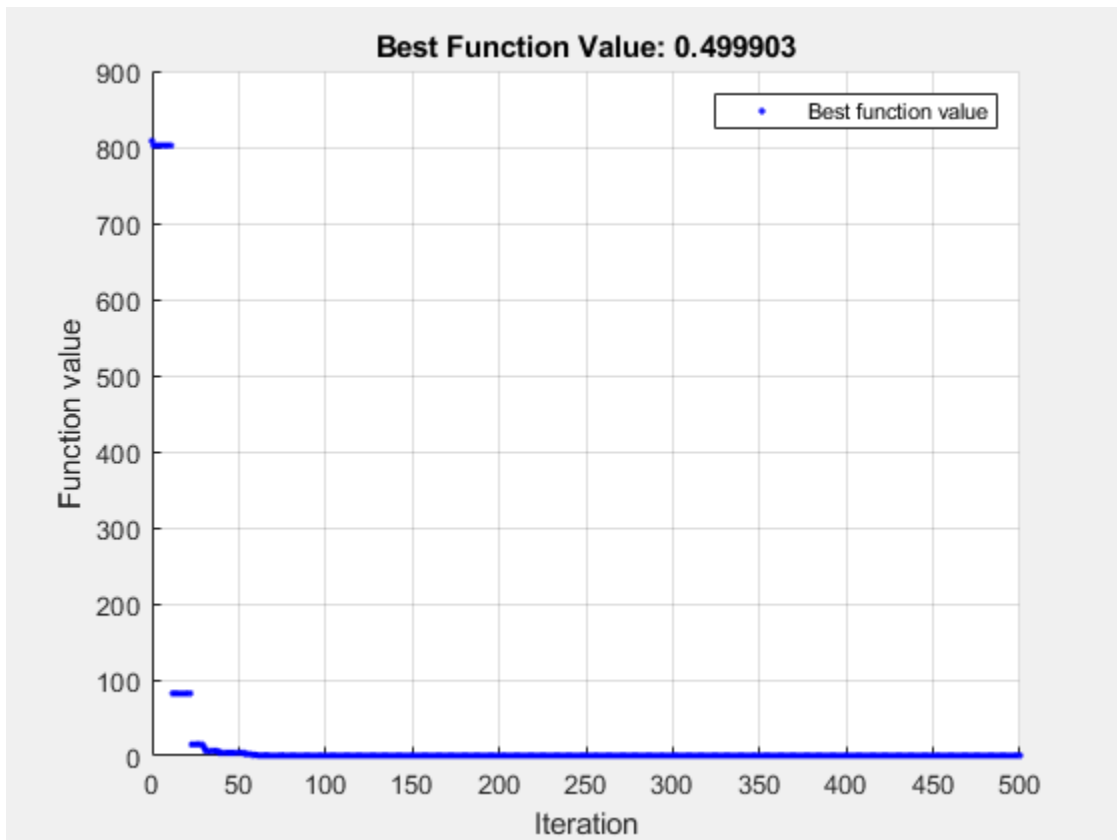
This example shows how to fix the values of some control variables, by removing them from an optimization. Although the easiest way to fix values is to set equal upper and lower bounds, some solvers do not allow equal bounds. However, `surrogateopt` handles equal bounds well by internally removing fixed variables from the problem before trying to optimize.

The `multirosenbrock` function accepts any even number of control variables. Its minimum value of 0 is attained at the point $[1, 1, \dots, 1, 1]$. Set lower bounds of -1 and upper bounds of 5 for ten variables, and then set the first six upper and lower bounds equal to 1. This setting removes six variables from the problem, leaving a problem with four variables.

```
lb = -1*ones(1,10);
ub = 5*ones(1,10);
lb(1:6) = 1;
ub(1:6) = 1;
```

Solve the problem.

```
fun = @multirosenbrock;
rng default % For reproducibility
[x,fval,exitflag] = surrogateopt(fun,lb,ub)
```



`surrogateopt` stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.


```
x = 1×10
```

```
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 0.5195 0.2679 1.5180 2.1
```

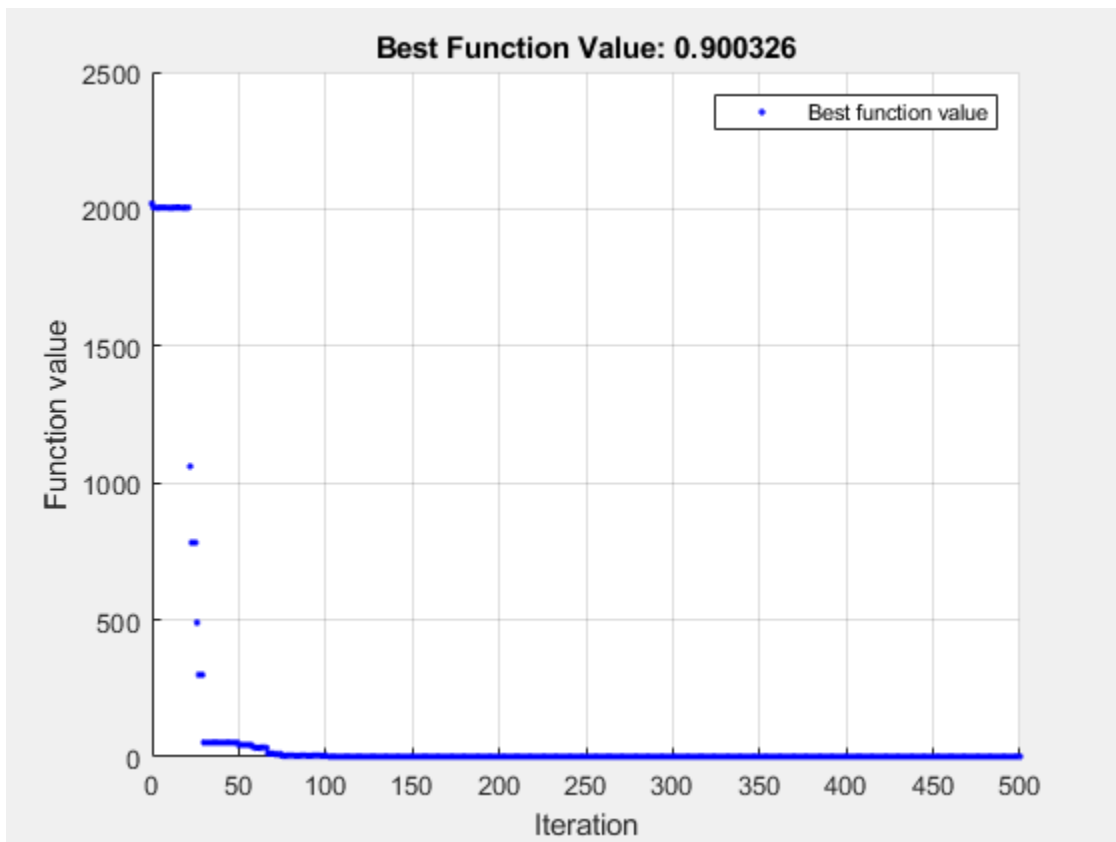
```
fval = 0.4999
```

```
exitflag = 0
```

The solver returns a point close to the global minimum. Notice that the solver takes 500 function evaluations, which is the default value for a problem with 10 variables. The solver does not change this default value even when you fix some variables.

When you do not fix any variables, the solver does not reach a point near the global minimum.

```
lb = -1*ones(1,10);
ub = 5*ones(1,10);
rng default % For reproducibility
[x,fval,exitflag] = surrogateopt(fun,lb,ub)
```



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
x = 1×10
```

```
1.4639 2.1451 1.3645 1.8633 1.4170 2.0147 1.4286 2.0438 1.4343 2.0
```

```
fval = 0.9003
```

```
exitflag = 0
```

See Also

surrogateopt

More About

- “Surrogate Optimization”

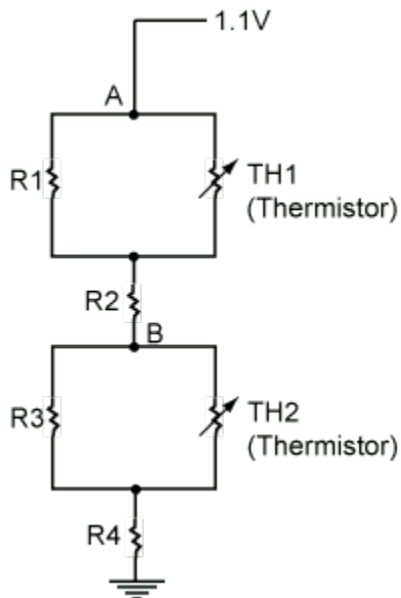
Optimal Component Choice Using surrogateopt

This example shows how to choose the resistors and thermistors in a circuit to best match a specified curve at one point in the circuit. You must choose all of the electronic components from a list of available components, which means this is a discrete optimization problem. To help visualize the progress of the optimization, the example includes a custom output function that displays the quality of the intermediate solutions as the optimization progresses. Because this is an integer problem with a nonlinear objective function, use the `surrogateopt` solver.

This example is adapted from Lyon [1].

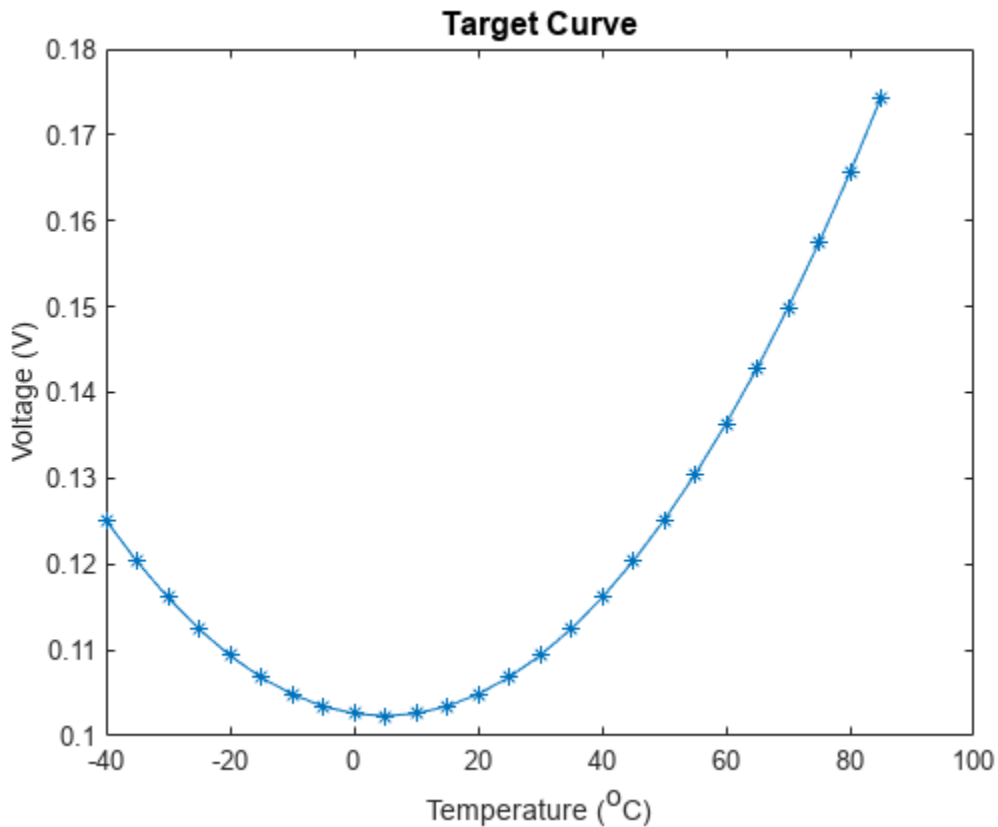
Problem Description

The problem involves this circuit.



A voltage source holds point A at 1.1V. The problem is to select resistors and thermistors from a list of standard components so that the voltage at point B matches the target curve as a function of temperature.

```
Tdata = -40:5:85;
Vdata = 1.026E-1 + -1.125E-4 * Tdata + 1.125E-5 * Tdata.^2;
plot(Tdata,Vdata,'-*');
title('Target Curve','FontSize',12);
xlabel('Temperature (^oC)'); ylabel('Voltage (V)')
```



Load the standard components list.

load `StandardComponentValues`

The `Res` vector contains the standard resistor values. The `ThBeta` and `ThVal` vectors contain standard parameters for the thermistors. Thermistor resistance as a function of temperature T is

$$R_{Th} = \frac{R_{25}}{\exp\left(\beta \frac{T - T_{25}}{T \cdot T_{25}}\right)}$$

- R_{Th} is the thermistor resistance.
- R_{25} is the resistance at 25 degrees Celsius, parameter `ThVal`.
- T_{25} is the temperature 25 degrees Celsius.
- T is the current temperature.
- β is the thermistor parameter `ThBeta`.

Based on standard voltage calculations, the equivalent series values of the resistances of the $R_1 - Th_1$ block is

$$R_1^{\text{equivalent}} = \frac{R_1 Th_1}{R_1 + Th_1},$$

and the equivalent resistance of the $R_3 - Th_2$ block is

$$R_3^{\text{equivalent}} = \frac{R_3 \text{Th}_2}{R_3 + \text{Th}_2}.$$

Therefore, the voltage at point B is

$$V = 1.1 \frac{R_3^{\text{equivalent}} + R_4}{R_1^{\text{equivalent}} + R_2 + R_3^{\text{equivalent}} + R_4}.$$

Convert Problem to Code

The problem is to choose resistors R_1 through R_4 and thermistors Th_1 and Th_2 so that the voltage V best matches the target curve. Have the control variable x represent these values:

- $x(i)$ = index of R_i , for i from 1 through 4
- $x(5)$ = index of Th_1
- $x(6)$ = index of Th_2

The `tempCompCurve` function calculates the resulting voltage in terms of x and the temperature `Tdata`.

type `tempCompCurve`

```
function F = tempCompCurve(x,Tdata)
%% Calculate Temperature Curve given Resistor and Thermistor Values
% Copyright (c) 2012-2019, MathWorks, Inc.
%% Input voltage
Vin = 1.1;

%% Thermistor Calculations
% Values in x: R1 R2 R3 R4 RTH1(T_25degc) Beta1 RTH2(T_25degc) Beta2
% Thermistors are represented by:
%   Room temperature is 25degc: T_25
%   Standard value is at 25degc: RTHx_25
%   RTHx is the thermistor resistance at various temperatures
% RTH(T) = RTH(T_25degc) / exp (Beta * (T-T_25)/(T*T_25))
T_25 = 298.15;
T_off = 273.15;
Beta1 = x(6);
Beta2 = x(8);
RTH1 = x(5) ./ exp(Beta1 * ((Tdata+T_off)-T_25)./((Tdata+T_off)*T_25));
RTH2 = x(7) ./ exp(Beta2 * ((Tdata+T_off)-T_25)./((Tdata+T_off)*T_25));

%% Define equivalent circuits for parallel Rs and RTHs
R1_eq = x(1)*RTH1./(x(1)+RTH1);
R3_eq = x(3)*RTH2./(x(3)+RTH2);

%% Calculate voltages at Point B
F = Vin * (R3_eq + x(4))./(R1_eq + x(2) + R3_eq + x(4));
```

The objective function is the sum of squares of the differences between the target curve and the resulting voltages for a set of resistors and thermistors, over the target range of temperatures.

type `objectiveFunction`

```
function G = objectiveFunction(x,StdRes, StdTherm_Val, StdTherm_Beta,Tdata,Vdata)
%% Objective function for the thermistor problem
```

```
% Copyright (c) 2012-2019, MathWorks, Inc.

% % StdRes = vector of resistor values
% StdTherm_val = vector of nominal thermistor resistances
% StdTherm_Beta = vector of thermistor temperature coefficients

% Extract component values from tables using integers in x as indices
y = zeros(8,1);
x = round(x); % in case of noninteger components
y(1) = StdRes(x(1));
y(2) = StdRes(x(2));
y(3) = StdRes(x(3));
y(4) = StdRes(x(4));
y(5) = StdTherm_Val(x(5));
y(6) = StdTherm_Beta(x(5));
y(7) = StdTherm_Val(x(6));
y(8) = StdTherm_Beta(x(6));

% Calculate temperature curve for a particular set of components
F = tempCompCurve(y, Tdata);

% Compare simulated results to target curve
Residual = F(:) - Vdata(:);
Residual = Residual(1:2:26);
%%
G = Residual'*Residual; % sum of squares
```

Monitor Progress

To observe the progress of an optimization, call an output function that plots the best response of the system found so far and the target curve. The `SurrOptimPlot` function plots these curves, and updates the curves only when the current objective function value decreases. This custom output function is lengthy, so it is not shown here. To see the content of this output function, enter type `SurrOptimPlot`.

Optimize Problem

To optimize the objective function, use `surrogateopt`, which accepts integer variables. First, set all variables to be integer.

```
intCon = 1:6;
```

Set the lower bounds on all variables to 1.

```
lb = ones(1,6);
```

The upper bounds for the resistors are all the same. Set the upper bounds to the number of entries in the `Res` data.

```
ub = length(Res)*ones(1,6);
```

Set the upper bounds for the thermistors to the number of entries in the `ThBeta` data.

```
ub(5:6) = length(ThBeta)*[1,1];
```

Set options to use the `SurrOptimPlot` custom output function, and to use no plot function. Also, to protect against possible interruptions of the optimization, specify a checkpoint file named `'checkfile.mat'`.

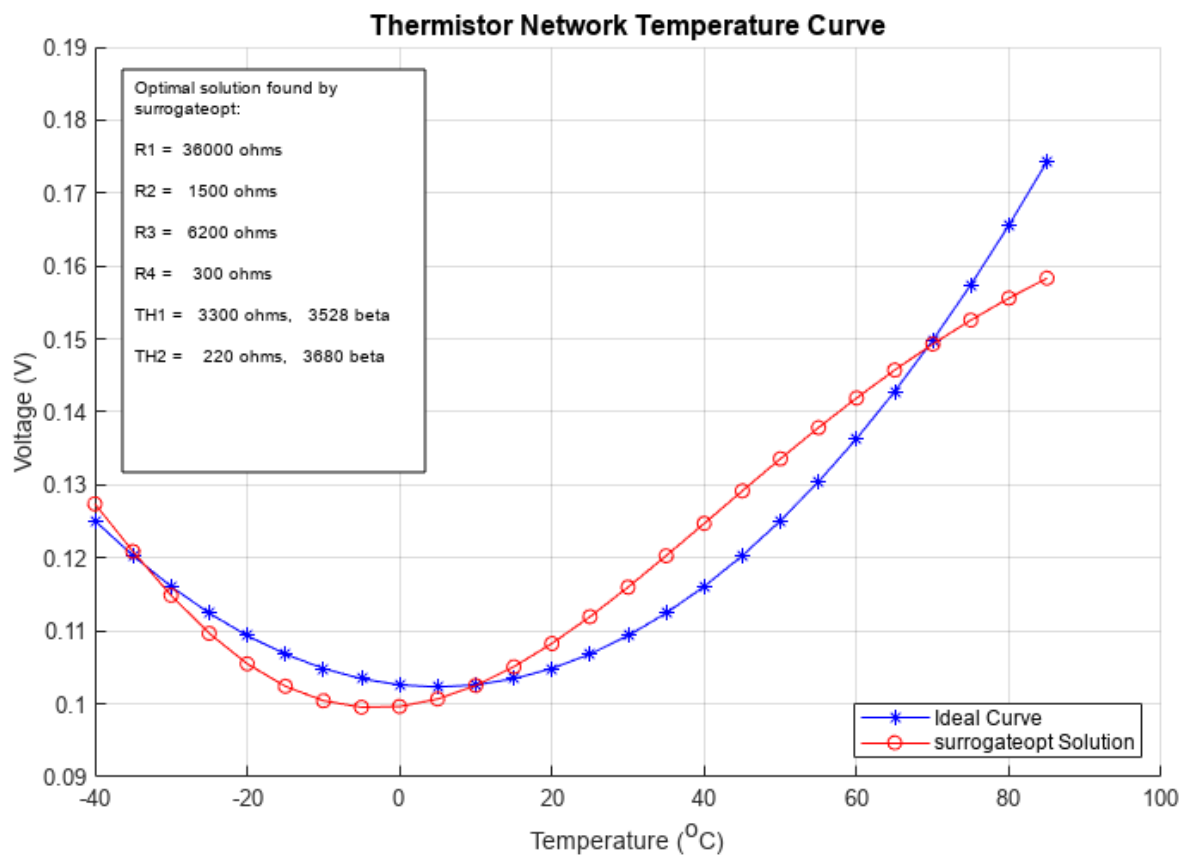
```
options = optimoptions('surrogateopt','CheckpointFile','C:\TEMP\checkfile.mat','PlotFcn',[],...
    'OutputFcn',@(a1,a2,a3)SurrOptimPlot(a1,a2,a3,Tdata,Vdata,Res,ThVal,ThBeta));
```

To give the algorithm a better initial set of points to search, specify a larger initial random sample than the default.

```
options.MinSurrogatePoints = 50;
```

Run the optimization.

```
rng default % For reproducibility
objconstr = @(x)objectiveFunction(x,Res,ThVal,ThBeta,Tdata,Vdata);
[xOpt,Fval] = surrogateopt(objconstr,lb,ub,intCon,options);
```

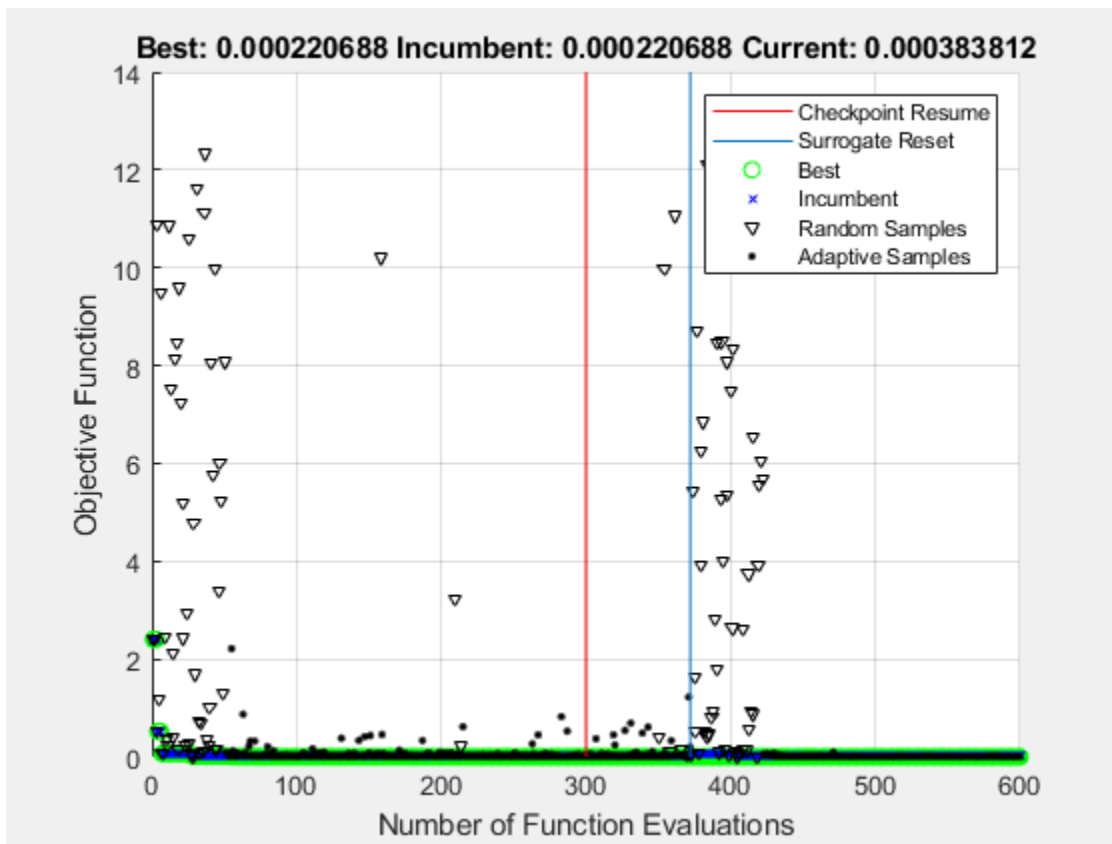


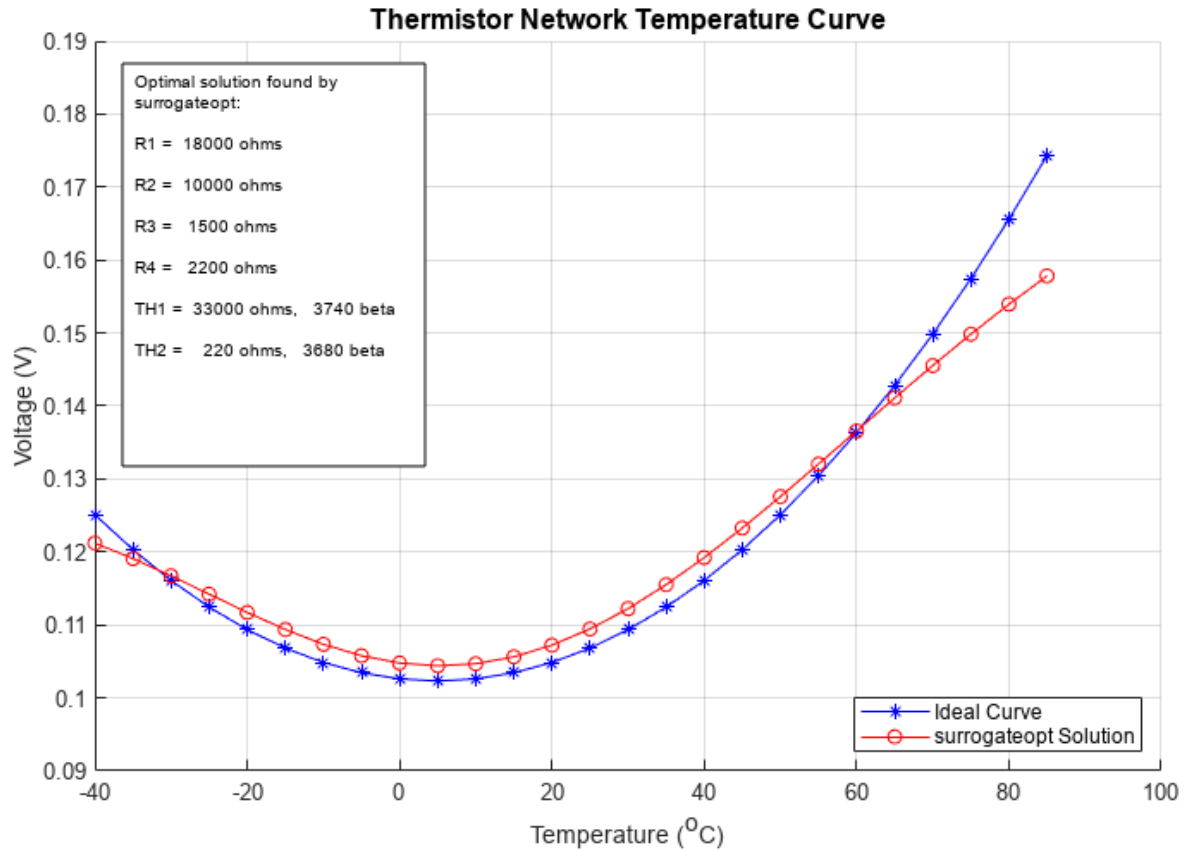
surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

Optimize with More Function Evaluations

To attempt to get a better fit, restart the optimization from the checkpoint file, and specify more function evaluations. This time, use the `surrogateoptplot` plot function to monitor the optimization process more closely.

```
clf % Clear previous figure
opts = optimoptions(options,'MaxFunctionEvaluations',600,'PlotFcn','surrogateoptplot');
[xOpt,Fval] = surrogateopt('C:\TEMP\checkfile.mat',opts);
```





surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

Using more function evaluations improves the fit slightly.

References

[1] Lyon, Craig K. *Genetic algorithm solves thermistor-network component values*. EDN Network, March 19, 2008. Available at <https://www.edn.com/genetic-algorithm-solves-thermistor-network-component-values/>.

See Also

surrogateopt

More About

- “Surrogate Optimization”

Convert Nonlinear Constraints Between surrogateopt Form and Other Solver Forms

Why Convert Constraint Forms?

To try various solvers including `surrogateopt` on a problem that has nonlinear inequality constraints, you must convert between the form required by `surrogateopt` and the form required by other solvers.

Convert from surrogateopt Structure Form to Other Solvers

The objective function `objconstr(x)` for `surrogateopt` returns a structure. The `Fval` field contains the objective function value, a scalar. The `Ineq` field contains a vector of constraint function values. The solver attempts to make all values in the `Ineq` field be less than or equal to zero. Positive values indicate a constraint violation.

Other solvers expect the objective function to return a scalar value, not a structure. Other solvers also expect the nonlinear constraint function to return two outputs, `c(x)` and `ceq(x)`, not a structure containing `c(x)`.

To convert the `surrogateopt` function `objconstr(x)` for use in other solvers:

- Set the objective function to `@(x) objconstr(x).Fval`.
- Set the nonlinear constraint function to `@(x) deal(objconstr(x).Ineq, [])`.

For example,

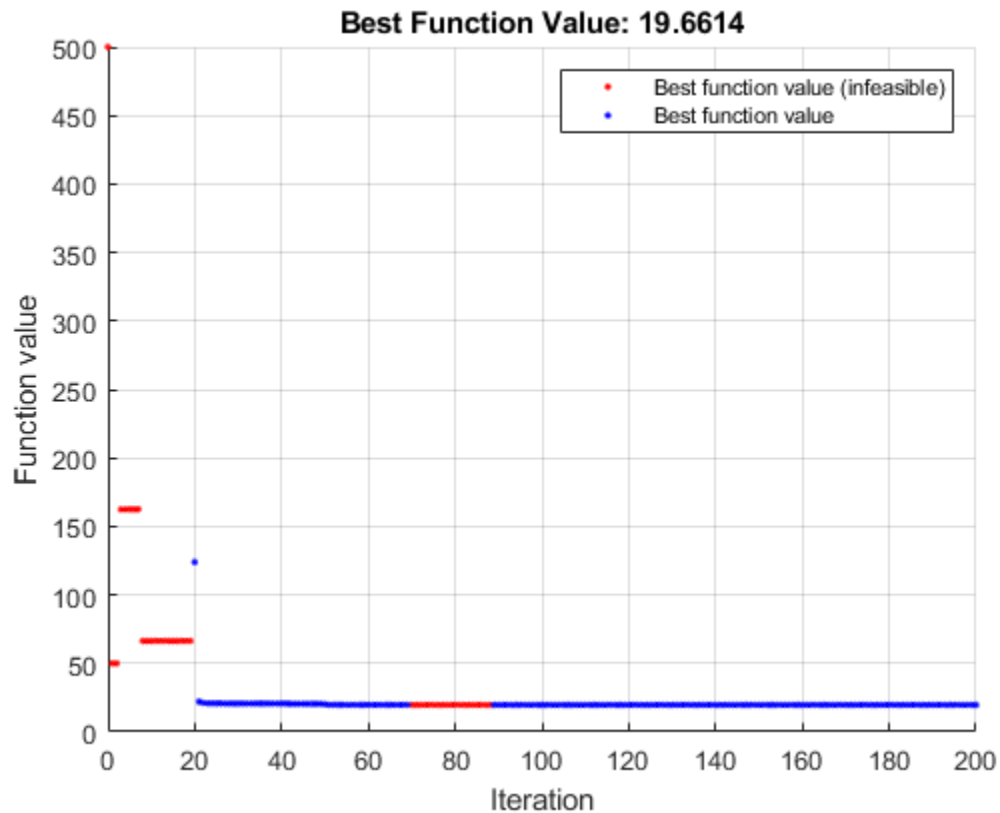
```
function ff = objconstr(x)
ff.Fval = norm(x)^2;
ff.Ineq = norm(x - [5,8])^2 - 25;
end
```

To solve a constrained minimization problem using `objconstr`, call `surrogateopt`.

```
lb = [-10, -20];
ub = [20, 10];
sol = surrogateopt(@objconstr, lb, ub)
```

```
Surrogateopt stopped because it exceeded the function evaluation limit set by
'options.MaxFunctionEvaluations'.
```

```
sol =
    2.3325    3.7711
```



To solve the same problem using `fmincon`, split the objective and constraint into separate functions. Include the nonlinear equality constraint as `[]` by using the `deal` function.

```
objfcn = @(x)objconstr(x).Fval;
nlcon = @(x)deal(objconstr(x).Ineq,[]);
```

Call `fmincon` with the objective function `objfcn` and nonlinear constraint function `nlcon`.

```
[solf,fvalf,eflag,output] = ...
    fmincon(objfcn,[0,0],[],[],[],[],[],lb,ub,nlcon)
```

Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in feasible directions, to within the value of the optimality tolerance, and constraints are satisfied to within the value of the constraint tolerance.

```
solf =
```

```
    2.3500    3.7600
```

```
fvalf =
```

```
    19.6602
```

```
eflag =
```

```

1

output =

    struct with fields:

        iterations: 7
        funcCount: 24
        constrviolation: 0
        stepsize: 2.0395e-05
        algorithm: 'interior-point'
        firstorderopt: 4.9651e-06
        cgiterations: 0
        message: 'Local minimum found that satisfies the constraints. Optimization complete'

```

You can also use `patternsearch` or `ga` to solve the problem using the same conversion.

Convert from Other Solvers to surrogateopt Structure Form

If you have a problem written in the form for other solvers, use the `packfcn` function to convert the objective and nonlinear constraints to the structure form for `surrogateopt`. If the objective function is a function handle `@obj` and the nonlinear constraint function is `@nlconst`, then use the objective function `objconstr` for `surrogateopt`.

```
objconstr = packfcn(@obj,@nlconst);
```

In this example, the objective function is Rosenbrock's function.

```
ros = @(x)100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;
```

Specify the constraint function to restrict the solution to lie inside a disk of radius 1/3 centered at the point [1/3,1/3].

```
function [c,ceq] = circlecon(x)
c = (x(1)-1/3)^2 + (x(2)-1/3)^2 - (1/3)^2;
ceq = [];
```

Set bounds of -2 and 2 on each component.

```
lb = [-2,-2];
ub = [2,2];
```

Solve the problem using `patternsearch` starting from [0,0].

```
x0 = [0,0];
x = patternsearch(ros,x0,[],[],[],[],lb,ub,@circlecon)
```

```
Optimization terminated: mesh size less than options.MeshTolerance
and constraint violation is less than options.ConstraintTolerance.
```

```
x =

    0.6523    0.4258
```

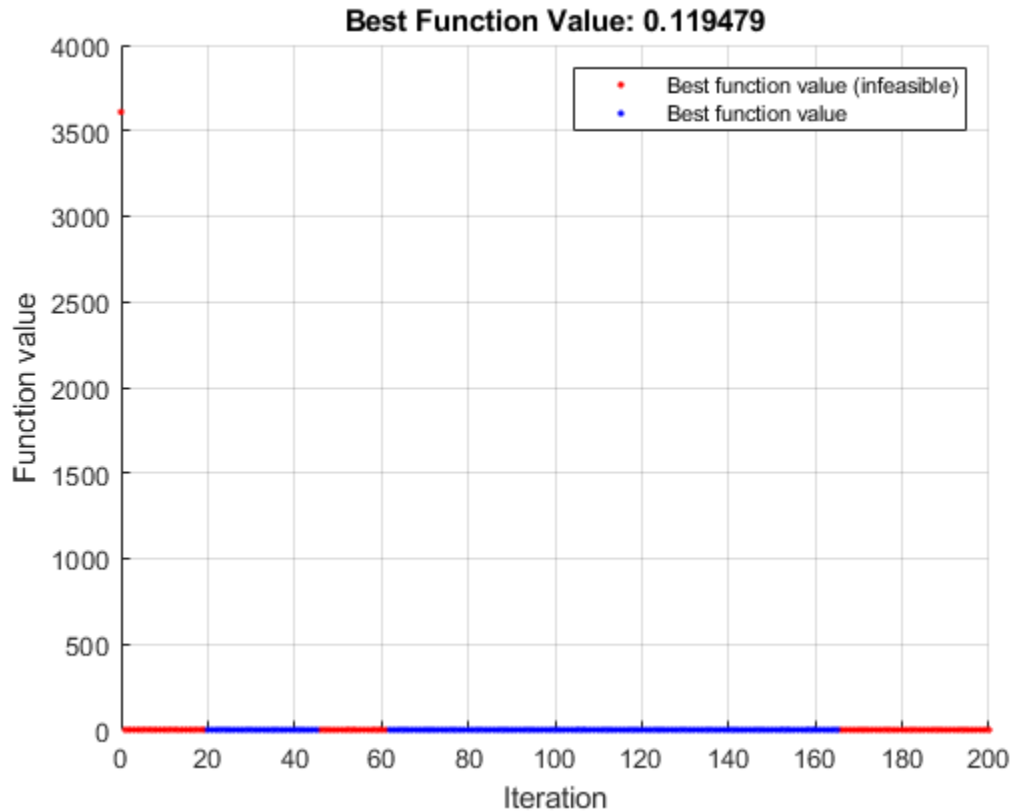
Convert the problem for solution by `surrogateopt`.

```
objconstr = packfcn(ros,@circlecon);  
xs = surrogateopt(objconstr,lb,ub)
```

Surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

xs =

```
0.6543 0.4284
```



See Also

surrogateopt | packfcn

More About

- “Surrogate Optimization”

Solve Feasibility Problem

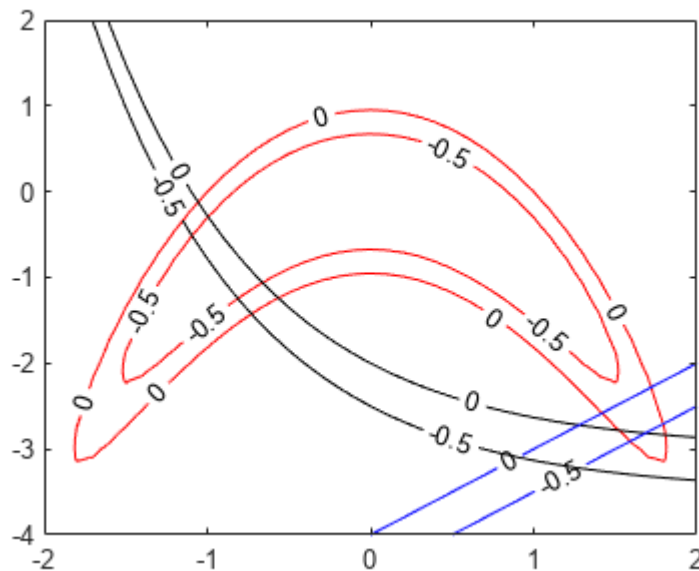
Some problems require you to find a point that satisfies all constraints, with no objective function to minimize. For example, suppose that you have the following constraints:

$$\begin{aligned}(y + x^2)^2 + 0.1y^2 &\leq 1 \\ y &\leq \exp(-x) - 3 \\ y &\leq x - 4.\end{aligned}$$

Do any points (x, y) satisfy the constraints? To find out, write a function that returns the constraints in a structure field `Ineq`. Write the constraints in terms of a two-element vector $x = (x_1, x_2)$ instead of (x, y) . Write each inequality as a function $c(x)$, meaning the inequalities $c(x) \leq 0$, by subtracting the right side of each inequality from both sides. To enable plotting, write the function in a vectorized manner, where each row represents one point. The code for this helper function, named `objconstr`, appears at the end of this example on page 11-81.

Plot the points where the three functions satisfy equalities for $-2 \leq x \leq 2$ and $-4 \leq y \leq 2$, and indicate the inequalities by plotting level lines for function values equal to $-1/2$.

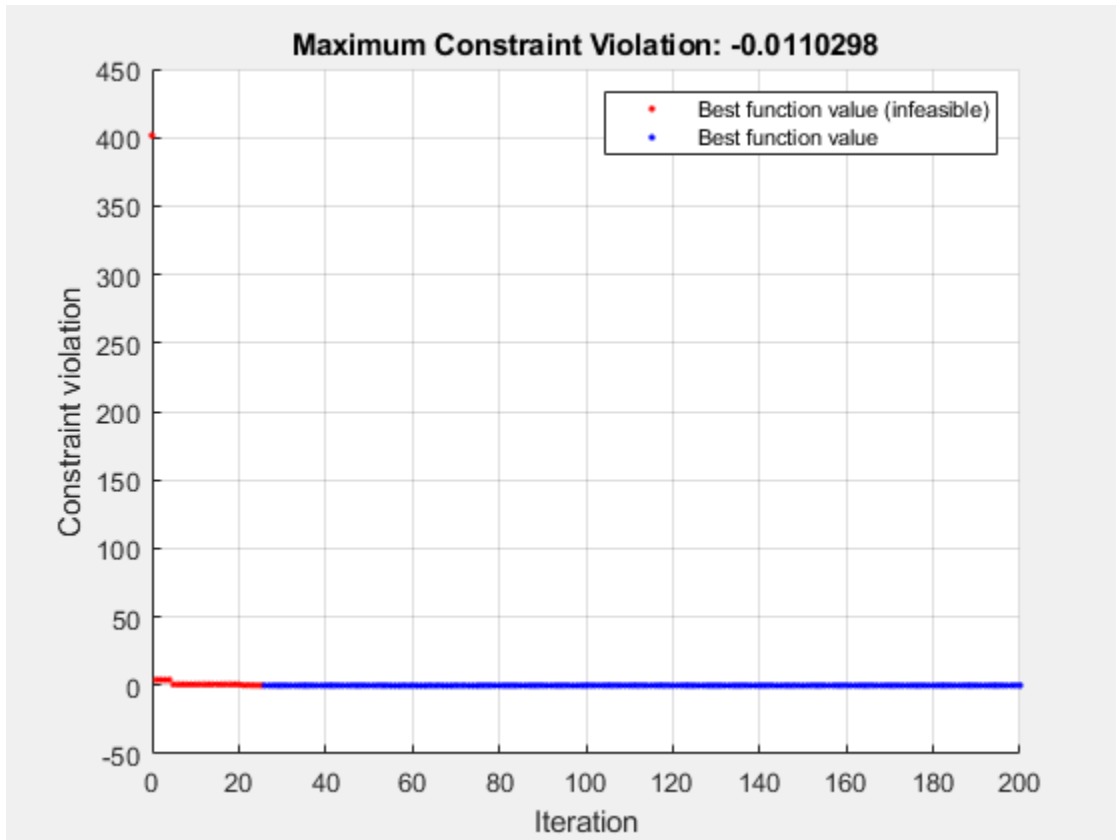
```
[XX,YY] = meshgrid(-2:0.1:2,-4:0.1:2);
ZZ = objconstr([XX(:),YY(:)]).Ineq;
ZZ = reshape(ZZ,[size(XX),3]);
h = figure;
ax = gca;
contour(ax,XX,YY,ZZ(:,:,1),[-1/2 0], 'r', 'ShowText', 'on');
hold on
contour(ax,XX,YY,ZZ(:,:,2),[-1/2 0], 'k', 'ShowText', 'on');
contour(ax,XX,YY,ZZ(:,:,3),[-1/2 0], 'b', 'ShowText', 'on');
hold off
```



The plot shows that feasible points exist near $[1.75, -3]$.

Set lower bounds of -5 and upper bounds of 3, and solve the problem using surrogateopt.

```
rng(1) % For reproducibility
lb = [-5,-5];
ub = [3,3];
[x,fval,exitflag,output,trials] = surrogateopt(@objconstr,lb,ub)
```



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
x = 1x2
```

```
    1.7964    -3.1296
```

```
fval =
```

```
    1x0 empty double row vector
```

```
exitflag = 0
```

```
output = struct with fields:
```

```
    elapsedtime: 37.4141
```

```
    funccount: 200
```

```
    constrviolation: -0.0110
```

```
        ineq: [-0.0110 -0.2955 -0.9261]
```

```
    rngstate: [1x1 struct]
```

```
    message: 'surrogateopt stopped because it exceeded the function evaluation limit set
```

```

trials = struct with fields:
    X: [200x2 double]
    Ineq: [200x3 double]

```

Check the feasibility at the returned solution x .

```

disp(output.ineq)
-0.0110 -0.2955 -0.9261

```

Equivalently, evaluate the function `objconstr` at the returned solution x .

```

disp(objconstr(x).Ineq)
-0.0110 -0.2955 -0.9261

```

Equivalently, examine the `Ineq` field in the `trials` structure for the solution x . First, find the index of x in the `trials.X` field.

```

indx = ismember(trials.X,x,'rows');
disp(trials.Ineq(indx,:))
-0.0110 -0.2955 -0.9261

```

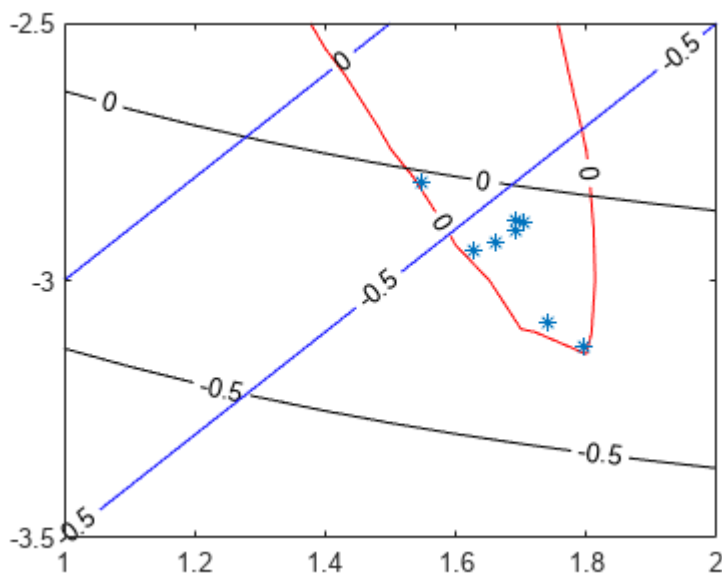
All constraint function values are negative, indicating that the point x is feasible.

View the feasible points evaluated by `surrogateopt`.

```

opts = optimoptions("surrogateopt");
indx = max(trials.Ineq,[],2) <= opts.ConstraintTolerance; % Indices of feasible points
figure(h);
hold on
plot(trials.X(indx,1),trials.X(indx,2),'*')
xlim([1 2])
ylim([-3.5 -2.5])
hold off

```



This code creates the `objconstr` helper function.

```
function f = objconstr(x)
c(:,1) = (x(:,2) + x(:,1).^2).^2 + 0.1*x(:,2).^2 - 1;
c(:,2) = x(:,2) - exp(-x(:,1)) + 3;
c(:,3) = x(:,2) - x(:,1) + 4;
f.Ineq = c;
end
```

See Also

`surrogateopt`

More About

- “Solve Nonlinear Feasibility Problem, Problem-Based”
- “Investigate Linear Infeasibilities”

Solve Nonlinear Problem with Integer and Nonlinear Constraints

The `surrogateopt` solver accepts both integer constraints and nonlinear constraints. Compare the solution of a nonlinear problem both with and without integer constraints. The integer constraints cause the solution to lie on a reasonably fine grid.

Objective and Constraint Functions

The objective function is

$$f(x) = \log(1 + 3(x_2 - (x_1^3 - x_1))^2 + (x_1 - 4/3)^2).$$

This objective function is nonnegative, and takes its minimum value of 0 at the point $x = [4/3, (4/3)^3 - 4/3] = [1.3333, 1.0370]$.

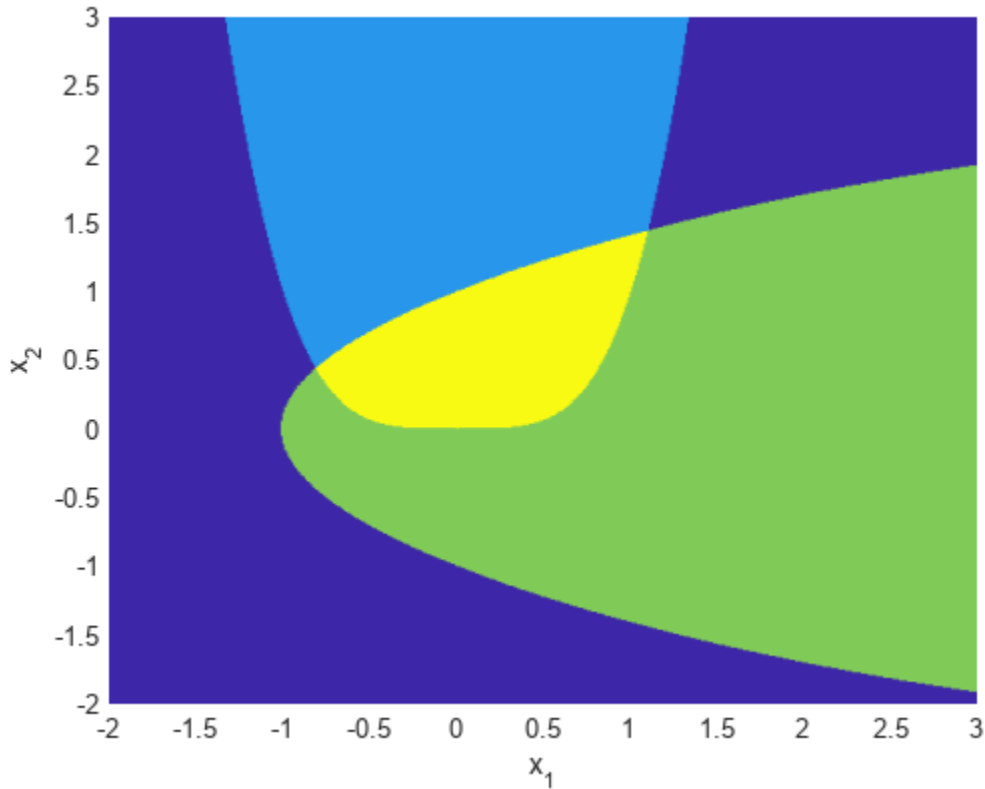
The problem has two nonlinear constraint functions.

$$x_1^4 \leq 5\sinh(x_2/5),$$

$$x_2^2 \leq 5\tanh(x_1/5) + 1.$$

Plot the feasible region for the nonlinear constraints.

```
[X,Y] = meshgrid(-2:.01:3);
Z = (5*sinh(Y./5) >= X.^4);
% Z=1 where the first constraint is satisfied, Z=0 otherwise
Z = Z + 2*(5*tanh(X./5) >= Y.^2 - 1);
% Z=2 where the second constraint is satisfied
% Z=3 where both constraints are satisfied
surf(X,Y,Z, 'LineStyle', 'none');
fig = gcf;
fig.Color = 'w'; % white background
view(0,90)
xlabel('x_1')
ylabel('x_2')
```



The yellow region shows where both constraints are satisfied.

`surrogateopt` requires that the objective and constraint functions are part of the same function, one that returns a structure. The objective function is in the `Fval` field of the structure, and the constraints are in the `Ineq` field. These fields are the output of the `objconstr` function at the end of this example on page 11-88.

Scale Integer Constraints to Lie on Fine Grid

Set the problem to have integer constraints in both variables, $x(1)$ and $x(2)$.

```
intcon = [1 2];
```

Scale the problem so that the variables are scaled by $s = 1/10$, where s multiplies the variables.

```
s = 0.1;
f = @(x)objconstr(x,s);
```

For this scaling to be effective, you need to scale the bounds by $1/s$. Set the unscaled bounds to $-2 \leq x_i \leq 3$ and scale each by $1/s$.

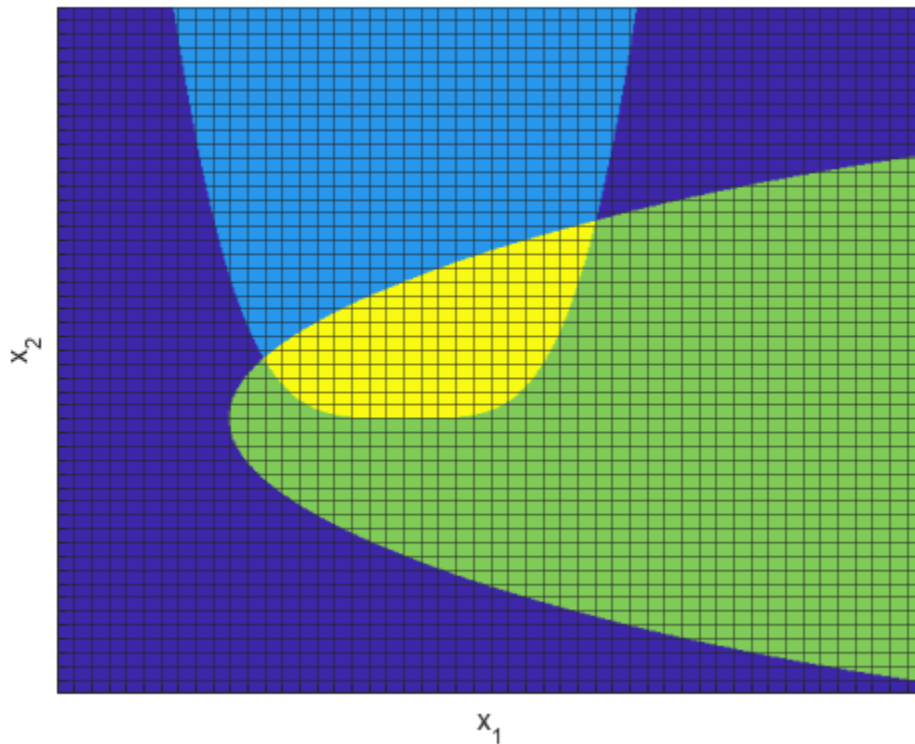
```
lb = [-2, -2]/s;
ub = [3, 3]/s;
```

By using the scaling s , the problem effectively has spacing of s in each component $x(1)$ and $x(2)$. Plot the integer points as a grid with spacing s .

```

hold on
grid on
ax = gca;
sp = -2:s:3;
ax.XTick = sp;
ax.YTick = sp;
ax.Layer = 'top';
ax.GridAlpha = 1/2;
ax.XTickLabel = '';
ax.YTickLabel = '';
xlabel('x_1')
ylabel('x_2')
hold off

```



Solve Scaled Problem

Set options to use tighter constraints than the default, and to use the `surrogateoptplot` plot function.

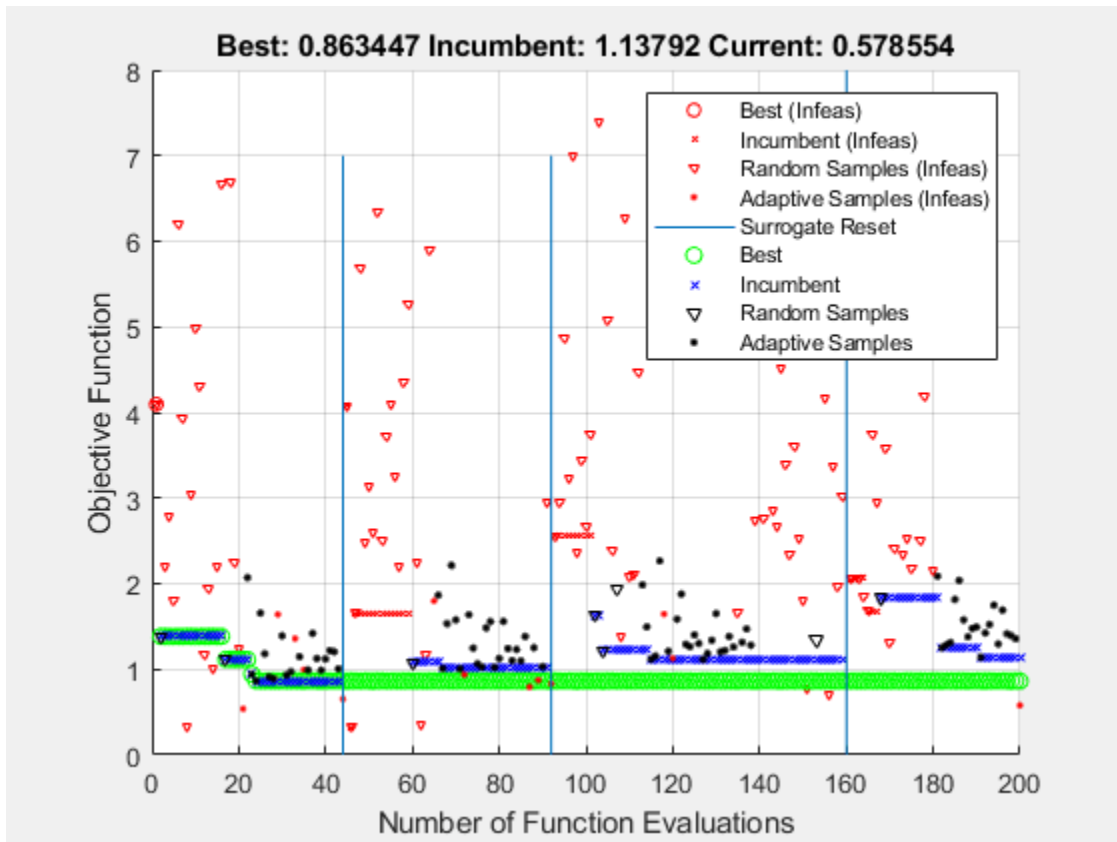
```
opts = optimoptions('surrogateopt', 'PlotFcn', "surrogateoptplot", "ConstraintTolerance", 1e-6);
```

Call `surrogateopt` to solve the problem.

```

rng default % For reproducibility
[sol, fval, eflag, outpt] = surrogateopt(f, lb, ub, intcon, opts)

```



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
sol = 1x2
```

```
    5    1
```

```
fval = 0.8634
```

```
eflag = 0
```

```
outpt = struct with fields:
```

```
    elapsedtime: 79.4430
```

```
    funccount: 200
```

```
    constrviolation: -0.0375
```

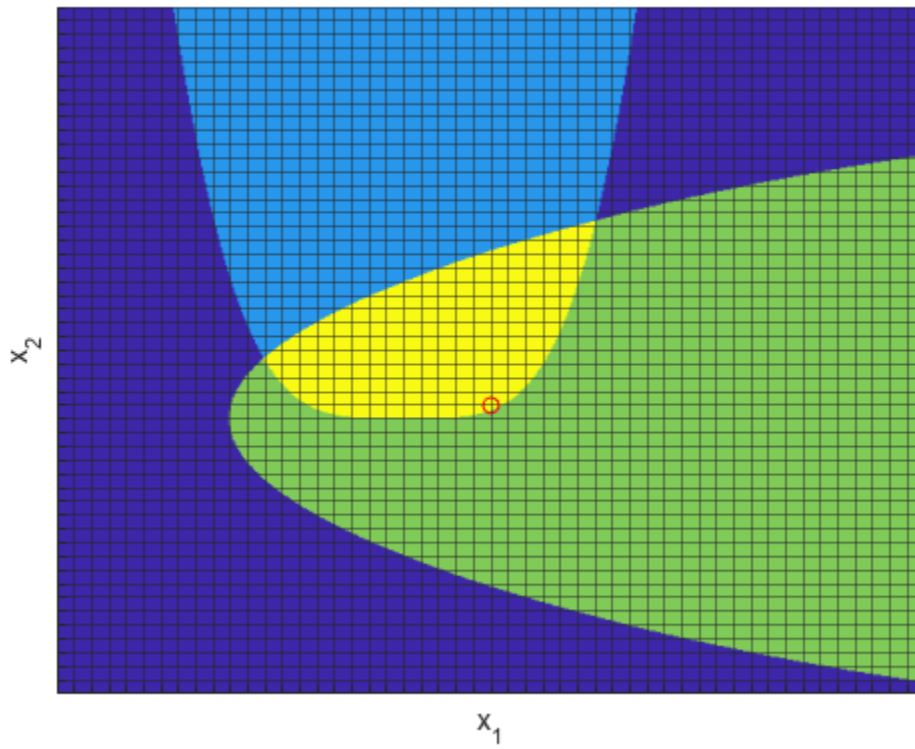
```
        ineq: [-0.0375 -1.4883]
```

```
    rngstate: [1x1 struct]
```

```
    message: 'surrogateopt stopped because it exceeded the function evaluation limit set
```

Plot the solution as a red circle on the figure. Notice that the objective function value is approximately 0.86.

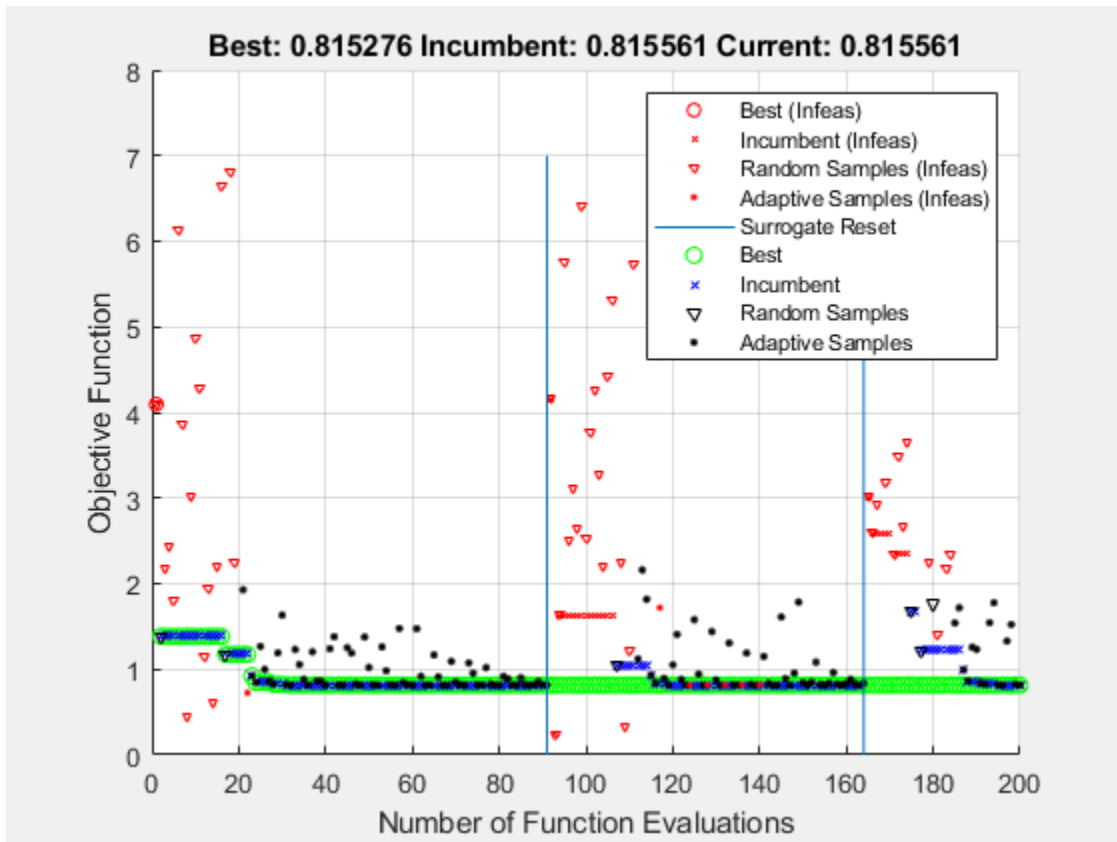
```
figure(fig);
hold on
plot3(sol(1)*s,sol(2)*s,5,'ro')
hold off
```



Compare to Solution Without Integer Constraints

Compare the solution with integer constraints to the solution without integer constraints.

```
[sol2,fval2,eflag2,outpt2] = surrogateopt(f,lb,ub,[],opts)
```



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
sol2 = 1x2
```

```
    4.3631    0.3626
```

```
fval2 = 0.8153
```

```
eflag2 = 0
```

```
outpt2 = struct with fields:
```

```
    elapsedtime: 41.6455
```

```
    funccount: 200
```

```
    constrviolation: -1.7294e-05
```

```
        ineq: [-1.7294e-05 -1.4339]
```

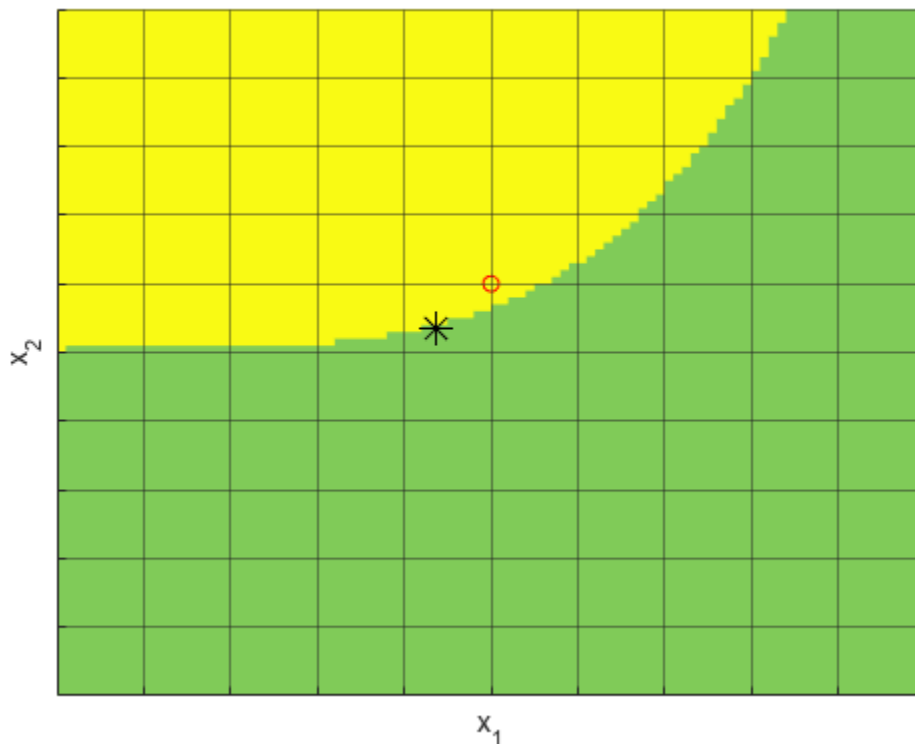
```
    rngstate: [1x1 struct]
```

```
    message: 'surrogateopt stopped because it exceeded the function evaluation limit set
```

Here, the objective function value is approximately 0.815. The integer constraints increase the objective function value by less than 10%. Plot the new solution along with the previous integer solution. Zoom in to see the solution points more clearly.

```
figure(fig)
hold on
plot3(sol2(1)*s,sol2(2)*s,5,'k*','MarkerSize',12)
```

```
xlim([0 1])
ylim([-1/2 1/2])
hold off
```



Helper Function

This code creates the `objconstr` helper function. This function scales the variable x by the factor s , returns the objective function value in the `Fval` field of the `F` structure, and returns the nonlinear constraints in the `Ineq` field of the `F` structure.

```
function F = objconstr(x,s)
x = x*s;
fun = log(1 + 3*(x(2) - (x(1)^3 - x(1)))^2 + (x(1) - 4/3)^2);
c1 = x(1)^4 - 5*sinh(x(2)/5);
c2 = x(2)^2 - 5*tanh(x(1)/5) - 1;
c = [c1 c2];
F.Fval = fun;
F.Ineq = c;
end
```

See Also

`surrogateopt`

More About

- “Mixed Integer ga Optimization” on page 8-40

Improve surrogateopt Solution or Process

surrogateopt Stalls

When you have both linear constraints and integer constraints, `surrogateopt` can fail to find any feasible points or enough distinct feasible points to create a surrogate. In these cases, the solver exits with exit flag -2 (no feasible point found) or 3 (too few feasible points). For details on exit flag -2, see “No Feasible Point Found” on page 11-90.

Exit flag 3 can occur in two different ways:

- There were too few feasible points to construct an initial surrogate.
- There were too few feasible points to construct a surrogate after a surrogate reset.

You can see which case applies by using the `surrogateoptplot` plot function.

```
options = optimoptions('surrogateopt','PlotFcn','surrogateoptplot');  
[sol,fval,exitflag] = surrogateopt(arguments,options);
```

After each surrogate reset, `surrogateopt` requires more feasible points to construct the next surrogate. When there are integer constraints, `surrogateopt` can exhaust the set of feasible points, or can fail to find new feasible points even when some remain

If `surrogateopt` has performed at least one reset, then it has successfully searched for a solution. In this case, you might have the solution to the problem.

If `surrogateopt` was unable to create an initial surrogate, or if `surrogateopt` reset and you want to try to find another solution, perform the following steps.

1 Relax some constraints.

- Change some linear constraints to nonlinear, which causes the solver to not insist on strict feasibility. This can give `surrogateopt` more feasible points to use in constructing surrogates.
- Relax some linear inequality constraints by choosing larger values for the `b` vector. You can relax all `b` values at once by adding a scalar:

```
b = b + 5;
```

2 Similarly, if your bound constraints are causing the problem to have too few feasible points, and if it makes sense for your problem, relax the bounds. Take larger upper bounds or smaller lower bounds or both. You can relax all bounds at once by adding or subtracting a scalar.

```
ub = ub + 3;  
lb = lb - 1;
```

No Feasible Point Found

When `surrogateopt` cannot find a point that is feasible with respect to bounds, integer constraints, and linear constraints, it returns exit flag -2. In this case, the problem is truly infeasible.

However, the solver can also return exit flag -2 when it cannot locate a point that is feasible with respect to nonlinear inequality constraints. This can sometimes occur even when feasible points exist. To proceed, follow the steps in “Converged to an Infeasible Point”.

Solution Might Not Be Optimal

Usually, `surrogateopt` stops when it runs out of function evaluations. This means that `surrogateopt` does not stop because it reaches an optimal solution. However, when a surrogate reset occurs, the current solution is usually near a local optimum.

How can you evaluate the quality of a solution? Generally, this is difficult to do. Here are some steps for investigating a solution to help determine its local quality. However, there is no procedure that guarantees that a point is a global solution. See “Can You Certify That a Solution Is Global?” on page 4-41.

- If the problem has no integer constraints, look at nearby points. To do so, call `patternsearch` on the returned solution. Set the `InitialMeshSize` option to the size of the search step you want to use. To keep `patternsearch` from taking too much time, set the `MaxIterations` option to 1 and the `UseCompletePoll` option to `true`:

```
options = optimoptions('patternsearch',...
    'InitialMeshSize',1e-3,...
    'MaxIterations',1,'UseCompletePoll',true);
```

If your problem has nonlinear constraints, first convert the constraints to the form that `patternsearch` accepts using “Convert Nonlinear Constraints Between surrogateopt Form and Other Solver Forms” on page 11-74.

- If the problem has no integer constraints, try running `fmincon` starting from the solution. Again, if your problem has nonlinear constraints, first convert the constraints to the form that `fmincon` accepts using “Convert Nonlinear Constraints Between surrogateopt Form and Other Solver Forms” on page 11-74. If the problem uses a simulation or ODE solver, you might need to set larger finite difference options for `fmincon`. See “Optimizing a Simulation or Ordinary Differential Equation”.
- If the problem has integer constraints, then there is little to do except to try to run `surrogateopt` for more function evaluations. Do so most efficiently by using a checkpoint file. See “Work with Checkpoint Files” on page 11-56. If you did not use a checkpoint file, you can also give a set of initial points using the `InitialPoints` option.

See Also

`surrogateopt`

More About

- “Surrogate Optimization”

Vectorized Surrogate Optimization for Custom Parallel Simulation

This example shows how to use the `surrogateopt UseVectorized` option to perform custom parallel optimization. You can use this technique when you cannot use the `UseParallel` option successfully. For example, the `UseParallel` option might not apply to a Simulink® simulation that requires `parsim` for parallel evaluation. Optimizing a vectorized parallel simulation involves considerable overhead, so this technique is most useful for time-consuming simulations.

The parallel strategy in this example is to break up the optimization into chunks of size N , where N is the number of parallel workers. The example prepares N sets of parameters in a `Simulink.SimulationInput` vector, and then calls `parsim` on the vector. When all N simulations are complete, `surrogateopt` updates the surrogate and evaluates another N sets of parameters.

Model System

This example attempts to fit the Lorenz dynamical system to uniform circular motion over a short time interval. The Lorenz system and its uniform circular approximation are described in the example “Fit an Ordinary Differential Equation (ODE)”.

The `Lorenz_system.slx` Simulink model implements the Lorenz ODE system. This model is included when you run this example using the live script.

The `fitlorenzfn` helper function at the end of this example on page 11-97 calculates points from uniform circular motion. Set circular motion parameters from the example “Fit an Ordinary Differential Equation (ODE)” that match the Lorenz dynamics reasonably well.

```
x = zeros(8,1);
x(1) = 1.2814;
x(2) = -1.4930;
x(3) = 24.9763;
x(4) = 14.1870;
x(5) = 0.0545;
x(6:8) = [13.8061;1.5475;25.3616];
```

This system does not take much time to simulate, so the parallel time for the optimization is not less than the time to optimize serially. The purpose of this example is to show how to create a vectorized parallel simulation, not to provide a specific example that runs better in parallel.

Objective Function

The objective function is to minimize the sum of squares of the difference between the Lorenz system and the uniform circular motion over a set of times from 0 through 1/10. For times `xdata`, the objective function is

```
objective = sum((fitlorenzfn(x,xdata) - lorenz(xdata)).^2) - (F(1) + F(end))/2
```

Here, `lorenz(xdata)` represents the 3-D evolution of the Lorenz system at times `xdata`, and `F` represents the vector of squared distances between corresponding points in the circular and Lorenz systems. The objective subtracts half of the values at the endpoints to best approximate an integral.

Consider the uniform circular motion as the curve to match, and modify the Lorenz parameters in the simulation to minimize the objective function.

Calculate Lorenz System for Specific Parameters

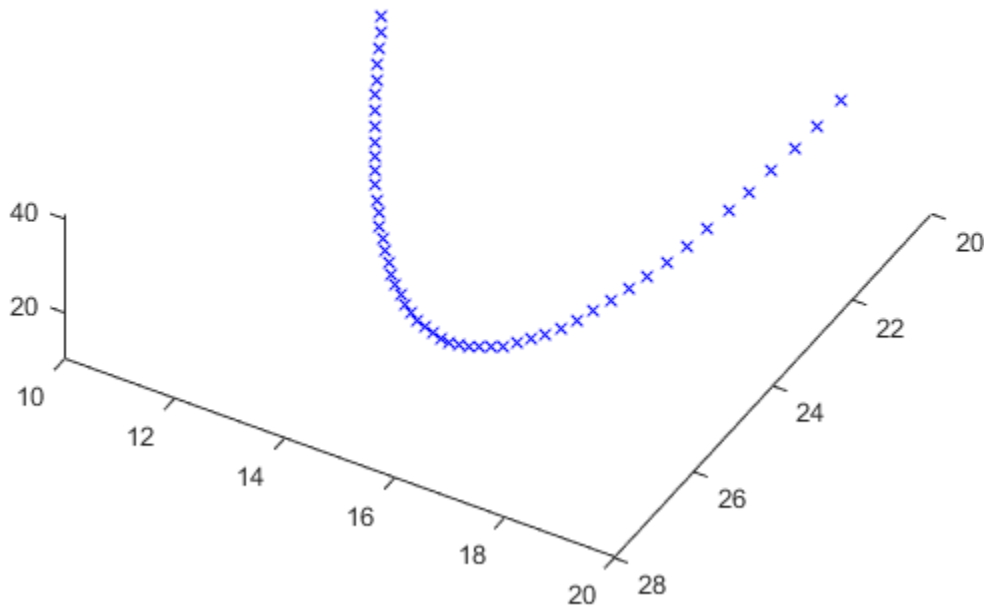
Calculate and plot the Lorenz system for Lorenz's original parameters.

```

model = 'Lorenz_system';
open_system(model);
in = Simulink.SimulationInput(model);
% params [X0,Y0,Z0,Sigma,Beta,Rho]
params = [10,20,10,10, 8/3, 28]; % The original parameters Sigma, Beta, Rho
in = setparams(in,model,params);
out = sim(in);

yout = out.yout;
h = figure;
plot3(yout{1}.Values.Data,yout{2}.Values.Data,yout{3}.Values.Data,'bx');
view([-30 -70])

```



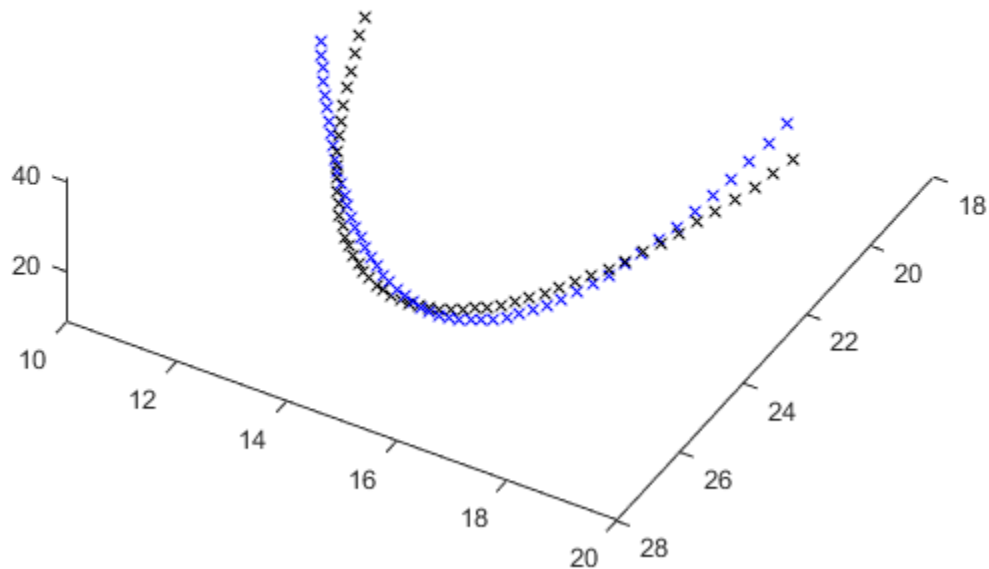
Calculate Uniform Circular Motion

Calculate the uniform circular motion for the x parameters given previously over the time interval in the Lorenz calculation, and plot the result along with the Lorenz plot.

```

tlist = yout{1}.Values.Time;
M = fitlorenzfn(x,tlist);
hold on
plot3(M(:,1),M(:,2),M(:,3),'kx')
hold off

```



The `objfun` helper function at the end of this example on page 11-98 calculates the sum of squares difference between the Lorenz system and the uniform circular motion. The objective is to minimize this sum of squares.

```
ssq = objfun(in,params,M,model)
```

```
ssq = 26.9975
```

Fit Lorenz System in Parallel

To optimize the fit, use `surrogateopt` to modify the parameters of the Simulink model. The `parobjfun` helper function at the end of this example on page 11-98 accepts a matrix of parameters, where each row of the matrix represents one set of parameters. The function calls the `setparams` helper function at the end of this example on page 11-98 to set parameters for a Simulink `SimulationInput` vector. The `parobjfun` function then calls `parsim` to evaluate the model on the parameters in parallel.

Open a parallel pool and specify `N` as the number of workers in the pool.

```
pool = gcp('nocreate'); % Check whether a parallel pool exists
if isempty(pool) % If not, create one
    pool = parpool;
end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

```
N = pool.NumWorkers
```

N = 6

Set the `BatchUpdateInterval` option to N and set the `UseVectorized` option to `true`. These settings cause `surrogateopt` to pass N points at a time to the objective function. Set the initial point to the parameters used earlier, because they give a reasonably good fit to the uniform circular motion. Set the `MaxFunctionEvaluations` option to 600, which is an integer multiple of the 6 workers on the computer used for this example.

```
options = optimoptions('surrogateopt','BatchUpdateInterval',N,...
    'UseVectorized',true,'MaxFunctionEvaluations',600,...
    'InitialPoints',params);
```

Set bounds of 20% above and below the current parameters.

```
lb = 0.8*params;
ub = 1.2*params;
```

For added speed, set the simulation to use fast restart.

```
set_param(model,'FastRestart','on');
```

Create a vector of N simulation inputs for the objective function.

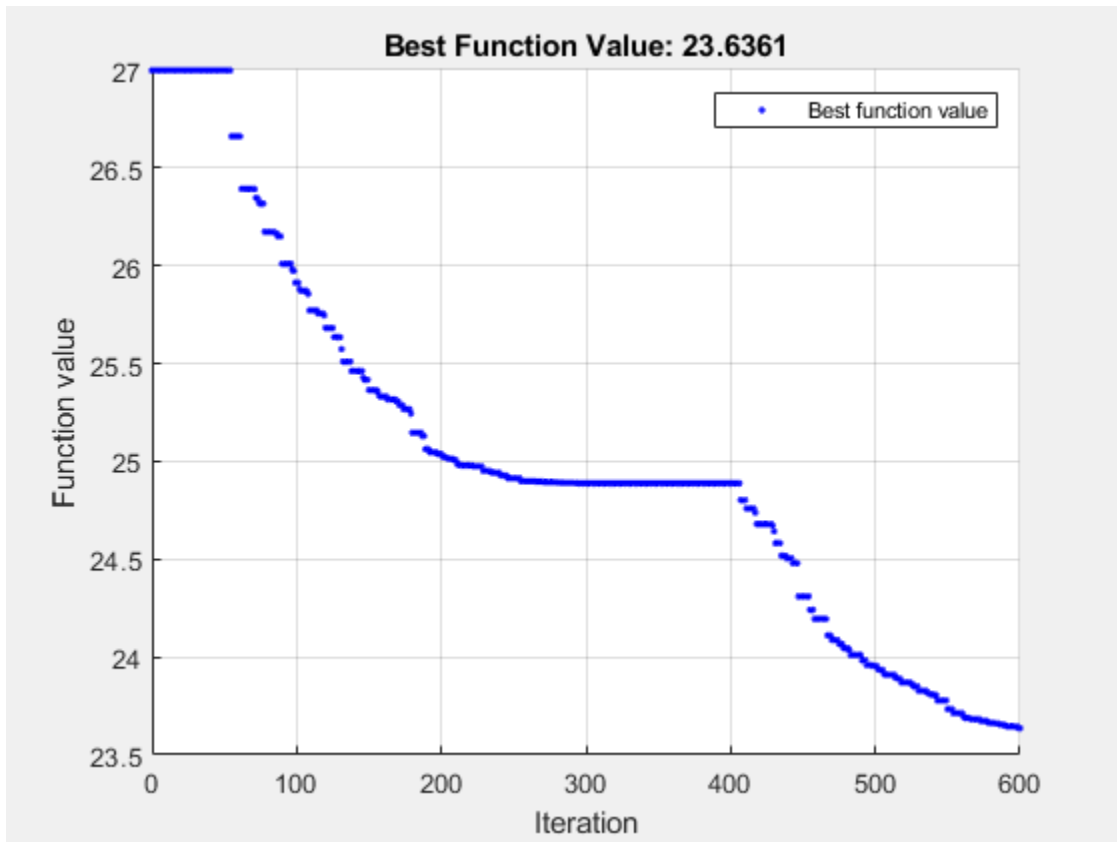
```
simIn(1:N) = Simulink.SimulationInput(model);
```

For reproducibility, set the random stream.

```
rng(100)
```

Optimize the objective in a vectorized parallel manner by calling `parobjfun`.

```
tic
[fittedparams,fval] = surrogateopt(@(params)parobjfun(simIn,params,M,model),lb,ub,options)
```



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
fittedparams = 1×6
```

```
10.5627 19.8962 9.8420 8.9616 2.5723 27.9687
```

```
fval = 23.6361
```

```
paralleltime = toc
```

```
paralleltime = 457.9271
```

The objective function value improves (decreases). Display the original and improved values.

```
disp([ssq,fval])
```

```
26.9975 23.6361
```

Plot the fitted points.

```
figure(h)
```

```
hold on
```

```
in = setparams(in,model,fittedparams);
```

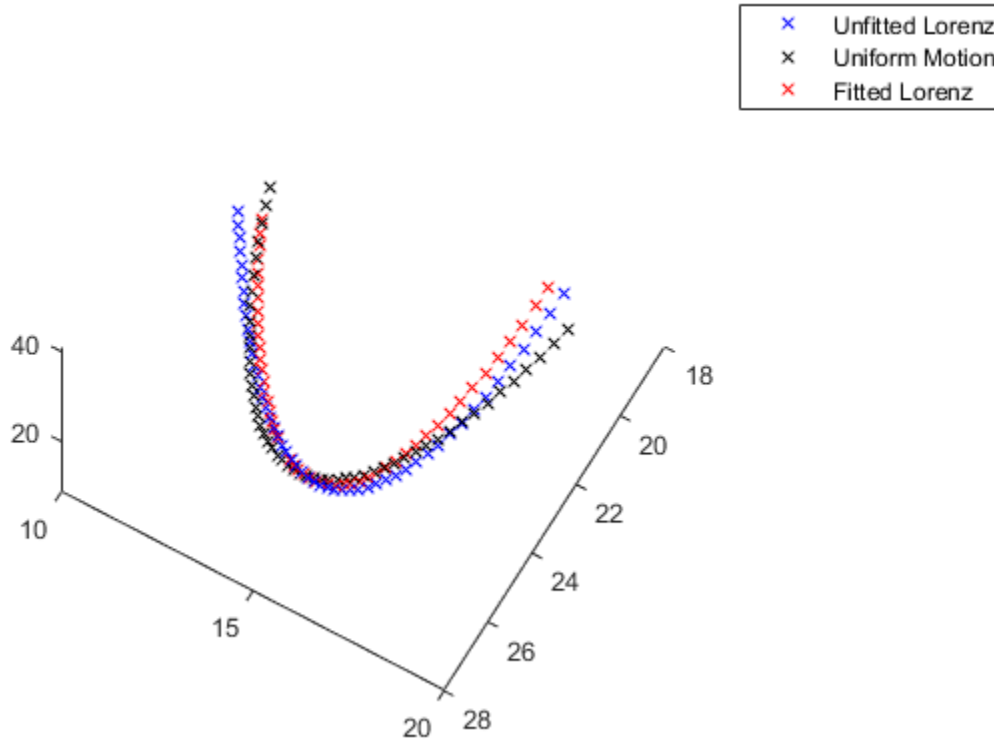
```
out = sim(in);
```

```
yout = out.yout;
```

```
plot3(yout{1}.Values.Data,yout{2}.Values.Data,yout{3}.Values.Data,'rx');
```



```
legend('Unfitted Lorenz','Uniform Motion','Fitted Lorenz')
hold off
```



To close the model, you must first disable fast restart.

```
set_param(model,'FastRestart','off');
close_system(model)
```

Conclusion

When you cannot use the `UseParallel` option successfully, you can optimize a simulation in parallel by setting the surrogateopt `UseVectorized` option to `true` and the `BatchUpdateInterval` option to a multiple of the number of parallel workers. This process speeds up the parallel optimization, but involves overhead, so is best suited for time-consuming simulations.

Helper Functions

The following code creates the `fitlorenzfn` helper function.

```
function f = fitlorenzfn(x,xdata)
theta = x(1:2);
R = x(3);
V = x(4);
t0 = x(5);
delta = x(6:8);
f = zeros(length(xdata),3);
```

```
f(:,3) = R*sin(theta(1))*sin(V*(xdata - t0)) + delta(3);
f(:,1) = R*cos(V*(xdata - t0))*cos(theta(2)) ...
- R*sin(V*(xdata - t0))*cos(theta(1))*sin(theta(2)) + delta(1);
f(:,2) = R*sin(V*(xdata - t0))*cos(theta(1))*cos(theta(2)) ...
- R*cos(V*(xdata - t0))*sin(theta(2)) + delta(2);
end
```

The following code creates the objfun helper function.

```
function f = objfun(in,params,M,model)
in = setparams(in,model,params);
out = sim(in);
yout = out.yout;
vals = [yout{1}.Values.Data,yout{2}.Values.Data,yout{3}.Values.Data];
f = sum((M - vals).^2,2);
f = sum(f) - (f(1) + f(end))/2;
end
```

The following code creates the parobjfun helper function.

```
function f = parobjfun(simIn,params,M,model)
N = size(params,1);
f = zeros(N,1);
for i = 1:N
    simIn(i) = setparams(simIn(i),model,params(i,:));
end
simOut = parsim(simIn,'ShowProgress','off'); % Suppress output
for i = 1:N
    yout = simOut(i).yout;
    vals = [yout{1}.Values.Data,yout{2}.Values.Data,yout{3}.Values.Data];
    g = sum((M - vals).^2,2);
    f(i) = sum(g) - (g(1) + g(end))/2;
end
end
```

The following code creates the setparams helper function.

```
function pp = setparams(in,model,params)
% parameters [X0,Y0,Z0,Sigma,Beta,Rho]
pp = in.setVariable('X0',params(1),'Workspace',model);
pp = pp.setVariable('Y0',params(2),'Workspace',model);
pp = pp.setVariable('Z0',params(3),'Workspace',model);
pp = pp.setVariable('Sigma',params(4),'Workspace',model);
pp = pp.setVariable('Beta',params(5),'Workspace',model);
pp = pp.setVariable('Rho',params(6),'Workspace',model);
end
```

See Also

surrogateopt | parsim

Related Examples

- “Surrogate Optimization”

Problem-Based Surrogate Optimization

- “Optimize Multidimensional Function Using surrogateopt, Problem-Based” on page 12-2
- “Solve Feasibility Problem Using surrogateopt, Problem-Based” on page 12-6
- “Feasibility Using Problem-Based Optimize Live Editor Task” on page 12-11
- “Mixed-Integer Surrogate Optimization, Problem-Based” on page 12-20
- “Specify Starting Points and Values for surrogateopt, Problem-Based” on page 12-24

Optimize Multidimensional Function Using surrogateopt, Problem-Based

This example shows how to minimize a multidimensional function using surrogate optimization in the problem-based approach. The function to minimize, `multirosenbrock(x)`, appears at the end of this example on page 12-4. The `multirosenbrock` function has a single local minimum of 0 at the point $[1, 1, \dots, 1]$. The function is designed to be challenging for solvers to minimize.

Note: The code for the `multirosenbrock` helper function is provided at the end of this example on page 12-4. Make sure the code is included at the end of your script or in a file on the path.

Create a 4-D optimization variable `x`. The `multirosenbrock` function expects the variable to be a row vector, so specify `x` as a 4-element row vector.

```
x = optimvar("x",1,4);
```

The `surrogateopt` solver requires finite bounds on all problem variables. Specify lower bounds of -3 and upper bounds of 3. When you specify scalar bounds, they apply to all problem variables.

```
x.LowerBound = -3;  
x.UpperBound = 3;
```

To use `multirosenbrock` as the objective function, convert the function to an optimization expression using `fcn2optimexpr`.

```
fun = fcn2optimexpr(@multirosenbrock,x);
```

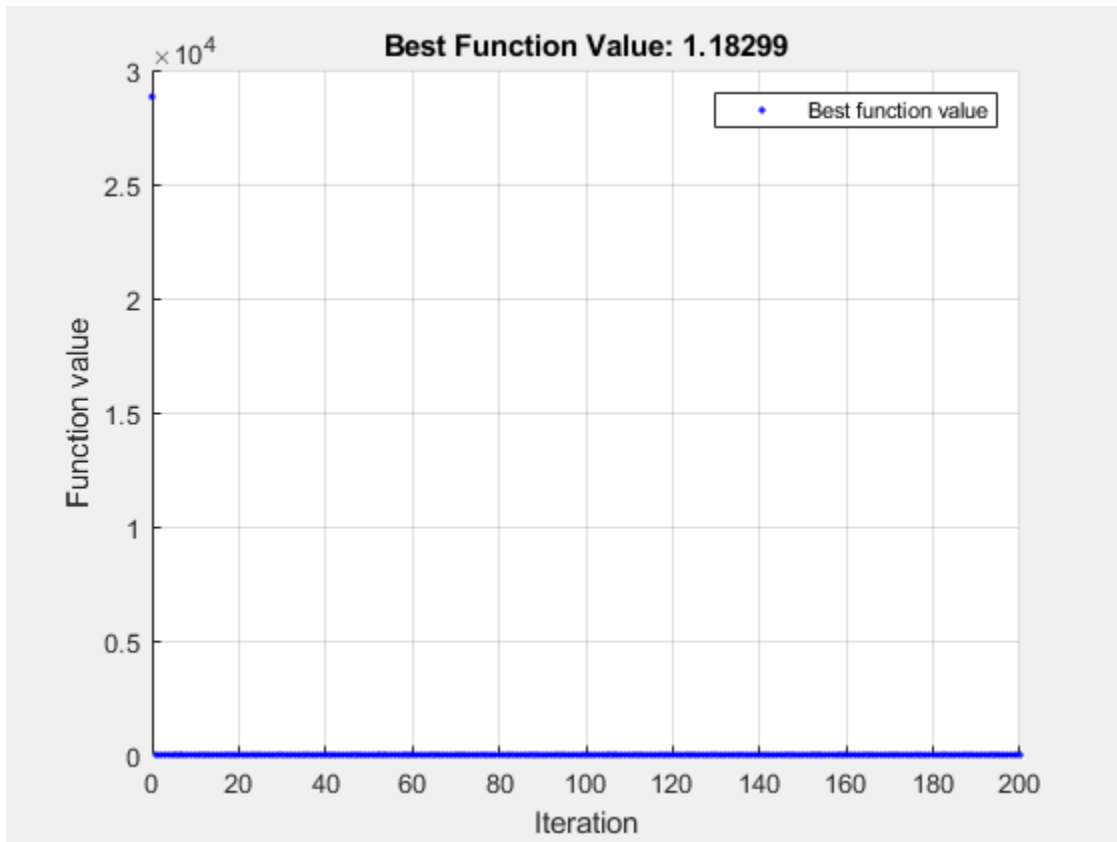
Create an optimization problem with the objective function `multirosenbrock`.

```
prob = optimproblem("Objective",fun);
```

Solve the problem, specifying the `surrogateopt` solver.

```
rng default % For reproducibility  
[sol,fval] = solve(prob,"Solver","surrogateopt")
```

```
Solving problem using surrogateopt.
```



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
sol = struct with fields:
    x: [0.1017 0.0169 0.3921 0.1585]
```

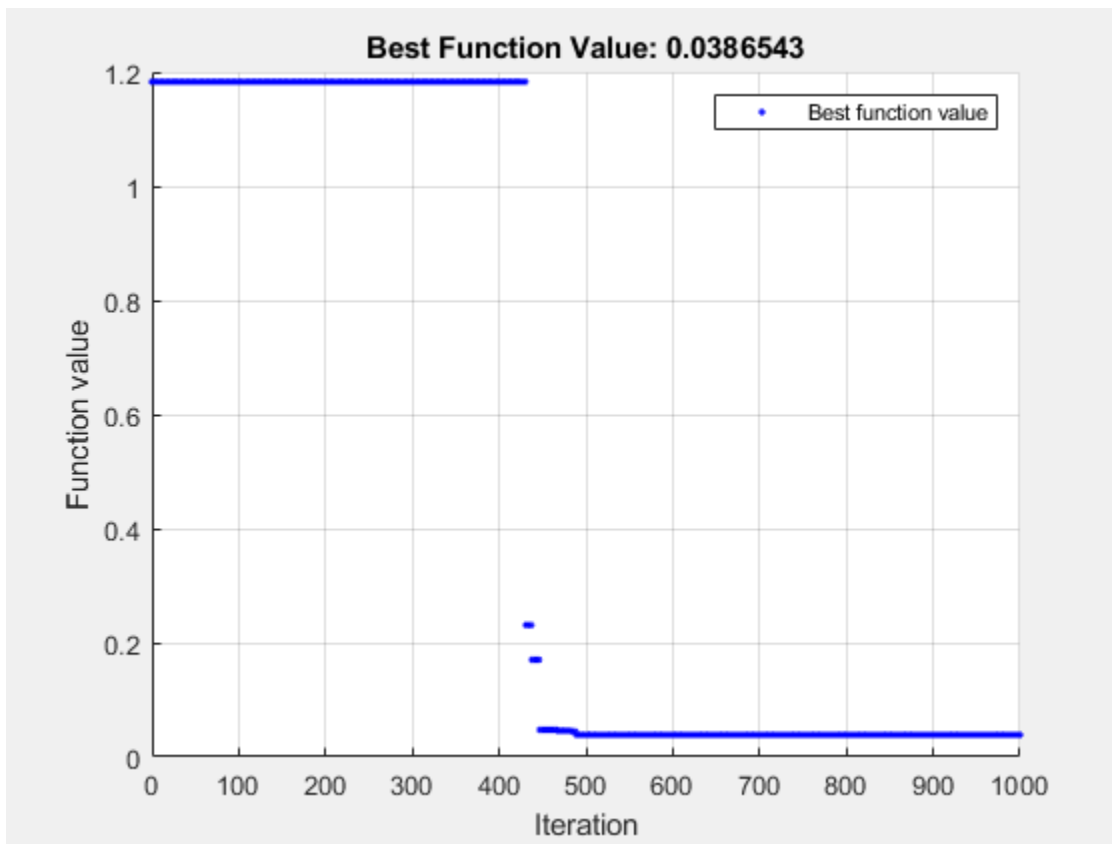
```
fval = 1.1830
```

Attempt to Improve Solution

The returned solution is not good, because the objective function value is not very close to 0. Try to improve the solution by running surrogateopt for more evaluations. Use the previous solution as a start point.

```
options = optimoptions("surrogateopt","MaxFunctionEvaluations",1000);
[sol2,fval2] = solve(prob,sol,"Solver","surrogateopt","Options",options)
```

Solving problem using surrogateopt.



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
sol2 = struct with fields:
    x: [0.9212 0.8487 0.8199 0.6725]
```

```
fval2 = 0.0387
```

This time, the solver reaches a good solution.

Helper Function

This code creates the `multirosenbrock` helper function.

```
function F = multirosenbrock(x)
% This function is a multidimensional generalization of Rosenbrock's
% function. It operates in a vectorized manner, assuming that x is a matrix
% whose rows are the individuals.

% Copyright 2014 by The MathWorks, Inc.

N = size(x,2); % assumes x is a row vector or 2-D matrix
if mod(N,2) % if N is odd
    error('Input rows must have an even number of elements')
end
```

```
odds = 1:2:N-1;  
evens = 2:2:N;  
F = zeros(size(x));  
F(:,odds) = 1-x(:,odds);  
F(:,evens) = 10*(x(:,evens)-x(:,odds).^2);  
F = sum(F.^2,2);  
end
```

See Also

surrogateopt | fcn2optimexpr | solve

Related Examples

- “Surrogate Optimization”

Solve Feasibility Problem Using surrogateopt, Problem-Based

Some problems require you to find a point that satisfies all constraints, with no objective function to minimize. For example, suppose that you have the following constraints:

$$\begin{aligned}(y + x^2)^2 + 0.1y^2 &\leq 1 \\ y &\leq \exp(-x) - 3 \\ y &\leq x - 4.\end{aligned}$$

Do any points (x, y) satisfy the constraints? To answer this question, you need to evaluate the expressions at a variety of points. The `surrogateopt` solver does not require you to provide initial points, and it searches a wide set of points. So, `surrogateopt` works well for feasibility problems.

To visualize the constraints, see [Visualize Constraints](#) on page 12-8. For a solver-based approach to this problem, see [“Solve Feasibility Problem”](#) on page 11-78.

Note: This example uses two helper functions, `outfun` and `evaluateExpr`. The code for each function is provided at the end of this example on page 12-9. Make sure the code for each function is included at the end of your script or in a file on the path.

Set Up Feasibility Problem

For the problem-based approach, create optimization variables x and y , and create expressions for the listed constraints. To use the `surrogateopt` solver, you must set finite bounds for all variables. Set lower bounds of -10 and upper bounds of 10 .

```
x = optimvar("x", "LowerBound", -10, "UpperBound", 10);
y = optimvar("y", "LowerBound", -10, "UpperBound", 10);
cons1 = (y + x^2)^2 + 0.1*y^2 <= 1;
cons2 = y <= exp(-x) - 3;
cons3 = y <= x - 4;
```

Create an optimization problem and include the constraints in the problem.

```
prob = optimproblem("Constraints", [cons1 cons2 cons3]);
```

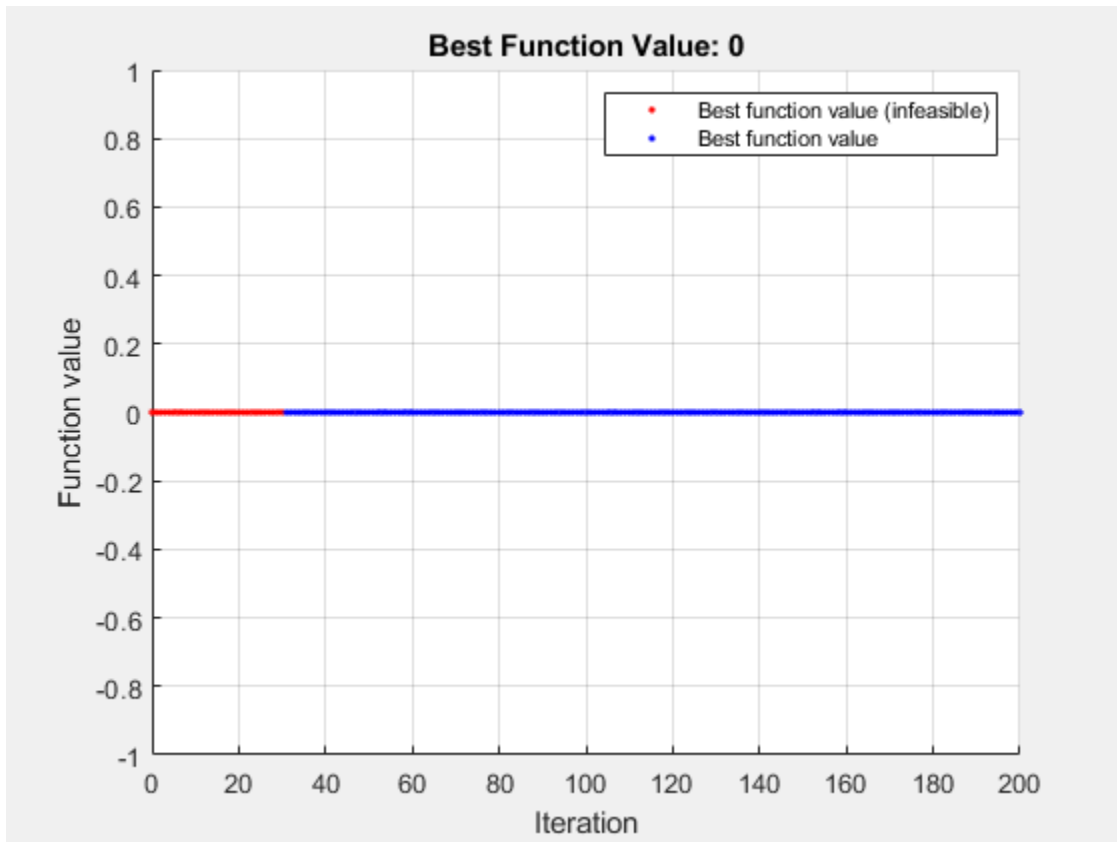
The problem has no objective function. Internally, the solver sets the objective function value to 0 for every point.

Solve Problem

Solve the problem using `surrogateopt`.

```
rng(1) % For reproducibility
[sol, fval] = solve(prob, "Solver", "surrogateopt")
```

Solving problem using `surrogateopt`.



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
sol = struct with fields:
  x: 1.7087
  y: -2.8453
```

```
fval = 0
```

The first several evaluated points are infeasible, as indicated by the color red in the plot. After about 60 evaluations, the solver finds a feasible point, plotted in blue.

Check the feasibility at the returned solution.

```
infeasibility(cons1,sol)
```

```
ans = 0
```

```
infeasibility(cons2,sol)
```

```
ans = 0
```

```
infeasibility(cons3,sol)
```

```
ans = 0
```

All infeasibilities are zero, indicating that the point `sol` is feasible.

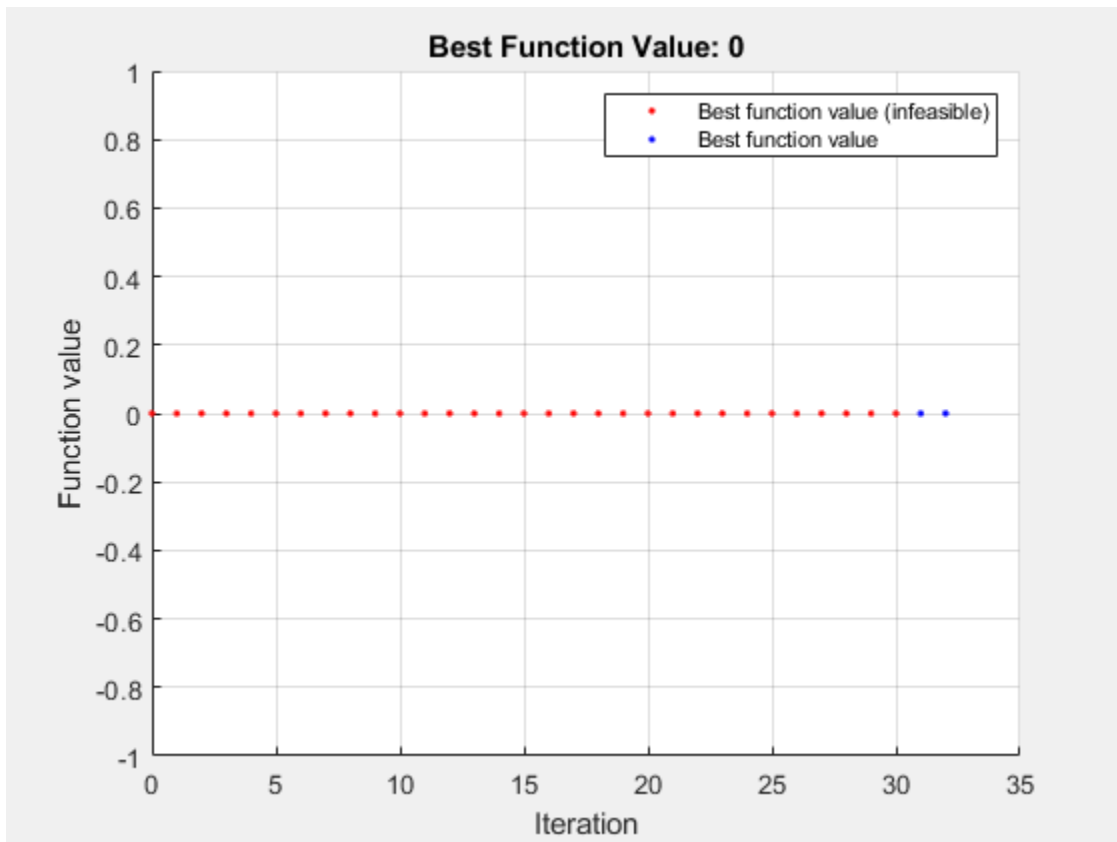
Stop Solver at First Feasible Point

To reach a solution faster, create an output function (see “Output Function” on page 17-53) that stops the solver whenever it reaches a feasible point. The `outfun` helper function at the end of this example on page 12-10 stops the solver when it reaches a point with no constraint violation.

Solve the problem using the `outfun` output function.

```
opts = optimoptions("surrogateopt", "OutputFcn", @outfun);
rng(1) % For reproducibility
[sol, fval] = solve(prob, "Solver", "surrogateopt", "Options", opts)
```

Solving problem using `surrogateopt`.



Optimization stopped by a plot function or output function.

```
sol = struct with fields:
  x: 1.7087
  y: -2.8453
```

```
fval = 0
```

This time, the solver stops earlier than before.

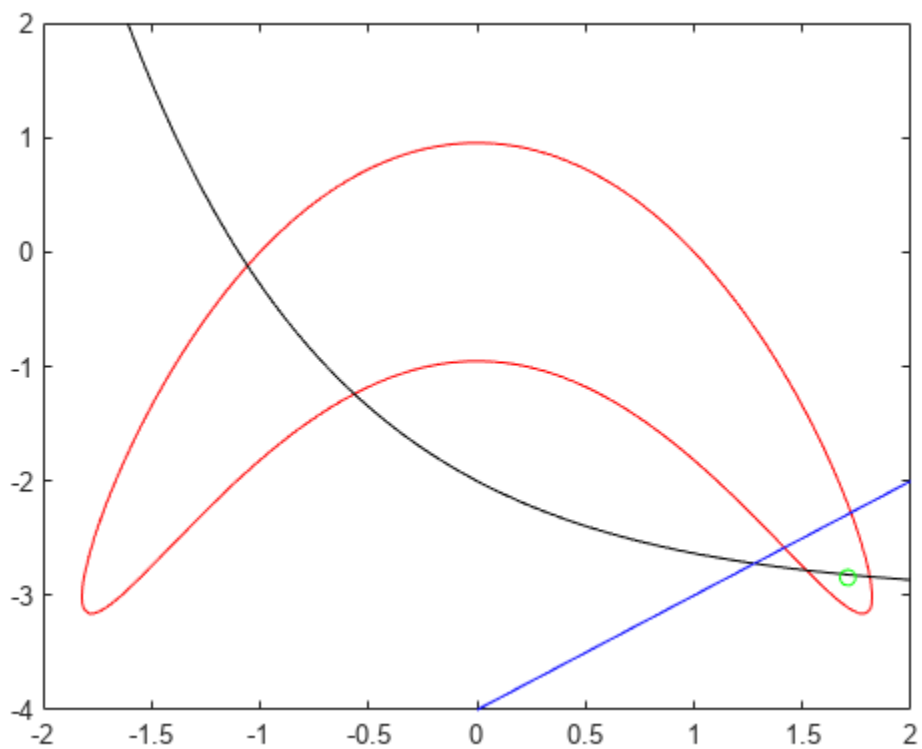
Visualize Constraints

To visualize the constraints, plot the points where each constraint function is zero by using `fimplicit`. The `fimplicit` function passes numeric values to its functions, whereas the `evaluate`

function requires a structure. To tie these functions together, use the `evaluateExpr` helper function, which appears at the end of this example on page 12-10. This function simply puts passed values into a structure with the appropriate names.

Avoid a warning that occurs because the `evaluateExpr` function does not work on vectorized inputs. Plot the returned solution point as a green circle.

```
s = warning('off','MATLAB:fplot:NotVectorized');
figure
cc1 = (y + x^2)^2 + 0.1*y^2 - 1;
fimplicit(@(a,b)evaluateExpr(cc1,a,b),[-2 2 -4 2], 'r')
hold on
cc2 = y - exp(-x) + 3;
fimplicit(@(a,b)evaluateExpr(cc2,a,b),[-2 2 -4 2], 'k')
cc3 = y - x + 4;
fimplicit(@(x,y)evaluateExpr(cc3,x,y),[-2 2 -4 2], 'b')
plot(sol.x,sol.y,'og')
hold off
```



```
warning(s);
```

The feasible region is inside the red outline and below the black and blue lines. The feasible region is at the lower right of the red outline.

Helper Functions

This code creates the `out fun` helper function.

```
function stop = outfun(~,optimValues,state)
stop = false;
switch state
    case 'iter'
        if optimValues.currentConstrviolation <= 0
            stop = true;
        end
    end
end
end
```

This code creates the evaluateExpr helper function.

```
function p = evaluateExpr(expr,x,y)
pt.x = x;
pt.y = y;
p = evaluate(expr,pt);
end
```

See Also

[solve](#) | [infeasibility](#) | [surrogateopt](#)

Related Examples

- “Feasibility Using Problem-Based Optimize Live Editor Task” on page 12-11
- “Solve Feasibility Problem” on page 11-78

Feasibility Using Problem-Based Optimize Live Editor Task

Problem Description

This example shows how to find a feasible point using the Optimize Live Editor task using a variety of solvers. The problem is to find a point $[x, y]$ satisfying these constraints:

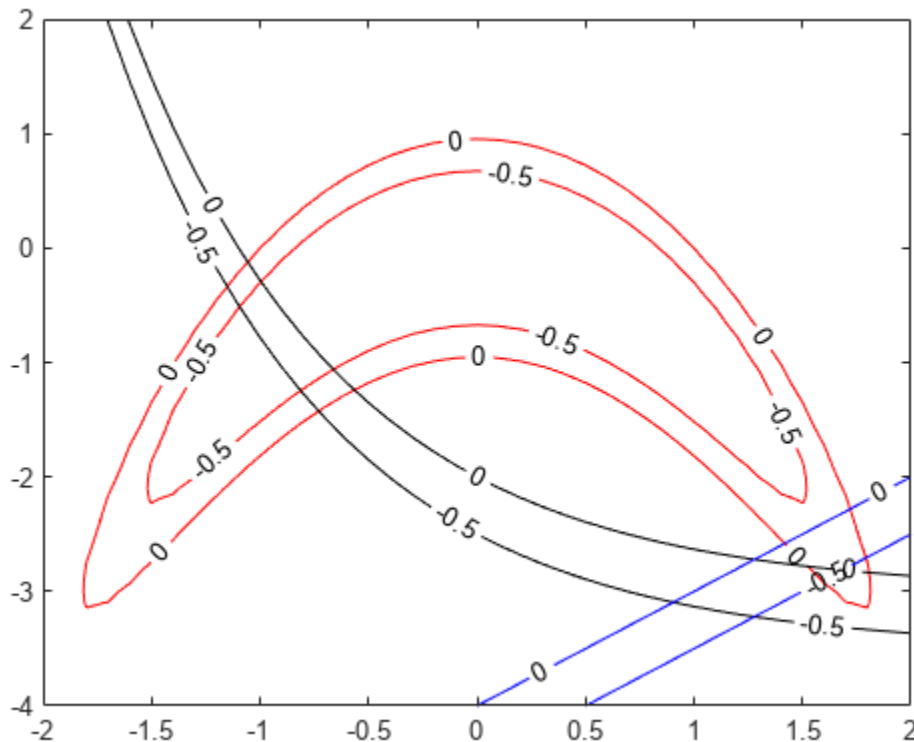
$$(y + x^2)^2 + 0.1y^2 \leq 1$$

$$y \leq \exp(-x) - 3$$

$$y \leq x - 4.$$

Graph the curves where the constraint functions are equal to zero. To see which part of the region is feasible (negative constraint function value), plot the curves where the constraint functions equal $-1/2$. Use the `plotobjconstr` function appearing at the end of this script on page 12-18.

`plotobjconstr`



There appears to be a small feasible region near $x = 1.75, y = -3$. Notice that there is no point where all constraint values are below $-1/2$, so the feasible set is small.

Use Problem-Based Optimize Live Editor Task

To find a feasible point, launch the Optimize Live Editor task from a Live Script by choosing **Task > Optimize** on the **Code** tab or **Insert** tab. Choose the problem-based task.

Set the problem variable x to have lower bound -5 and upper bound 5 . Set the problem variable y to have lower bound -10 and upper bound 10 . Set the initial point for x to 2 and for y to -2 .

Set the **Goal** to **Feasibility**.

Create inequalities representing the three constraints. Your task should match this picture.

Optimize

problem, solution = Find feasible solution to problem

▼ Create optimization variables

Name	Dimensions	Type	Lower bound	Upper bound	Initial point
x	1x1	Continuous	-5	5	2
y	1x1	Continuous	-10	10	-2

▼ Define problem

Goal: Minimize Maximize Feasibility Solve equations

Constraints:

Define on one line	Constraint	Operator	Value
Define on one line	$(y + x^2)^2 + 0.1y^2$	\leq	1
Define on one line	y	\leq	$\exp(-x) - 3$
Define on one line	y	\leq	$x - 4$

Switch the task mode to **Solve problem**. The task chooses the `fmincon` solver, and reaches the following solution.

Solving problem using `fmincon`.

Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in feasible directions, to within the value of the optimality tolerance, and constraints are satisfied to within the value of the constraint tolerance.

<stopping criteria details>

solution = struct with fields:

x: 1.7440
y: -2.9860

reasonSolverStopped =
OptimalSolution

objectiveValue = 0

Effect of Initial Point

Starting from a different initial point can cause `fmincon` to fail to find a solution. Set the initial point for `x` to `-2`.

Name	Dimensions	Type	Lower bound	Upper bound	Initial point
x	1x1 ▼	Continuous ▼	-5 ▼	5 ▼	-2 ▼
y	1x1 ▼	Continuous ▼	-10 ▼	10 ▼	-2 ▼

This time `fmincon` fails to find a feasible solution.

```
Solving problem using fmincon.
```

```
Converged to an infeasible point.
```

```
fmincon stopped because the size of the current step is less than
the value of the step size tolerance but constraints are not
satisfied to within the value of the constraint tolerance.
```

```
<stopping criteria details>
```

```
Consider enabling the interior point method feasibility mode.
```

```
solution = struct with fields:
```

```
  x: -2.1266
```

```
  y: -4.6657
```

```
reasonSolverStopped =
```

```
  NoFeasiblePointFound
```

```
objectiveValue = 0
```

Try Different Solver

To attempt to find a solution, try a different solver. Set the solver to `ga`. To do so, specify the solver in the **Specify problem-dependent solver options** expander. And to monitor the solver progress, set the plot function to **Max constraint violation**.

Specify problem-dependent solver options

Solver ?

Options

▼ Display progress

Text display

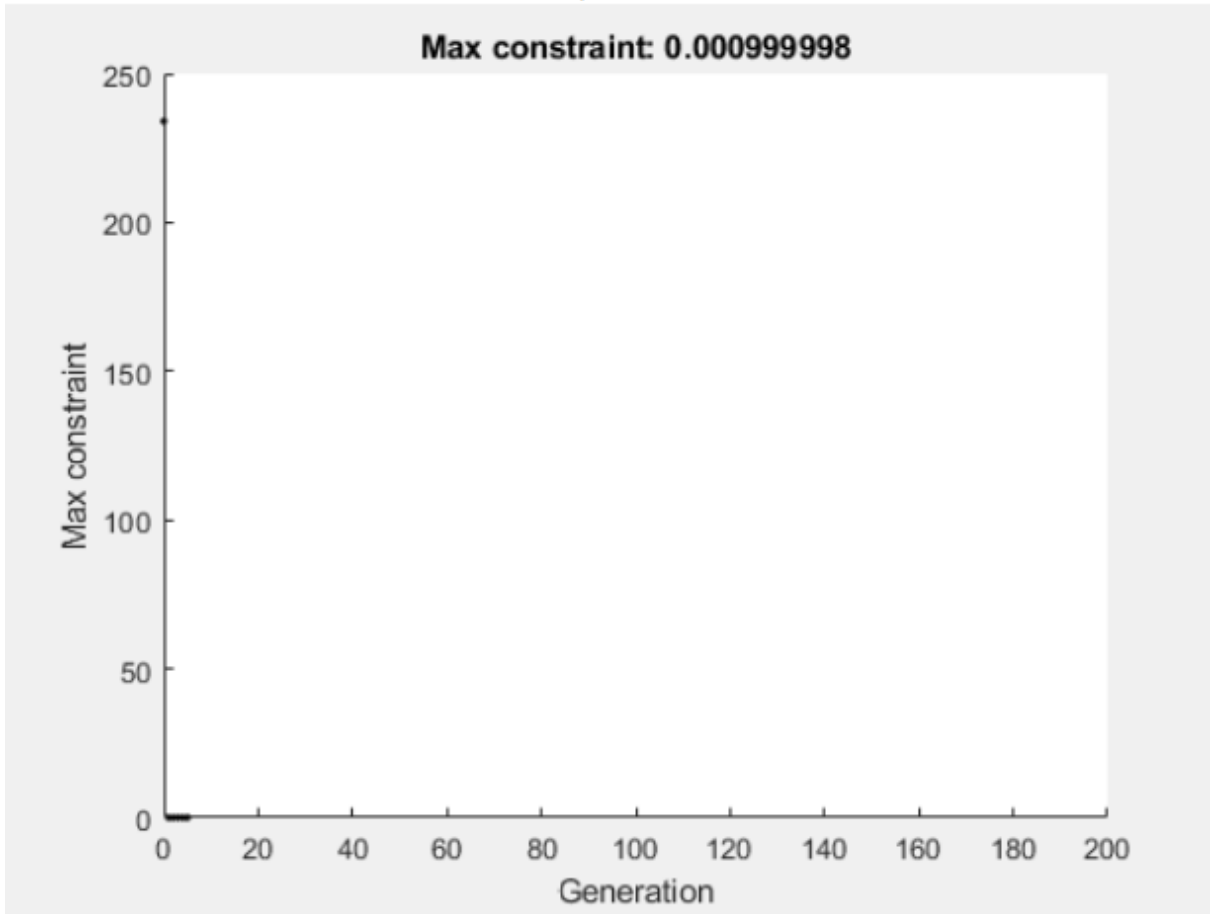
Plot

<input type="checkbox"/> Distance	<input type="checkbox"/> Genealogy	<input type="checkbox"/> Selection	<input type="checkbox"/> Score diversity
<input type="checkbox"/> Scores	<input type="checkbox"/> Stopping criteria	<input checked="" type="checkbox"/> Max constraint violation	<input type="checkbox"/> Best fitness
<input type="checkbox"/> Best individual	<input type="checkbox"/> Expectation value	<input type="checkbox"/> Range	

`ga` finds a feasible point to within the constraint tolerance.

Solving problem using ga.

Optimization terminated: average change in the fitness value less than options.FunctionTolerance and constraint violation is less than options.ConstraintTolerance.



`solution = struct with fields:`

```
x: 1.6718
y: -2.8111
```

`reasonSolverStopped =`

```
SolverConvergedSuccessfully
```

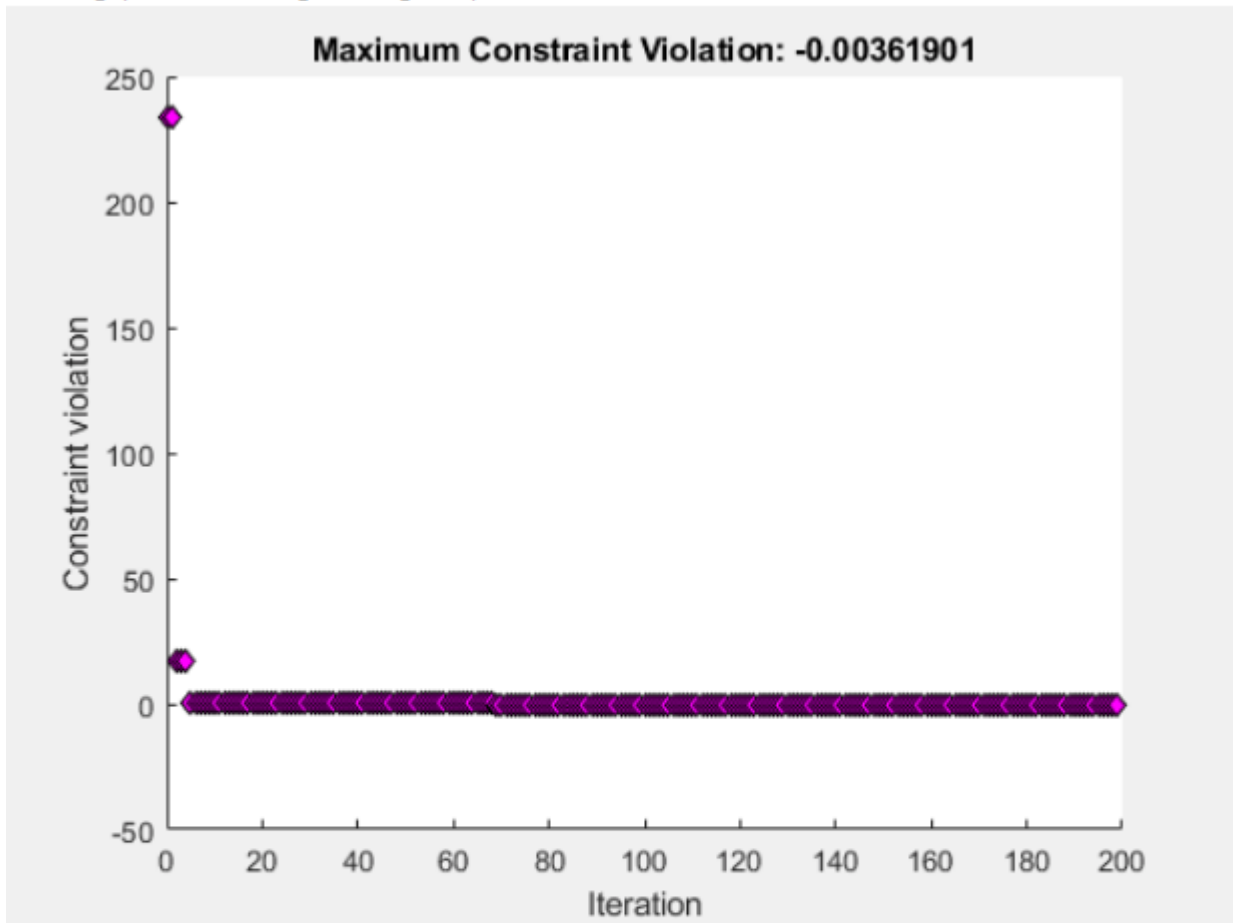
`objectiveValue = 0`

ga finds a different solution than fmincon. The solution is slightly infeasible. To get a solution with lower infeasibility, you can set the constraint tolerance option to a lower value than the default. Alternatively, try a different solver.

Try surrogateopt

Try using the surrogateopt solver. Set the plot function as **Max constraint violation**; this setting does not carry forward automatically from the ga solution.

Solving problem using `surrogateopt`.



`surrogateopt` stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
solution = struct with fields:
```

```
  x: 1.5778
  y: -2.7972
```

```
reasonSolverStopped =
  SolverLimitExceeded
```

```
objectiveValue = 0
```

`surrogateopt` reaches a feasible solution, but does not stop when it first reaches a solution. Instead, `surrogateopt` continues to iterate until it reaches its function evaluation limit. To stop the iterations earlier, specify an output function that halts the solver as soon as the maximum constraint violation reaches $1e-6$ or less. Doing so causes the solver to stop much earlier. Use the `surrou` helper function, which appears at the end of this script on page 12-19. To specify this function, create a function handle to the function.

```
outfun = @surrou;
```

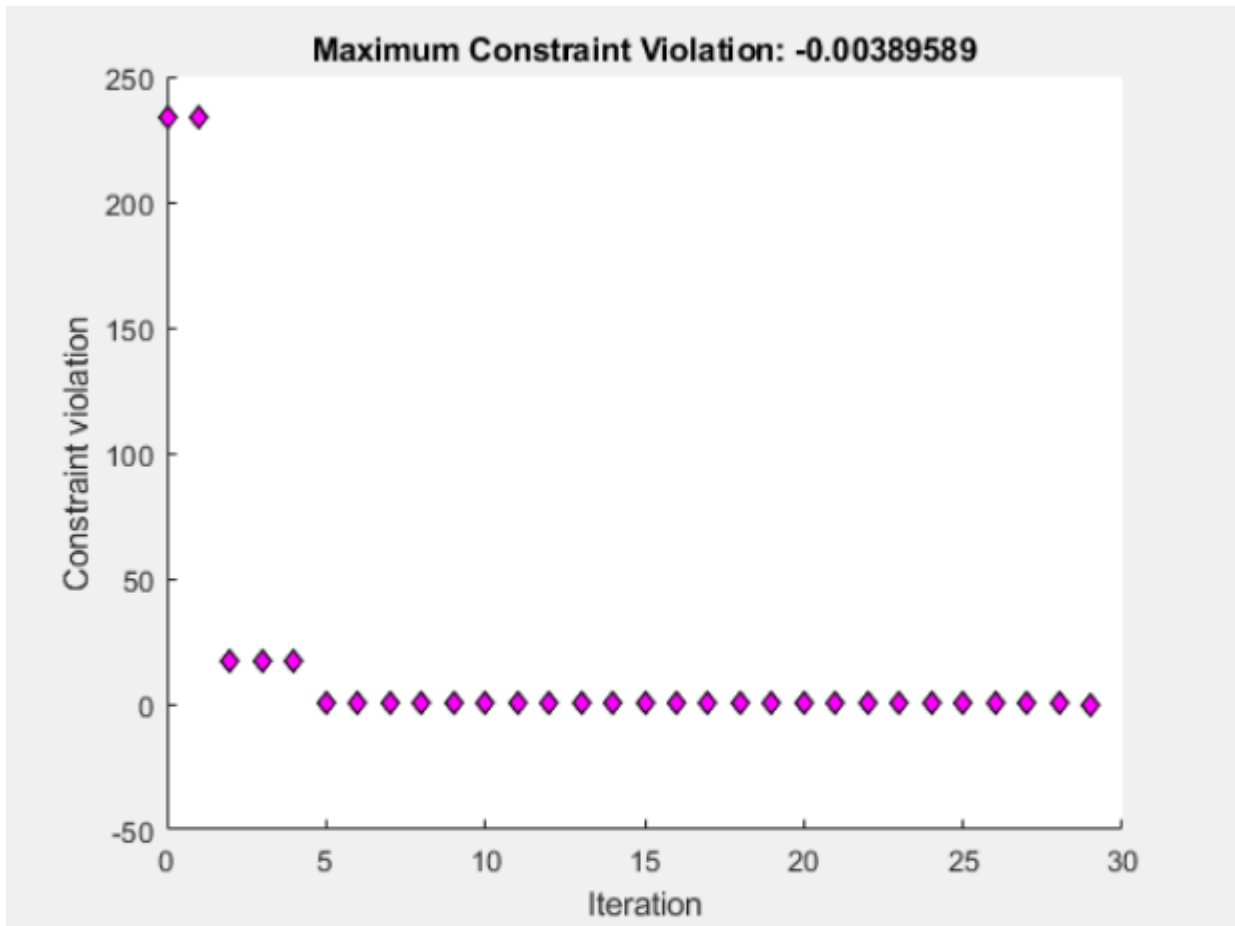
Specify this function handle in the **Specify problem-dependent solver options > Options > Diagnostics > Output function** drop-down menu.

Specify problem-dependent solver options

Solver ?

Options

Solving problem using surrogateopt.



Optimization stopped by a plot function or output function.

`solution = struct with fields:`

`x: 1.5248`

`y: -2.7862`

`reasonSolverStopped =`

`OutputFcnStop`

`objectiveValue = 0`

This time the solver stops after about 30 function evaluations instead of 200. The solution is slightly different than the previous one, but both are feasible solutions.

Conclusions

The problem-based Optimize Live Editor task helps you try using different solvers on a problem, even solvers that have different syntaxes such as `fmincon` and `surrogateopt`. The task also helps you set plot functions, and set other options.

The task appears here in its final state. Feel free to experiment using different solvers and options.

Optimize

problem = Find feasible solution to problem

~ Create optimization variables

Name	Dimensions	Type	Lower bound	Upper bound	Initial point		
x	1x1	Continuous	-5	5	-2	-	+
y	1x1	Continuous	-10	10	-2	-	+

~ Define problem

Goal: Minimize Maximize Feasibility Solve equations

Constraints:

Define on one line					
$(y + x^2)^2 + 0.1y^2$	\leq	1	-	+	
y	\leq	$\exp(-x) - 3$	-	+	
y	\leq	$x - 4$	-	+	

~ Specify problem-dependent solver options

~ Display results

Problem Solution Reason solver stopped Objective value

Select task mode ?

Helper Functions

This code creates the `plotobjconstr` helper function.

```
function plotobjconstr
[XX,YY] = meshgrid(-2:0.1:2,-4:0.1:2);
ZZ = objconstr([XX(:),YY(:)]).Ineq;
ZZ = reshape(ZZ,[size(XX),3]);
h = figure;
ax = gca;
contour(ax,XX,YY,ZZ(:,:,1),[-1/2 0], 'r', 'ShowText', 'on');
hold on
contour(ax,XX,YY,ZZ(:,:,2),[-1/2 0], 'k', 'ShowText', 'on');
contour(ax,XX,YY,ZZ(:,:,3),[-1/2 0], 'b', 'ShowText', 'on');
hold off
end
```

This code creates the `objconstr` helper function.

```
function f = objconstr(x)
c(:,1) = (x(:,2) + x(:,1).^2).^2 + 0.1*x(:,2).^2 - 1;
c(:,2) = x(:,2) - exp(-x(:,1)) + 3;
c(:,3) = x(:,2) - x(:,1) + 4;
```

```
f.Ineq = c;  
end
```

This code creates the `surrout` helper function

```
function stop = surrout(~,optimValues,~)  
stop = false;  
if optimValues.constrviolation <= 1e-6 % Tolerance for constraint  
    stop = true;  
end  
end
```

See Also

[Optimize](#) | [fmincon](#) | [ga](#) | [surrogateopt](#)

Related Examples

- “Solve Feasibility Problem Using `surrogateopt`, Problem-Based” on page 12-6
- “Investigate Linear Infeasibilities”
- “Problem-Based Optimization Workflow”

Mixed-Integer Surrogate Optimization, Problem-Based

This example shows how to solve an optimization problem that involves integer variables. In this example, find the point x that minimizes the `multirosenbrock` function over integer-valued arguments ranging from -3 to 6 in 10 dimensions. The `multirosenbrock` function is a poorly scaled function that is difficult to optimize. Its minimum value is 0, which is attained at the point $[1, 1, \dots, 1]$. The code for the `multirosenbrock` function appears at the end of this example on page 12-22.

Create a 10-D row vector variable x of type integer with bounds -3 to 6. When you specify scalar bounds, the bounds apply to all variable components.

```
x = optimvar("x",1,10,"LowerBound",-3,"UpperBound",6,"Type","integer");
```

To use `multirosenbrock` as the objective function, convert the function to an optimization expression using `fcn2optimexpr`.

```
fun = fcn2optimexpr(@multirosenbrock,x);
```

Create an optimization problem with the objective function `multirosenbrock`.

```
prob = optimproblem("Objective",fun);
```

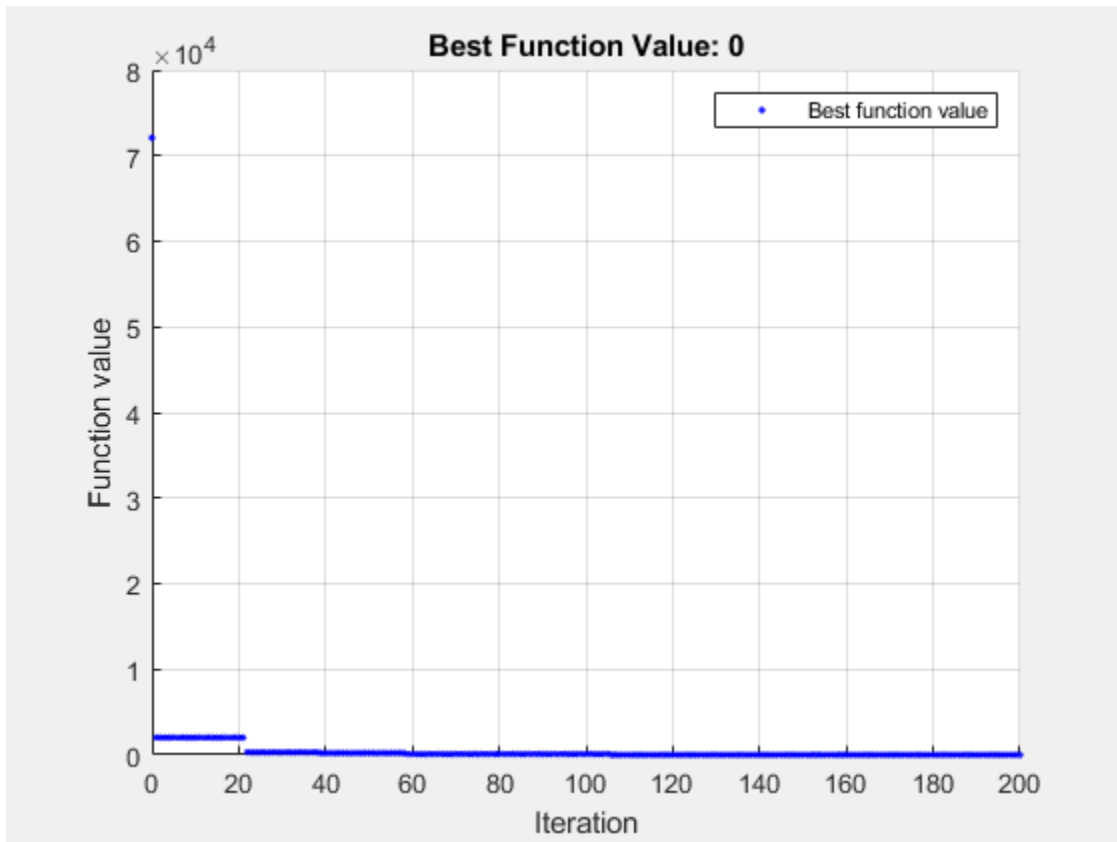
Set the maximum number of function evaluations to 200.

```
opts = optimoptions("surrogateopt","MaxFunctionEvaluations",200);
```

Solve the problem.

```
rng(1,'twister') % For reproducibility  
[sol,fval] = solve(prob,"Solver","surrogateopt","Options",opts)
```

```
Solving problem using surrogateopt.
```



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
sol = struct with fields:
    x: [1 1 1 1 1 1 1 1 1 1]
```

```
fval = 0
```

In this case, surrogateopt reaches the correct solution.

Mixed-Integer Problem

Suppose that only the first six variables are integer-valued. To reformulate the problem, create a 6-D integer variable `xint` and a 4-D continuous variable `xcont`.

```
xint = optimvar("xint",1,6,"LowerBound",-3,"UpperBound",6,"Type","integer");
xcont = optimvar("xcont",1,4,"LowerBound",-3,"UpperBound",6);
```

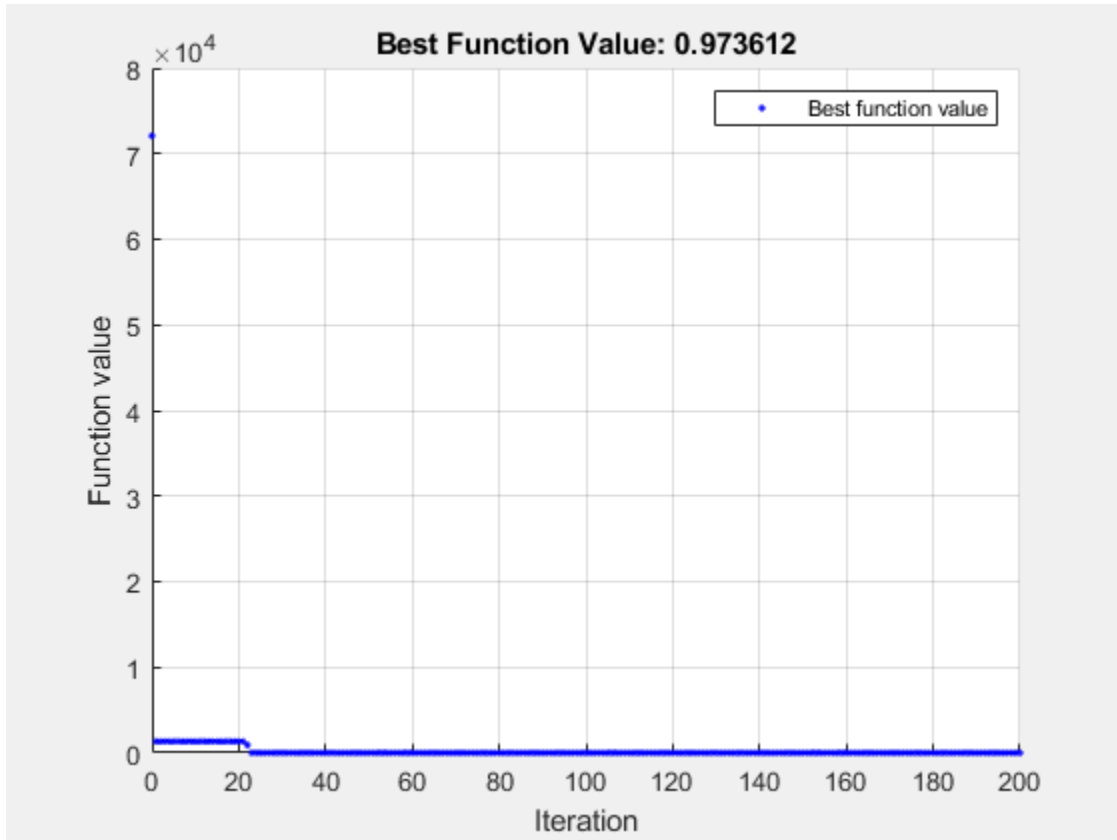
Convert `multirosenbrock` to an optimization expression using the input `[xint xcont]`.

```
fun2 = fcn2optimexpr(@multirosenbrock,[xint xcont]);
```

Create and solve the problem.

```
prob2 = optimproblem("Objective",fun2);
rng(1,'twister') % For reproducibility
[sol2,fval2] = solve(prob2,"Solver","surrogateopt","Options",opts)
```

Solving problem using surrogateopt.



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
sol2 = struct with fields:
    xcont: [1.2133 1.4719 1.1857 1.5003]
    xint: [1 1 1 1 1 1]
```

```
fval2 = 0.9736
```

This time the integer variables reach the correct solution, and the continuous variables are near the solution, but are not completely accurate.

Helper Function

This code creates the multirosenbrock helper function.

```
function F = multirosenbrock(x)
% This function is a multidimensional generalization of Rosenbrock's
% function. It operates in a vectorized manner, assuming that x is a matrix
% whose rows are the individuals.
% Copyright 2014 by The MathWorks, Inc.
N = size(x,2); % assumes x is a row vector or 2-D matrix
if mod(N,2) % if N is odd
    error('Input rows must have an even number of elements')
```



```
end
odds = 1:2:N-1;
evens = 2:2:N;
F = zeros(size(x));
F(:,odds) = 1-x(:,odds);
F(:,evens) = 10*(x(:,evens)-x(:,odds).^2);
F = sum(F.^2,2);
end
```

See Also

[solve](#) | [surrogateopt](#)

Related Examples

- “Mixed-Integer Surrogate Optimization” on page 11-62

Specify Starting Points and Values for `surrogateopt`, Problem-Based

For some solvers, you can pass the objective and constraint function values, if any, to `solve` in the `x0` argument. This can save time in the solver. Pass a vector of `OptimizationValues` objects. Create this vector using the `optimvalues` function.

The solvers that can use the objective function values are:

- `ga`
- `gamultiobj`
- `paretosearch`
- `surrogateopt`

The solvers that can use nonlinear constraint function values are:

- `paretosearch`
- `surrogateopt`

For example, minimize the `peaks` function using `surrogateopt`, starting with values from a grid of initial points. Create a grid from -10 to 10 in the `x` variable, and $-5/2$ to $5/2$ in the `y` variable with spacing $1/2$. Compute the objective function values at the initial points.

```
x = optimvar("x",LowerBound=-10,UpperBound=10);
y = optimvar("y",LowerBound=-5/2,UpperBound=5/2);
prob = optimproblem("Objective",peaks(x,y));
xval = -10:10;
yval = (-5:5)/2;
[x0x,x0y] = meshgrid(xval,yval);
peaksvals = peaks(x0x,x0y);
```

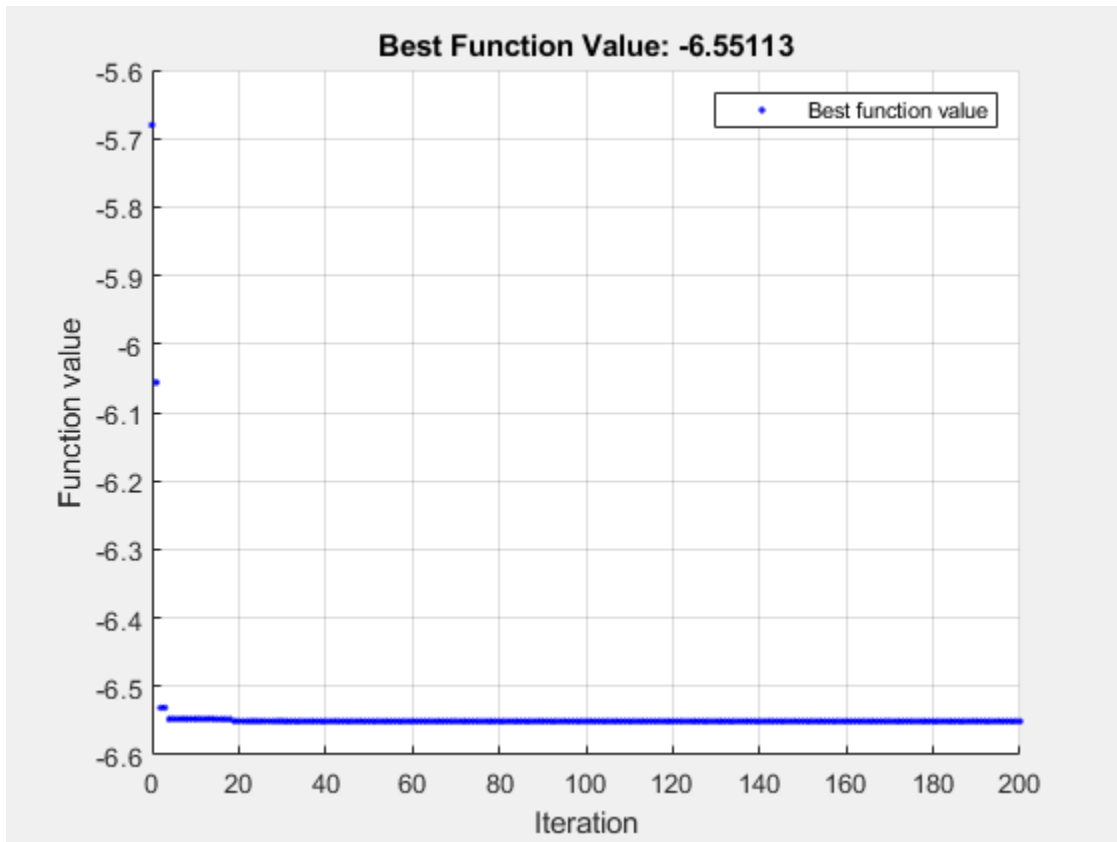
Pass the values in the `x0` argument by using `optimvalues`. This saves time for `solve`, as `solve` does not need to compute the values. Pass the values as row vectors.

```
x0 = optimvalues(prob,'x',x0x(:)','y',x0y(:)','...
    "Objective",peaksvals(:)');
```

Solve the problem using `surrogateopt` with the initial values.

```
[sol,fval,eflag,output] = solve(prob,x0,Solver="surrogateopt")
```

Solving problem using `surrogateopt`.



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

sol = struct with fields:

x: 0.2283
y: -1.6256

fval = -6.5511

eflag =

SolverLimitExceeded

output = struct with fields:

elapsedtime: 28.3878
funccount: 200
constrviolation: 0

ineq: [1x1 struct]

rngstate: [1x1 struct]

message: 'surrogateopt stopped because it exceeded the function evaluation limit set

solver: 'surrogateopt'

See Also

surrogateopt | solve | optimvalues

Related Examples

- “Specify Start Points for MultiStart, Problem-Based” on page 5-3

Using Simulated Annealing

- “What Is Simulated Annealing?” on page 13-2
- “Optimize Function Using `simulannealbnd`, Problem-Based” on page 13-3
- “Minimize Function with Many Local Minima” on page 13-5
- “Simulated Annealing Terminology” on page 13-12
- “How Simulated Annealing Works” on page 13-14
- “Reproduce Your Results” on page 13-17
- “Minimization Using Simulated Annealing Algorithm” on page 13-19
- “Simulated Annealing Options” on page 13-22
- “Multiprocessor Scheduling Using Simulated Annealing with a Custom Data Type” on page 13-28

What Is Simulated Annealing?

Simulated annealing is a method for solving unconstrained and bound-constrained optimization problems. The method models the physical process of heating a material and then slowly lowering the temperature to decrease defects, thus minimizing the system energy.

At each iteration of the simulated annealing algorithm, a new point is randomly generated. The distance of the new point from the current point, or the extent of the search, is based on a probability distribution with a scale proportional to the temperature. The algorithm accepts all new points that lower the objective, but also, with a certain probability, points that raise the objective. By accepting points that raise the objective, the algorithm avoids being trapped in local minima, and is able to explore globally for more possible solutions. An *annealing schedule* is selected to systematically decrease the temperature as the algorithm proceeds. As the temperature decreases, the algorithm reduces the extent of its search to converge to a minimum.

See Also

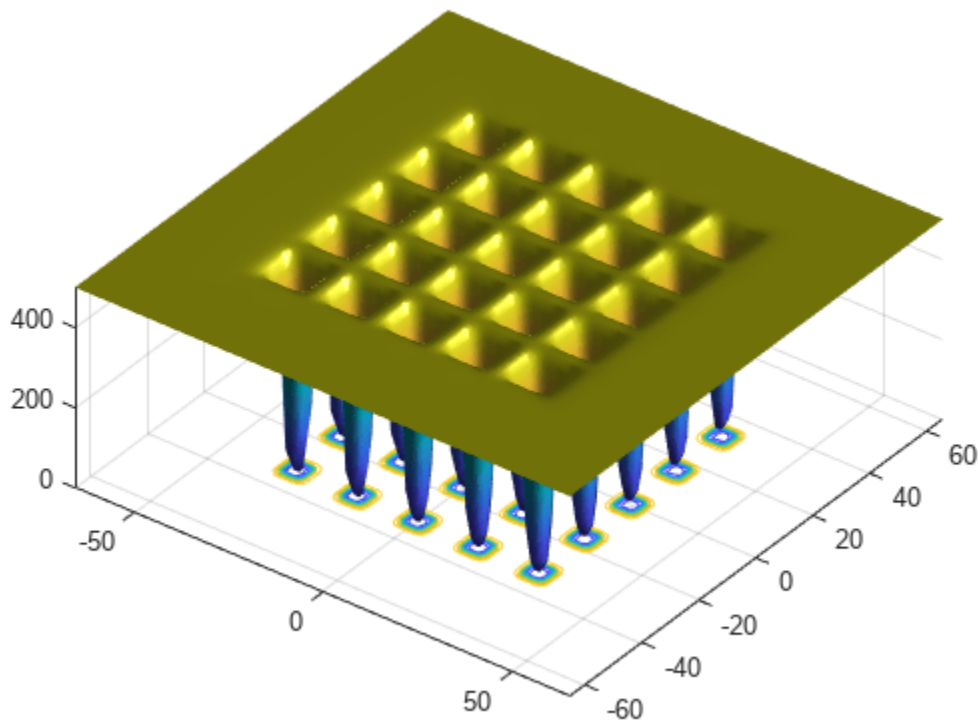
More About

- “Simulated Annealing Terminology” on page 13-12
- “How Simulated Annealing Works” on page 13-14
- “Minimize Function with Many Local Minima” on page 13-5
- “Minimization Using Simulated Annealing Algorithm” on page 13-19

Optimize Function Using `simulannealbnd`, Problem-Based

This example shows how to minimize a function using simulated annealing in the problem-based approach when the objective is a function file, possibly of unknown content (a "black box" function). The function to minimize, `dejong5fcn(x)`, is available when you run this example. Plot the function.

```
dejong5fcn
```



Create a 2-D optimization variable `x`. The `dejong5fcn` function expects the variable to be a row vector, so specify `x` as a 2-element row vector.

```
x = optimvar("x",1,2);
```

To use `dejong5fcn` as the objective function, convert the function to an optimization expression using `fcn2optimexpr`.

```
fun = fcn2optimexpr(@dejong5fcn,x);
```

Create an optimization problem with the objective function `fun`.

```
prob = optimproblem("Objective",fun);
```

Set variable bounds from -50 to 50 in all components. When you specify scalar bounds, the software expands the bounds to all variables.

```
x.LowerBound = -50;  
x.UpperBound = 50;
```

Set a pseudorandom initial point within the bounds. The initial point is a structure with field `x`.

```
rng default % For reproducibility  
x0.x = x.LowerBound + rand(size(x.LowerBound)).*x.UpperBound;
```

Solve the problem, specifying the `simulannealbnd` solver.

```
[sol,fval] = solve(prob,x0,"Solver","simulannealbnd")
```

```
Solving problem using simulannealbnd.  
Optimization terminated: change in best function value less than options.FunctionTolerance.
```

```
sol = struct with fields:  
  x: [-32.0371 -31.8792]
```

```
fval = 0.9980
```

See Also

`simulannealbnd` | `fcn2optimexpr` | `solve`

Related Examples

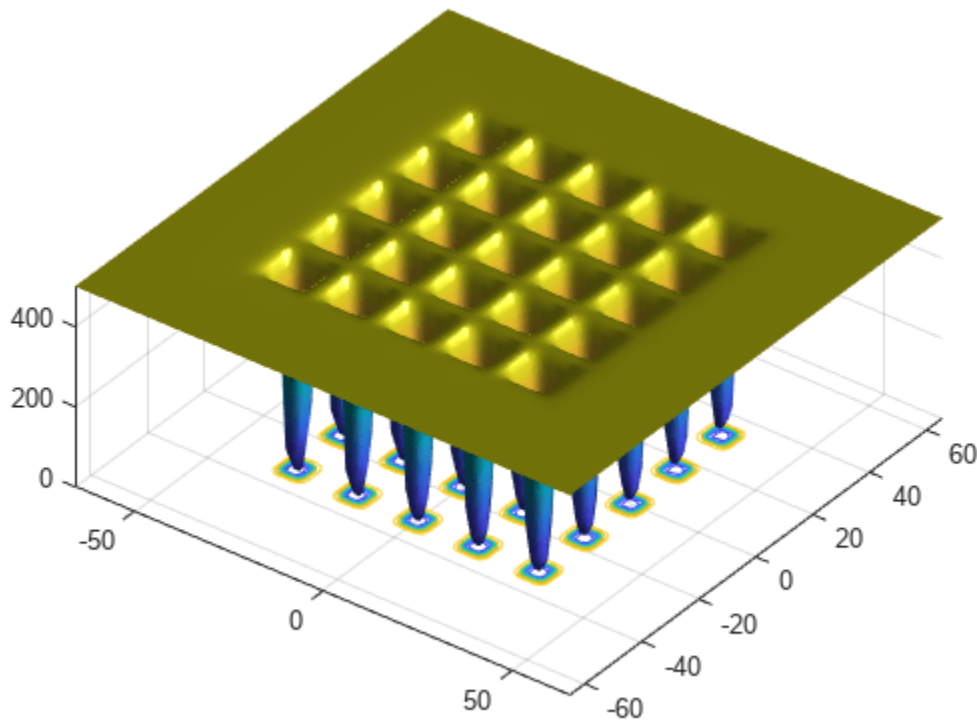
- “Simulated Annealing”

Minimize Function with Many Local Minima

This example shows how to find a local minimum of a function using simulated annealing. The example presents two approaches for minimizing: working at the command line and using the **Optimize** Live Editor task.

De Jong's fifth function is a two-dimensional function with many (25) local minima. The function is available when you run this example. In the following plot, it is unclear which of the local minima is the global minimum.

```
dejong5fcn
```



Many standard optimization algorithms become stuck in local minima. Because the simulated annealing algorithm performs a wide random search, the chance of being trapped in a local minimum is decreased.

Note: Because simulated annealing uses random number generators, each time you run this algorithm you can get different results. See “Reproduce Your Results” on page 13-17 for more information.

Minimize at the Command Line

To run the simulated annealing algorithm without constraints, call `simulannealbnd` at the command line using the objective function in `dejong5fcn.m`, referenced by the anonymous function `@dejong5fcn` in the following code.

```
rng(10, 'twister') % for reproducibility
fun = @dejong5fcn;
[x, fval] = simulannealbnd(fun, [0 0])
```

Optimization terminated: change in best function value less than options.FunctionTolerance.

x = 1×2

-16.1292 -15.8214

fval = 6.9034

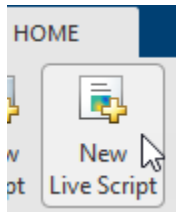
In the results:

- x is the final point returned by the algorithm.
- fval is the objective function value at the final point.

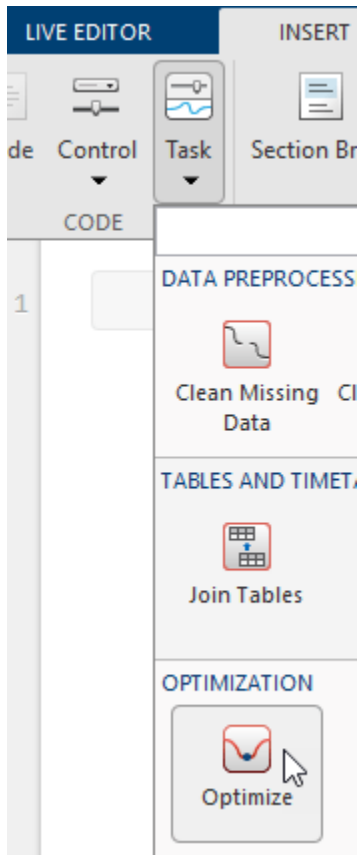
Minimize Using the Optimize Live Editor Task

You can also run the minimization using the Optimize Live Editor task, which provides a visual approach.

- Create a new live script by clicking the **New Live Script** button in the **File** section on the **Home** tab.





- Insert an Optimize Live Editor task. Click the **Insert** tab and then, in the **Code** section, select **Task > Optimize**.



Optimize

Solve an optimization problem or system of equations

Select approach

 <p>Problem-based (recommended)</p> <ul style="list-style-type: none">• Easier to define problem• Represents problem inputs symbolically• Built-in automatic differentiation	 <p>Solver-based</p> <ul style="list-style-type: none">• Start with a solver• Represents inputs as matrices/functions• Allows specialized solution methods
--	--


- Click the **Solver-based** task.


Optimize


Minimize a function with or without constraints


▼ **Specify problem type**


Objective


 Linear



 Quadratic



 Least squares



 Nonlinear



 Nonsmooth


Select an objective type to see example functions


 Unconstrained


 Lower bounds


 Upper bounds

 Linear inequality

 Linear equality

 Second-order cone

 Nonlinear

 Integer

Select constraint types to see example formulas

Solver fmincon - Constrained nonlinear minimization (recommended) ?

▼ **Select problem data**

Objective function From file ▼ Browse... New... ?

Initial point (x0) select ▼

► **Specify solver options**

▼ **Display progress**

Text display Final output ▼

Plot Current point Evaluation count Objective value and feasibility Objective value

Max constraint violation Step size Optimality measure

- For use in entering problem data, insert a new section by clicking the **Section Break** button on the **Insert** tab. New sections appear above and below the task.
- In the new section above the task, enter the following code to define the initial point and the objective function.

```
x0 = [0 0];
fun = @dejong5fcn;
rng default % For reproducibility
```

- To place these variables into the workspace, run the section by pressing **Ctrl+Enter**.
- In the **Specify problem type** section of the task, click the **Objective > Nonlinear** button.
- Select **Solver > simulannealbnd - Simulated annealing algorithm**.
- In the **Select problem data** section of the task, select **Objective function > Function handle** and then choose fun.

- Select **Initial point (x0) > x0**.

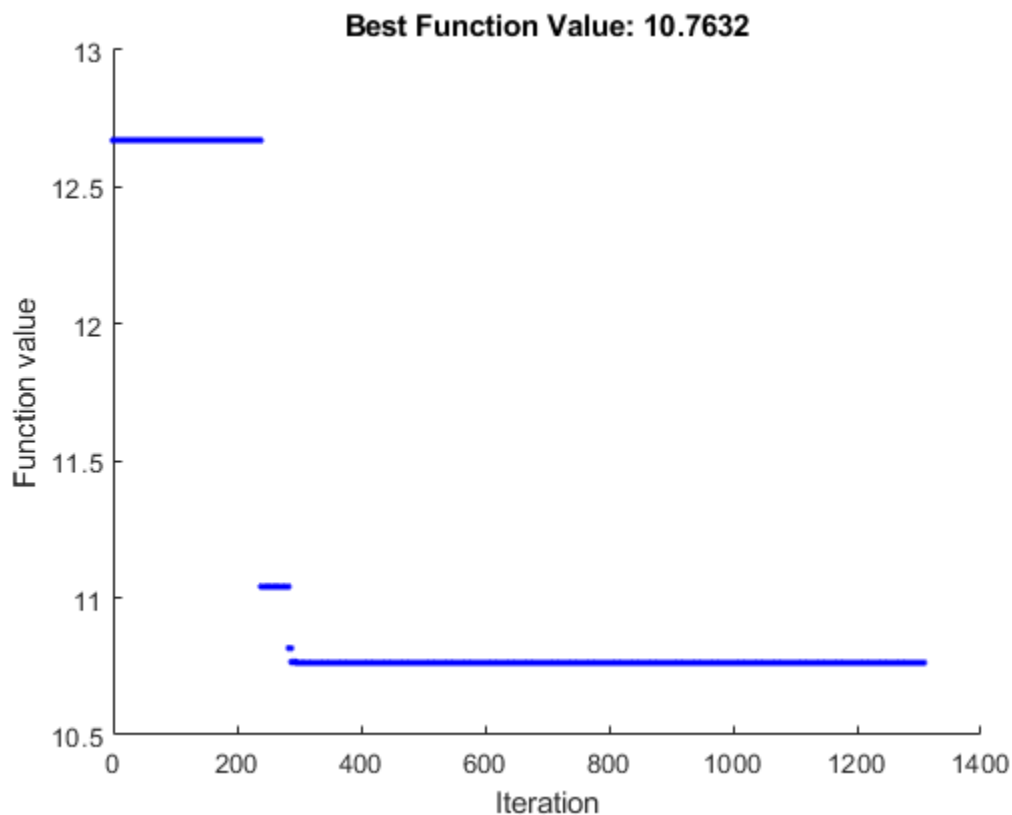
Solver

Select problem data

Objective function

Initial point (x0)

- In the **Display progress** section of the task, select the **Best value** plot.
- To run the solver, click the options button **:** at the top right of the task window, and select **Run Section**. The plot appears in a separate figure window and in the task output area. Note that your plot might be different from the one shown, because `simulannealbnd` is a stochastic algorithm.



- To see the solution and best objective function value, look at the top of the task.

Optimize

```
solution, objectiveValue = Minimize fun using simulannealbnd solver
```

- The **Optimize Live Editor** task returns variables named `solution` and `objectiveValue` to the workspace.
- To view the values these variables, enter the following code in the section below the task.

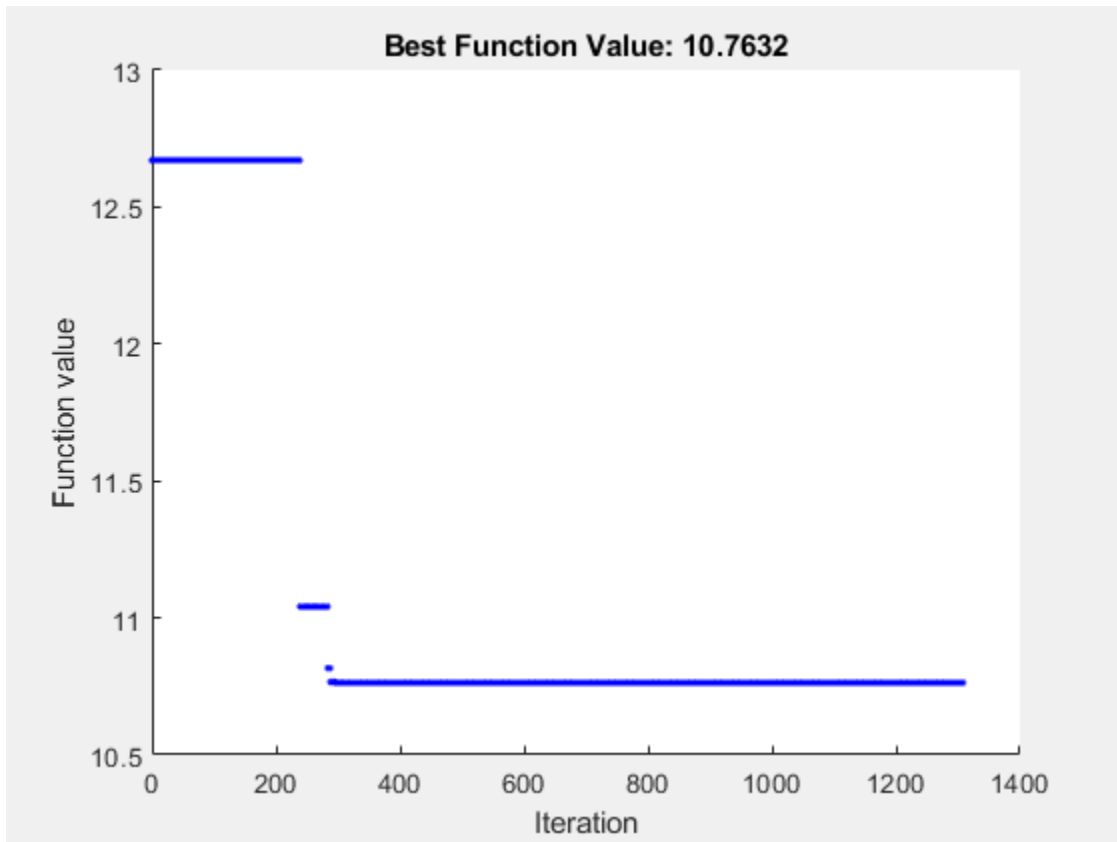
```
disp(solution)
disp(objectiveValue)
```

- Run the section by pressing **Ctrl+Enter**.

The end of this example has the Optimize task in its final state.

The screenshot shows the MATLAB **Optimize** app interface. At the top, the task is defined as: `solution, objectiveValue = Minimize fun using simulannealbnnd solver`. The interface is divided into several sections:

- Specify problem type:**
 - Objective:** Includes icons for Linear, Quadratic, Least squares, Nonlinear (selected), and Nonsmooth. Below these are examples: $f(x, y) = x/y$, $f(x) = \cos(x)$, $f(x) = \log(x)$, $f(x) = e^x$, $f(x) = x^3$, and $\text{Solve } F(x) = 0, \dots$.
 - Constraints:** Includes icons for Unconstrained, Lower bounds, Upper bounds, Linear inequality, Linear equality, Second-order cone, Nonlinear, and Integer.
 - Solver:** A dropdown menu is set to `simulannealbnnd - Simulated annealing algorithm`.
- Select problem data:**
 - Objective function:** Set to `Function handle` with the value `fun`.
 - Initial point (x0):** Set to `x0`.
- Specify solver options:** (Currently collapsed)
- Display progress:**
 - Text display:** Set to `Final output`.
 - Plot:** Checkboxes for `Best value` (checked), `Best point`, `Current value`, `Stopping criteria`, and `Current temperature`.



Optimization terminated: change in best function value less than options.FunctionTolerance.

Both the `Optimize` Live Editor task and the command line allow you to formulate and solve problems, and they give identical results. The command line is more streamlined, but provides less help for choosing a solver, setting up the problem, and choosing options such as plot functions. You can also start a problem using `Optimize`, and then generate code for command line use, as in “Constrained Nonlinear Problem Using `Optimize` Live Editor Task or Solver”.

See Also

`simulannealbnd`

More About

- “Minimization Using Simulated Annealing Algorithm” on page 13-19
- “Add Interactive Tasks to a Live Script”

Simulated Annealing Terminology

In this section...
“Objective Function” on page 13-12
“Temperature” on page 13-12
“Annealing Parameter” on page 13-12
“Reannealing” on page 13-12

Objective Function

The *objective function* is the function you want to optimize. Global Optimization Toolbox algorithms attempt to find the minimum of the objective function. Write the objective function as a file or anonymous function, and pass it to the solver as a function handle. For more information, see “Compute Objective Functions” on page 2-2 and “Create Function Handle”.

Temperature

The *temperature* is a parameter in simulated annealing that affects two aspects of the algorithm:

- The distance of a trial point from the current point (See “Outline of the Algorithm” on page 13-14, Step 1.)
- The probability of accepting a trial point with higher objective function value (See “Outline of the Algorithm” on page 13-14, Step 2.)

Temperature can be a vector with different values for each component of the current point. Typically, the initial temperature is a scalar.

Temperature decreases gradually as the algorithm proceeds. You can specify the initial temperature as a positive scalar or vector in the `InitialTemperature` option. You can specify the temperature as a function of iteration number as a function handle in the `TemperatureFcn` option. The temperature is a function of the “Annealing Parameter” on page 13-12, which is a proxy for the iteration number. The slower the rate of temperature decrease, the better the chances are of finding an optimal solution, but the longer the run time. For a list of built-in temperature functions and the syntax of a custom temperature function, see “Temperature Options” on page 17-59.

Annealing Parameter

The *annealing parameter* is a proxy for the iteration number. The algorithm can raise temperature by setting the annealing parameter to a lower value than the current iteration. (See “Reannealing” on page 13-12.) You can specify the temperature schedule as a function handle with the `TemperatureFcn` option.

Reannealing

Annealing is the technique of closely controlling the temperature when cooling a material to ensure that it reaches an optimal state. *Reannealing* raises the temperature after the algorithm accepts a certain number of new points, and starts the search again at the higher temperature. Reannealing avoids the algorithm getting caught at local minima. Specify the reannealing schedule with the `ReannealInterval` option.

See Also

simulannealbnd

More About

- “What Is Simulated Annealing?” on page 13-2
- “How Simulated Annealing Works” on page 13-14

How Simulated Annealing Works

In this section...

“Outline of the Algorithm” on page 13-14

“Stopping Conditions for the Algorithm” on page 13-15

“Bibliography” on page 13-16

Outline of the Algorithm

The simulated annealing algorithm performs the following steps:

- 1 The algorithm generates a random trial point. The algorithm chooses the distance of the trial point from the current point by a probability distribution with a scale depending on the current temperature. You set the trial point distance distribution as a function with the `AnnealingFcn` option. Choices:

- `@annealingfast` (default) — Step length equals the current temperature, and direction is uniformly random.
- `@annealingboltz` — Step length equals the square root of temperature, and direction is uniformly random.
- `@myfun` — Custom annealing algorithm, `myfun`. For custom annealing function syntax, see “Algorithm Settings” on page 17-60.

After generating the trial point, the algorithm shifts it, if necessary, to stay within bounds. The algorithm shifts each infeasible component of the trial point to a value chosen uniformly at random between the violated bound and the (feasible) value at the previous iteration.

- 2 The algorithm determines whether the new point is better or worse than the current point. If the new point is better than the current point, it becomes the next point. If the new point is worse than the current point, the algorithm can still make it the next point. The algorithm accepts a worse point based on an acceptance function. Choose the acceptance function with the `AcceptanceFcn` option. Choices:

- `@acceptancesa` (default) — Simulated annealing acceptance function. The probability of acceptance is

$$\frac{1}{1 + \exp\left(\frac{\Delta}{\max(T)}\right)},$$

where

Δ = new objective - old objective.

T_0 = initial temperature of component i

T = the current temperature.

Since both Δ and T are positive, the probability of acceptance is between 0 and 1/2. Smaller temperature leads to smaller acceptance probability. Also, larger Δ leads to smaller acceptance probability.

- `@myfun` — Custom acceptance function, `myfun`. For custom acceptance function syntax, see “Algorithm Settings” on page 17-60.

- 3 The algorithm systematically lowers the temperature, storing the best point found so far. The `TemperatureFcn` option specifies the function the algorithm uses to update the temperature. Let k denote the annealing parameter. (The annealing parameter is the same as the iteration number until reannealing.) Options:
- `@temperatureexp` (default) — $T = T_0 * 0.95^k$.
 - `@temperaturefast` — $T = T_0 / k$.
 - `@temperatureboltz` — $T = T_0 / \log(k)$.
 - `@myfun` — Custom temperature function, `myfun`. For custom temperature function syntax, see “Temperature Options” on page 17-59.
- 4 `simulannealbnd` reanneals after it accepts `ReannealInterval` points. Reannealing sets the annealing parameters to lower values than the iteration number, thus raising the temperature in each dimension. The annealing parameters depend on the values of estimated gradients of the objective function in each dimension. The basic formula is

$$k_i = \log \left(\frac{T_0 \max_j(s_j)}{T_i s_i} \right),$$

where

k_i = annealing parameter for component i .

T_0 = initial temperature of component i .

T_i = current temperature of component i .

s_i = gradient of objective in direction i times difference of bounds in direction i .

`simulannealbnd` safeguards the annealing parameter values against `Inf` and other improper values.

- 5 The algorithm stops when the average change in the objective function is small relative to `FunctionTolerance`, or when it reaches any other stopping criterion. See “Stopping Conditions for the Algorithm” on page 13-15.

For more information on the algorithm, see Ingber [1].

Stopping Conditions for the Algorithm

The simulated annealing algorithm uses the following conditions to determine when to stop:

- `FunctionTolerance` — The algorithm runs until the average change in value of the objective function in `StallIterLim` iterations is less than the value of `FunctionTolerance`. The default value is `1e-6`.
- `MaxIterations` — The algorithm stops when the number of iterations exceeds this maximum number of iterations. You can specify the maximum number of iterations as a positive integer or `Inf`. The default value is `Inf`.
- `MaxFunctionEvaluations` specifies the maximum number of evaluations of the objective function. The algorithm stops if the number of function evaluations exceeds the value of `MaxFunctionEvaluations`. The default value is `3000*numberofvariables`.
- `MaxTime` specifies the maximum time in seconds the algorithm runs before stopping. The default value is `Inf`.

- `ObjectiveLimit` — The algorithm stops when the best objective function value is less than the value of `ObjectiveLimit`. The default value is `-Inf`.

Bibliography

[1] Ingber, L. *Adaptive simulated annealing (ASA): Lessons learned*. Invited paper to a special issue of the *Polish Journal Control and Cybernetics* on “Simulated Annealing Applied to Combinatorial Optimization.” 1995. Available from https://www.ingber.com/asa96_lessons.ps.gz

See Also

`simulannealbnd`

More About

- “What Is Simulated Annealing?” on page 13-2
- “Simulated Annealing Terminology” on page 13-12
- “Minimize Function with Many Local Minima” on page 13-5
- “Minimization Using Simulated Annealing Algorithm” on page 13-19

Reproduce Your Results

Because the simulated annealing algorithm is stochastic—that is, it makes random choices—you get slightly different results each time you run it. The algorithm uses the default MATLAB® pseudorandom number stream. For more information about random number streams, see `RandStream`. Each time the algorithm calls the stream, its state changes. So the next time the algorithm calls the stream, the stream returns a different random number.

If you need to reproduce your results exactly, call `simulannealbnd` with the `output` argument. The `output` structure contains the current random number generator state in the `output.rngstate` field. Reset the state before running the function again.

For example, to reproduce the output of `simulannealbnd` applied to De Jong's fifth function (which is available when you run this example), call `simulannealbnd` with the syntax

```
rng(10,'twister') % For reproducibility
[x,fval,exitflag,output] = simulannealbnd(@dejong5fcn,[0 0]);
```

Optimization terminated: change in best function value less than options.FunctionTolerance.

Examine the `x` and `fval` outputs.

```
x,fval
x = 1×2
    -16.1292    -15.8214
```

```
fval = 6.9034
```

The state of the random number generator, `rngstate`, is stored in `output.rngstate`. Reset the stream as follows.

```
stream = RandStream.getGlobalStream;
stream.State = output.rngstate.State;
```

Run `simulannealbnd` a second time, and you get the same results.

```
[x,fval,exitflag,output] = simulannealbnd(@dejong5fcn,[0 0]);
```

Optimization terminated: change in best function value less than options.FunctionTolerance.

```
x,fval
x = 1×2
    -16.1292    -15.8214
```

```
fval = 6.9034
```

If you run `simulannealbnd` again without resetting the random number stream, the results change.

```
[x,fval,exitflag,output] = simulannealbnd(@dejong5fcn,[0 0]);
```

Optimization terminated: change in best function value less than options.FunctionTolerance.

```
x, fval
```

```
x = 1×2
```

```
    0    0
```

```
fval = 12.6705
```

Note: If you do not need to reproduce your results, it is better not to set the states of `RandStream`, so that you get the benefit of the randomness in these algorithms.

See Also

`simulannealbnd`

More About

- “Simulated Annealing”

Minimization Using Simulated Annealing Algorithm

This example shows how to create and minimize an objective function using the simulated annealing algorithm (`simulannealbnd` function) in Global Optimization Toolbox. For algorithmic details, see “How Simulated Annealing Works” on page 13-14.

Simple Objective Function

The objective function to minimize is a simple function of two variables:

$$\min_x f(x) = (4 - 2.1*x_1^2 + x_1^4/3)*x_1^2 + x_1*x_2 + (-4 + 4*x_2^2)*x_2^2;$$

This function is known as "cam," as described in L.C.W. Dixon and G.P. Szego [1].

To implement the objective function calculation, the MATLAB® file `simple_objective.m` has the following code:

```
type simple_objective

function y = simple_objective(x)
%SIMPLE_OBJECTIVE Objective function for PATTERNSEARCH solver

% Copyright 2004 The MathWorks, Inc.

x1 = x(1);
x2 = x(2);
y = (4-2.1.*x1.^2+x1.^4./3).*x1.^2+x1.*x2+(-4+4.*x2.^2).*x2.^2;
```

All Global Optimization Toolbox solvers assume that the objective has one input `x`, where `x` has as many elements as the number of variables in the problem. The objective function computes the scalar value of the objective function and returns it in its single output argument `y`.

Minimize Using `simulannealbnd`

To minimize the objective function using `simulannealbnd`, pass in a function handle to the objective function and a starting point `x0` as the second argument. For reproducibility, set the random number stream.

```
ObjectiveFunction = @simple_objective;
x0 = [0.5 0.5]; % Starting point
rng default % For reproducibility
[x,fval,exitFlag,output] = simulannealbnd(ObjectiveFunction,x0)

Optimization terminated: change in best function value less than options.FunctionTolerance.

x = 1x2

    -0.0896    0.7130

fval = -1.0316

exitFlag = 1

output = struct with fields:
    iterations: 2948
```

```

    funccount: 2971
      message: 'Optimization terminated: change in best function value less than options.Funct
    rngstate: [1x1 struct]
  problemtype: 'unconstrained'
  temperature: [2x1 double]
    totaltime: 1.1844

```

`simulannealbnd` returns four output arguments:

- `x` — Best point found
- `fval` — Function value at the best point
- `exitFlag` — Integer corresponding to the reason the function stopped
- `output` — Information about the optimization steps

Bound Constrained Minimization

You can use `simulannealbnd` to solve problems with bound constraints. Pass lower and upper bounds as vectors. For each coordinate `i`, the solver ensures that $lb(i) \leq x(i) \leq ub(i)$. Impose the bounds $-64 \leq x(i) \leq 64$.

```

lb = [-64 -64];
ub = [64 64];

```

Run the solver with the lower and upper bound arguments.

```
[x,fval,exitFlag,output] = simulannealbnd(ObjectiveFunction,x0,lb,ub);
```

```
Optimization terminated: change in best function value less than options.FunctionTolerance.
```

```
fprintf('The number of iterations was : %d\n', output.iterations);
```

```
The number of iterations was : 2428
```

```
fprintf('The number of function evaluations was : %d\n', output.funccount);
```

```
The number of function evaluations was : 2447
```

```
fprintf('The best function value found was : %g\n', fval);
```

```
The best function value found was : -1.03163
```

The solver finds essentially the same solution as before.

Minimize Using Additional Arguments

Sometimes you want an objective function to be parameterized by extra arguments that act as constants during the optimization. For example, in the previous objective function, you might want to replace the constants 4, 2.1, and 4 with parameters that you can change to create a family of objective functions. For more information, see “Passing Extra Parameters”.

Rewrite the objective function to take three additional parameters in a new minimization problem.

```

min f(x) = (a - b*x1^2 + x1^4/3)*x1^2 + x1*x2 + (-c + c*x2^2)*x2^2;
x

```


a, b, and c are parameters to the objective function that act as constants during the optimization (they are not varied as part of the minimization). To implement the objective function calculation, the MATLAB file `parameterized_objective.m` contains the following code:

```
type parameterized_objective

function y = parameterized_objective(x,p1,p2,p3)
%PARAMETERIZED_OBJECTIVE Objective function for PATTERNSEARCH solver

% Copyright 2004 The MathWorks, Inc.

x1 = x(1);
x2 = x(2);
y = (p1-p2.*x1.^2+x1.^4./3).*x1.^2+x1.*x2+(-p3+p3.*x2.^2).*x2.^2;
```

Again, you need to pass in a function handle to the objective function as well as a starting point as the second argument.

`simulannealbnd` calls the objective function with just one argument `x`, but the objective function has four arguments: `x`, `a`, `b`, and `c`. To indicate which variable is the argument, use an anonymous function to capture the values of the additional arguments (the constants `a`, `b`, and `c`). Create a function handle `ObjectiveFunction` to an anonymous function that takes one input `x`, but calls `parameterized_objective` with `x`, `a`, `b` and `c`. When you create the function handle `ObjectiveFunction`, the variables `a`, `b`, and `c` have values that are stored in the anonymous function.

```
a = 4; b = 2.1; c = 4; % Define constant values
ObjectiveFunction = @(x) parameterized_objective(x,a,b,c);
x0 = [0.5 0.5];
[x,fval] = simulannealbnd(ObjectiveFunction,x0)
```

Optimization terminated: change in best function value less than options.FunctionTolerance.

```
x = 1x2
    0.0898    -0.7127
```

```
fval = -1.0316
```

The solver finds essentially the same solution as before.

References

[1] Dixon, L. C. W., and G .P. Szego (eds.). *Towards Global Optimisation 2*. North-Holland: Elsevier Science Ltd., Amsterdam, 1978.

See Also

`simulannealbnd`

More About

- “Minimize Function with Many Local Minima” on page 13-5
- “What Is Simulated Annealing?” on page 13-2
- “Passing Extra Parameters”

Simulated Annealing Options

This example shows how to create and manage options for the simulated annealing function `simulannealbnd` using `optimoptions` in the Global Optimization Toolbox.

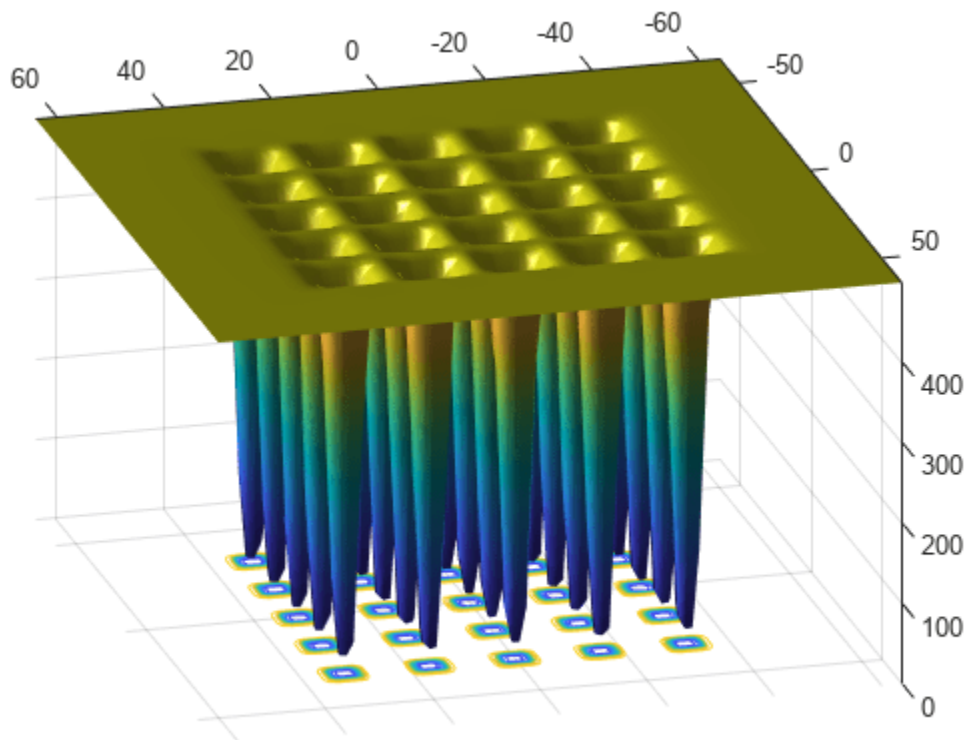
Optimization Problem Setup

`simulannealbnd` searches for a minimum of a function using simulated annealing. For this example we use `simulannealbnd` to minimize the objective function `dejong5fcn`. This function is available when you run this example. `dejong5fcn` is a real-valued function of two variables and has many local minima making it difficult to optimize. There is only one global minimum at $x = (-32, -32)$, where $f(x) = 0.998$. To define our problem, we must define the objective function, start point, and bounds specified by the range $-64 \leq x(i) \leq 64$ for each $x(i)$.

```
ObjectiveFunction = @dejong5fcn;
startingPoint = [-30 0];
lb = [-64 -64];
ub = [64 64];
```

The function `plotobjective`, which is available when you run this example, plots the objective function over the range $-64 \leq x_1 \leq 64$, $-64 \leq x_2 \leq 64$.

```
plotobjective(ObjectiveFunction,[-64 64; -64 64]);
view(-15,150);
```



Now, we can run the `simulannealbnd` solver to minimize our objective function.

```

rng default % For reproducibility
[x,fval,exitFlag,output] = simulannealbnd(ObjectiveFunction,startingPoint,lb,ub);

Optimization terminated: change in best function value less than options.FunctionTolerance.

fprintf('The number of iterations was : %d\n', output.iterations);
The number of iterations was : 1095

fprintf('The number of function evaluations was : %d\n', output.funccount);
The number of function evaluations was : 1104

fprintf('The best function value found was : %g\n', fval);
The best function value found was : 2.98211

```

Note that when you run this example, your results may be different from the results shown above because simulated annealing algorithm uses random numbers to generate points.

Adding Visualization

`simulannealbnd` can accept one or more plot functions through an 'options' argument. This feature is useful for visualizing the performance of the solver at run time. Plot functions are selected using `optimoptions`. The toolbox contains a set of plot functions to choose from, or you can provide your own custom plot functions.

To select multiple plot functions, set the `PlotFcn` option via the `optimoptions` function. For this example, we select `saplotbestf`, which plots the best function value every iteration, `saplottemperature`, which shows the current temperature in each dimension at every iteration, `saplotf`, which shows the current function value (remember that the current value is not necessarily the best one), and `saplotstopping`, which plots the percentage of stopping criteria satisfied every ten iterations.

```

options = optimoptions(@simulannealbnd, ...
    'PlotFcn',{@saplotbestf,@saplottemperature,@saplotf,@saplotstopping});

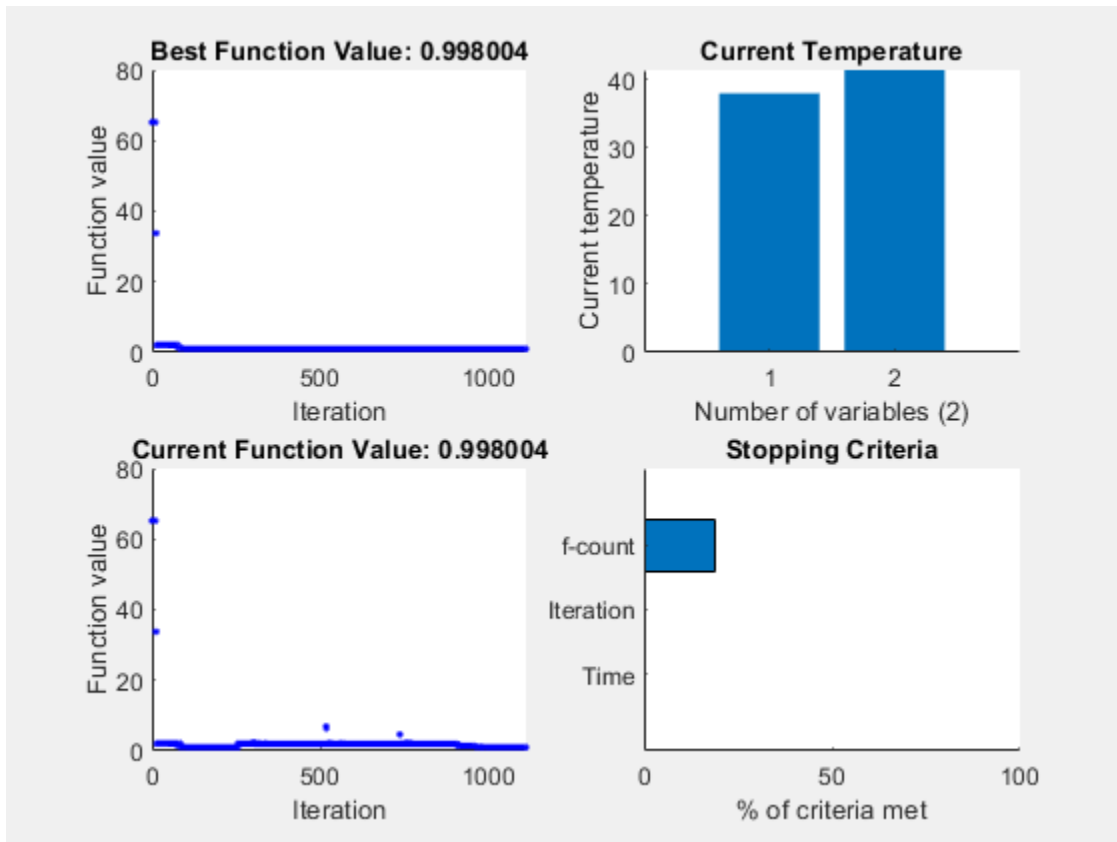
```

Run the solver.

```

simulannealbnd(ObjectiveFunction,startingPoint,lb,ub,options);

```



Optimization terminated: change in best function value less than options.FunctionTolerance.

Specifying Temperature Options

The temperature parameter used in simulated annealing controls the overall search results. The temperature for each dimension is used to limit the extent of search in that dimension. The toolbox lets you specify initial temperature as well as ways to update temperature during the solution process. The two temperature-related options are the `InitialTemperature` and the `TemperatureFcn`.

Specifying initial temperature

The default initial temperature is set to 100 for each dimension. If you want the initial temperature to be different in different dimensions then you must specify a vector of temperatures. This may be necessary in cases when problem is scaled differently in each dimension. For example,

```
options = optimoptions(@simulannealbnd,'InitialTemperature',[300 50]);
```

`InitialTemperature` can be set to a vector of length less than the number of variables (dimension); the solver expands the vector to the remaining dimensions by taking the last element of the initial temperature vector. Here we want the initial temperature to be the same in all dimensions so we need only specify the single temperature.

```
options.InitialTemperature = 100;
```

Specifying a temperature function

The default temperature function used by `simulannealbnd` is called `temperatureexp`. In the `temperatureexp` schedule, the temperature at any given step is .95 times the temperature at the previous step. This causes the temperature to go down slowly at first but ultimately get cooler faster than other schemes. If another scheme is desired, e.g. Boltzmann schedule or "Fast" schedule annealing, then `temperatureboltz` or `temperaturefast` can be used respectively. To select the fast temperature schedule, we can update our previously created options, changing `TemperatureFcn` directly.

```
options.TemperatureFcn = @temperaturefast;
```

Specifying reannealing

Reannealing is a part of annealing process. After a certain number of new points are accepted, the temperature is raised to a higher value in hope to restart the search and move out of a local minima. Performing reannealing too soon may not help the solver identify a minimum, so a relatively high interval is a good choice. The interval at which reannealing happens can be set using the `ReannealInterval` option. Here, we reduce the default reannealing interval to 50 because the function seems to be flat in many regions and solver might get stuck rapidly.

```
options.ReannealInterval = 50;
```

Now that we have set up the new temperature options we run the solver again.

```
[x,fval,exitFlag,output] = simulannealbnd(ObjectiveFunction,startingPoint,lb,ub,options);
Optimization terminated: change in best function value less than options.FunctionTolerance.

fprintf('The number of iterations was : %d\n', output.iterations);
The number of iterations was : 1306

fprintf('The number of function evaluations was : %d\n', output.funccount);
The number of function evaluations was : 1321

fprintf('The best function value found was : %g\n', fval);
The best function value found was : 16.4409
```

Reproducing Results

`simulannealbnd` is a nondeterministic algorithm. This means that running the solver more than once without changing any settings may give different results. This is because `simulannealbnd` utilizes MATLAB® random number generators when it generates subsequent points and also when it determines whether or not to accept new points. Every time a random number is generated the state of the random number generators change.

To see this, two runs of `simulannealbnd` solver yields:

```
[x,fval] = simulannealbnd(ObjectiveFunction,startingPoint,lb,ub,options);
Optimization terminated: change in best function value less than options.FunctionTolerance.

fprintf('The best function value found was : %g\n', fval);
The best function value found was : 1.99203
```

And,

```
[x,fval] = simulannealbnd(ObjectiveFunction,startingPoint,lb,ub,options);
Optimization terminated: change in best function value less than options.FunctionTolerance.
fprintf('The best function value found was : %g\n', fval);
The best function value found was : 10.7632
```

In the previous two runs `simulannealbnd` gives different results.

We can reproduce our results if we reset the states of the random number generators between runs of the solver by using information returned by `simulannealbnd`. `simulannealbnd` returns the states of the random number generators at the time `simulannealbnd` is called in the output argument. This information can be used to reset the states. Here we reset the states between runs using this output information so the results of the next two runs are the same.

```
[x,fval,exitFlag,output] = simulannealbnd(ObjectiveFunction,startingPoint,lb,ub,options);
Optimization terminated: change in best function value less than options.FunctionTolerance.
fprintf('The best function value found was : %g\n', fval);
The best function value found was : 20.1535
```

We reset the state of the random number generator.

```
strm = RandStream.getGlobalStream;
strm.State = output.rngstate.State;
```

Now, let's run `simulannealbnd` again.

```
[x,fval] = simulannealbnd(ObjectiveFunction,startingPoint,lb,ub,options);
Optimization terminated: change in best function value less than options.FunctionTolerance.
fprintf('The best function value found was : %g\n', fval);
The best function value found was : 20.1535
```

Modifying the Stopping Criteria

`simulannealbnd` uses six different criteria to determine when to stop the solver. `simulannealbnd` stops when the maximum number of iterations or function evaluation is exceeded; by default the maximum number of iterations is set to `Inf` and the maximum number of function evaluations is `3000*numberOfVariables`. `simulannealbnd` keeps track of the average change in the function value for `MaxStallIterations` iterations. If the average change is smaller than the function tolerance, `FunctionTolerance`, then the algorithm will stop. The solver will also stop when the objective function value reaches `ObjectiveLimit`. Finally the solver will stop after running for `MaxTime` seconds. Here we set the `FunctionTolerance` to `1e-5`.

```
options.FunctionTolerance = 1e-5;
```

Run the `simulannealbnd` solver.

```
[x,fval,exitFlag,output] = simulannealbnd(ObjectiveFunction,startingPoint,lb,ub,options);
Optimization terminated: change in best function value less than options.FunctionTolerance.
fprintf('The number of iterations was : %d\n', output.iterations);
```

The number of iterations was : 1843

```
fprintf('The number of function evaluations was : %d\n', output.funccount);
```

The number of function evaluations was : 1864

```
fprintf('The best function value found was : %g\n', fval);
```

The best function value found was : 6.90334

See Also

`simulannealbnd`

More About

- “Simulated Annealing Options” on page 17-58
- “How Simulated Annealing Works” on page 13-14

Multiprocessor Scheduling Using Simulated Annealing with a Custom Data Type

This example shows how to use simulated annealing to minimize a function using a custom data type. Here simulated annealing is customized to solve the multiprocessor scheduling problem.

Multiprocessor Scheduling Problem

The multiprocessor scheduling problem consists of finding an optimal distribution of tasks on a set of processors. The number of processors and number of tasks are given. Time taken to complete a task by a processor is also provided as data. Each processor runs independently, but each can only run one job at a time. We call an assignment of all jobs to available processors a "schedule". The goal of the problem is to determine the shortest schedule for the given set of tasks.

First we determine how to express this problem in terms of a custom data type optimization problem that `simulannealbnd` function can solve. We come up with the following scheme: first, let each task be represented by an integer between 1 and the total number of tasks. Similarly, each processor is represented by an integer between 1 and the number of processors. Now we can store the amount of time a given job will take on a given processor in a matrix called "lengths". The amount of time "t" that the processor "i" takes to complete the task "j" will be stored in `lengths(i,j)`.

We can represent a schedule in a similar manner. In a given schedule, the rows (integer between 1 to number of processors) will represent the processors and the columns (integer between 1 to number of tasks) will represent the tasks. For example, the schedule `[1 2 3;4 5 0;6 0 0]` would be tasks 1, 2, and 3 performed on processor 1, tasks 4 and 5 performed on processor 2, and task 6 performed on processor 3.

Here we define our number of tasks, number of processors, and lengths array. The different coefficients for the various rows represent the fact that different processors work with different speeds. We also define a "sampleSchedule" which will be our starting point input to `simulannealbnd`.

```
rng default % for reproducibility
numberOfProcessors = 11;
numberOfTasks = 40;
lengths = [10*rand(1,numberOfTasks);
           7*rand(1,numberOfTasks);
           2*rand(1,numberOfTasks);
           5*rand(1,numberOfTasks);
           3*rand(1,numberOfTasks);
           4*rand(1,numberOfTasks);
           1*rand(1,numberOfTasks);
           6*rand(1,numberOfTasks);
           4*rand(1,numberOfTasks);
           3*rand(1,numberOfTasks);
           1*rand(1,numberOfTasks)];

% Random distribution of task on processors (starting point)
sampleSchedule = zeros(numberOfProcessors,numberOfTasks);
for task = 1:numberOfTasks
    processorID = 1 + floor(rand*(numberOfProcessors));
    index = find(sampleSchedule(processorID,:)==0);
    sampleSchedule(processorID,index(1)) = task;
end
```


Simulated Annealing For a Custom Data Type

By default, the simulated annealing algorithm solves optimization problems assuming that the decision variables are double data types. Therefore, the annealing function for generating subsequent points assumes that the current point is a vector of type double. However, if the `DataType` option is set to 'custom' the simulated annealing solver can also work on optimization problems involving arbitrary data types. You can use any valid MATLAB® data structure you like as decision variable. For example, if we want `simulannealbnd` to use "sampleSchedule" as decision variable, a custom data type can be specified using a matrix of integers. In addition to setting the `DataType` option to 'custom' we also need to provide a custom annealing function via the `AnnealingFcn` option that can generate new points.

Custom Annealing Functions

This section shows how to create and use the required custom annealing function. A trial point for the multiprocessor scheduling problem is a matrix of processor (rows) and tasks (columns) as discussed before. The custom annealing function for the multiprocessor scheduling problem will take a job schedule as input. The annealing function will then modify this schedule and return a new schedule that has been changed by an amount proportional to the temperature (as is customary with simulated annealing). Here we display our custom annealing function.

type `mulprocpermute.m`

```
function schedule = mulprocpermute(optimValues,problemData)
% MULPROCPERMUTE Moves one random task to a different processor.
% NEWX = MULPROCPERMUTE(optimValues,problemData) generate a point based
% on the current point and the current temperature

% Copyright 2006 The MathWorks, Inc.

schedule = optimValues.x;
% This loop will generate a neighbor of "distance" equal to
% optimValues.temperature. It does this by generating a neighbor to the
% current schedule, and then generating a neighbor to that neighbor, and so
% on until it has generated enough neighbors.
for i = 1:floor(optimValues.temperature)+1
    [nrows ncols] = size(schedule);
    schedule = neighbor(schedule, nrows, ncols);
end

%=====
function schedule = neighbor(schedule, nrows, ncols)
% NEIGHBOR generates a single neighbor to the given schedule. It does so
% by moving one random task to a different processor. The rest of the code
% is to ensure that the format of the schedule remains the same.

row1 = randinteger(1,1,nrows)+1;
col = randinteger(1,1,ncols)+1;
while schedule(row1, col)==0
    row1 = randinteger(1,1,nrows)+1;
    col = randinteger(1,1,ncols)+1;
end
row2 = randinteger(1,1,nrows)+1;
while row1==row2
    row2 = randinteger(1,1,nrows)+1;
end
end
```

```

for j = 1:ncols
    if schedule(row2,j)==0
        schedule(row2,j) = schedule(row1,col);
        break
    end
end

schedule(row1, col) = 0;
for j = col:ncols-1
    schedule(row1,j) = schedule(row1,j+1);
end
schedule(row1,ncols) = 0;
%=====
function out = randinteger(m,n,range)
%RANDINTEGER generate integer random numbers (m-by-n) in range

len_range = size(range,1) * size(range,2);
% If the IRANGE is specified as a scalar.
if len_range < 2
    if range < 0
        range = [range+1, 0];
    elseif range > 0
        range = [0, range-1];
    else
        range = [0, 0];    % Special case of zero range.
    end
end
% Make sure RANGE is ordered properly.
range = sort(range);

% Calculate the range the distance for the random number generator.
distance = range(2) - range(1);
% Generate the random numbers.
r = floor(rand(m, n) * (distance+1));

% Offset the numbers to the specified value.
out = ones(m,n)*range(1);
out = out + r;

```

Objective Function

We need an objective function for the multiprocessor scheduling problem. The objective function returns the total time required for a given schedule (which is the maximum of the times that each processor is spending on its tasks). As such, the objective function also needs the lengths matrix to be able to calculate the total times. We are going to attempt to minimize this total time. Here we display our objective function

type `mulprocfitness.m`

```

function timeToComplete = mulprocfitness(schedule, lengths)
%MULPROCFITNESS determines the "fitness" of the given schedule.
% In other words, it tells us how long the given schedule will take using the
% knowledge given by "lengths"

% Copyright 2006 The MathWorks, Inc.

```

```

[nrows ncols] = size(schedule);
timeToComplete = zeros(1,nrows);
for i = 1:nrows
    timeToComplete(i) = 0;
    for j = 1:ncols
        if schedule(i,j)~=0
            timeToComplete(i) = timeToComplete(i)+lengths(i,schedule(i,j));
        else
            break
        end
    end
end
timeToComplete = max(timeToComplete);

```

`simulannealbnd` will call our objective function with just one argument `x`, but our fitness function has two arguments: `x` and "lengths". We can use an anonymous function to capture the values of the additional argument, the lengths matrix. We create a function handle 'ObjectiveFcn' to an anonymous function that takes one input `x`, but calls 'mulprocfitness' with `x` and "lengths". The variable "lengths" has a value when the function handle 'FitnessFcn' is created so these values are captured by the anonymous function.

```

% lengths was defined earlier
fitnessfcn = @(x) mulprocfitness(x,lengths);

```

We can add a custom plot function to plot the length of time that the tasks are taking on each processor. Each bar represents a processor, and the different colored chunks of each bar are the different tasks.

type `mulprocplot.m`

```

function stop = mulprocplot(~,optimvalues,flag,lengths)
%MULPROC PLOT PlotFcn used for SAMULTIPROCESSORDEMO
% STOP = MULPROC PLOT(OPTIONS,OPTIMVALUES,FLAG) where OPTIMVALUES is a
% structure with the following fields:
%     x: current point
%     fval: function value at x
%     bestx: best point found so far
%     bestfval: function value at bestx
%     temperature: current temperature
%     iteration: current iteration
%     funccount: number of function evaluations
%     t0: start time
%     k: annealing parameter 'k'
%
% FLAG: Current state in which PlotFcn is called. Possible values are:
%     init: initialization state
%     iter: iteration state
%     done: final state
%
% STOP: A boolean to stop the algorithm.
%
% Copyright 2006-2015 The MathWorks, Inc.

persistent thisTitle %#ok

stop = false;

```

```

switch flag
    case 'init'
        set(gca,'xlimmode','manual','zlimmode','manual', ...
            'alimmode','manual')
        titleStr = sprintf('Current Point - Iteration %d', optimvalues.iteration);
        thisTitle = title(titleStr,'interp','none');
        topplot = i_generatePlotData(optimvalues, lengths);
        ylabel('Time','interp','none');
        bar(topplot, 'stacked','edgecolor','none');
        Xlength = size(topplot,1);
        set(gca,'xlim',[0,1 + Xlength])
    case 'iter'
        if ~rem(optimvalues.iteration, 100)
            topplot = i_generatePlotData(optimvalues, lengths);
            bar(topplot, 'stacked','edgecolor','none');
            titleStr = sprintf('Current Point - Iteration %d', optimvalues.iteration);
            thisTitle = title(titleStr,'interp','none');
        end
end

function topplot = i_generatePlotData(optimvalues, lengths)

schedule = optimvalues.x;
nrows = size(schedule,1);
% Remove zero columns (all processes are idle)
maxlen = 0;
for i = 1:nrows
    if max(nnz(schedule(i,:))) > maxlen
        maxlen = max(nnz(schedule(i,:)));
    end
end
schedule = schedule(:,1:maxlen);

topplot = zeros(size(schedule));
[nrows, ncols] = size(schedule);
for i = 1:nrows
    for j = 1:ncols
        if schedule(i,j)==0 % idle process
            topplot(i,j) = 0;
        else
            topplot(i,j) = lengths(i,schedule(i,j));
        end
    end
end
end

```

But remember, in simulated annealing the current schedule is not necessarily the best schedule found so far. We create a second custom plot function that will display to us the best schedule that has been discovered so far.

type `mulprocplotbest.m`

```

function stop = mulprocplotbest(~,optimvalues,flag,lengths)
%MULPROCPLTBEST PlotFcn used for SAMULTIPROCESSORDEMO
% STOP = MULPROCPLTBEST(OPTIONS,OPTIMVALUES,FLAG) where OPTIMVALUES is a
% structure with the following fields:
%     x: current point
%     fval: function value at x

```

```

%         bestx: best point found so far
%         bestfval: function value at bestx
%     temperature: current temperature
%     iteration: current iteration
%     funccount: number of function evaluations
%         t0: start time
%         k: annealing parameter 'k'
%
% FLAG: Current state in which PlotFcn is called. Possible values are:
%     init: initialization state
%     iter: iteration state
%     done: final state
%
% STOP: A boolean to stop the algorithm.
%
% Copyright 2006-2015 The MathWorks, Inc.

persistent thisTitle %#ok

stop = false;
switch flag
    case 'init'
        set(gca,'xlimmode','manual','zlimmode','manual', ...
            'alimmode','manual')
        titleStr = sprintf('Current Point - Iteration %d', optimvalues.iteration);
        thisTitle = title(titleStr,'interp','none');
        topplot = i_generatePlotData(optimvalues, lengths);
        Xlength = size(topplot,1);
        ylabel('Time','interp','none');
        bar(topplot, 'stacked','edgecolor','none');
        set(gca,'xlim',[0,1 + Xlength])
    case 'iter'
        if ~rem(optimvalues.iteration, 100)
            topplot = i_generatePlotData(optimvalues, lengths);
            bar(topplot, 'stacked','edgecolor','none');
            titleStr = sprintf('Best Point - Iteration %d', optimvalues.iteration);
            thisTitle = title(titleStr,'interp','none');
        end
end

end

function topplot = i_generatePlotData(optimvalues, lengths)

schedule = optimvalues.bestx;
nrows = size(schedule,1);
% Remove zero columns (all processes are idle)
maxlen = 0;
for i = 1:nrows
    if max(nnz(schedule(i,:))) > maxlen
        maxlen = max(nnz(schedule(i,:)));
    end
end
schedule = schedule(:,1:maxlen);

topplot = zeros(size(schedule));
[nrows, ncols] = size(schedule);
for i = 1:nrows

```

```

    for j = 1:ncols
        if schedule(i,j)==0
            topplot(i,j) = 0;
        else
            topplot(i,j) = lengths(i,schedule(i,j));
        end
    end
end
end

```

Simulated Annealing Options Setup

We choose the custom annealing and plot functions that we have created, as well as change some of the default options. `ReannealInterval` is set to 800 because lower values for `ReannealInterval` seem to raise the temperature when the solver was beginning to make a lot of local progress. We also decrease the `StallIterLimit` to 800 because the default value makes the solver too slow. Finally, we must set the `DataType` to 'custom'.

```

options = optimoptions(@simulannealbnd,'DataType', 'custom', ...
    'AnnealingFcn', @mulprocpermute, 'MaxStallIterations',800, 'ReannealInterval', 800, ...
    'PlotFcn', {@mulprocplot, lengths},{@mulprocplotbest, lengths},@saplotf,@saplotbestf});

```

Finally, we call simulated annealing with our problem information.

```

schedule = simulannealbnd(fitnessfcn,sampleSchedule,[],[],options);
% Remove zero columns (all processes are idle)
maxlen = 0;
for i = 1:size(schedule,1)
    if max(nnz(schedule(i,:)))>maxlen
        maxlen = max(nnz(schedule(i,:)));
    end
end
% Display the schedule
schedule = schedule(:,1:maxlen)

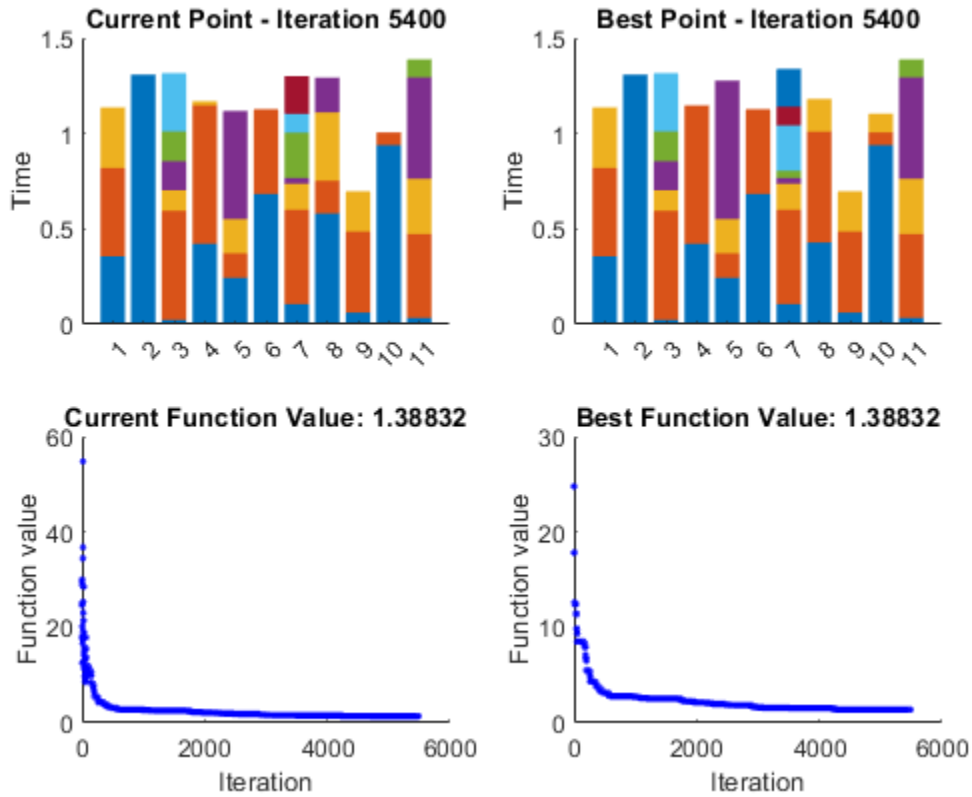
```

Optimization terminated: change in best function value less than options.FunctionTolerance.

```

schedule =
    22    34    32     0     0     0     0     0
     5     0     0     0     0     0     0     0
    19     6    12    11    39    35     0     0
     7    20     0     0     0     0     0     0
    30    15    10     3     0     0     0     0
    18    28     0     0     0     0     0     0
    31    33    29     4    21     9    25    40
    24    26    14     0     0     0     0     0
    13    16    23     0     0     0     0     0
    38    36     1     0     0     0     0     0
     8    27    37    17     2     0     0     0

```



See Also

`simulannealbnd`

More About

- “Algorithm Settings” on page 17-60
- “How Simulated Annealing Works” on page 13-14
- “Minimize Makespan in Parallel Processing”

Multiobjective Optimization

- “What Is Multiobjective Optimization?” on page 14-2
- “gamultiobj Algorithm” on page 14-5
- “paretosearch Algorithm” on page 14-10
- “gamultiobj Options and Syntax: Differences from ga” on page 14-18
- “Pareto Front for Two Objectives” on page 14-19
- “Compare paretosearch and gamultiobj” on page 14-27
- “Plot 3-D Pareto Front” on page 14-38
- “Performing a Multiobjective Optimization Using the Genetic Algorithm” on page 14-48
- “Effects of Multiobjective Genetic Algorithm Options” on page 14-53
- “Design Optimization of a Welded Beam” on page 14-62

What Is Multiobjective Optimization?

You might need to formulate problems with more than one objective, since a single objective with several constraints may not adequately represent the problem being faced. If so, there is a vector of objectives,

$$F(x) = [F_1(x), F_2(x), \dots, F_m(x)], \quad (14-1)$$

that must be traded off in some way. The relative importance of these objectives is not generally known until the system's best capabilities are determined and tradeoffs between the objectives fully understood. As the number of objectives increases, tradeoffs are likely to become complex and less easily quantified. The designer must rely on his or her intuition and ability to express preferences throughout the optimization cycle. Thus, requirements for a multiobjective design strategy must enable a natural problem formulation to be expressed, and be able to solve the problem and enter preferences into a numerically tractable and realistic design problem.

Multiobjective optimization is concerned with the minimization of a vector of objectives $F(x)$ that can be the subject of a number of constraints or bounds:

$$\begin{aligned} & \min F(x), \text{ subject to} \\ & x \in \mathbb{R}^n \\ & G_i(x) = 0, \quad i = 1, \dots, k_e; \quad G_i(x) \leq 0, \quad i = k_e + 1, \dots, k; \quad l \leq x \leq u. \end{aligned}$$

Note that because $F(x)$ is a vector, if any of the components of $F(x)$ are competing, there is no unique solution to this problem. Instead, the concept of noninferiority in Zadeh [4] (also called Pareto optimality in Censor [1] and Da Cunha and Polak [2]) must be used to characterize the objectives. A noninferior solution is one in which an improvement in one objective requires a degradation of another. To define this concept more precisely, consider a feasible region, Ω , in the parameter space. x is an element of the n -dimensional real numbers $x \in \mathbb{R}^n$ that satisfies all the constraints, that is,

$$\Omega = \{x \in \mathbb{R}^n\},$$

subject to

$$\begin{aligned} & G_i(x) = 0, \quad i = 1, \dots, k_e, \\ & G_i(x) \leq 0, \quad i = k_e + 1, \dots, k, \\ & l \leq x \leq u. \end{aligned}$$

This allows definition of the corresponding feasible region for the objective function space Λ :

$$\Lambda = \{y \in \mathbb{R}^m : y = F(x), x \in \Omega\}.$$

The performance vector $F(x)$ maps parameter space into objective function space, as represented in two dimensions in the figure "Figure 14-1, Mapping from Parameter Space into Objective Function Space" on page 14-3.

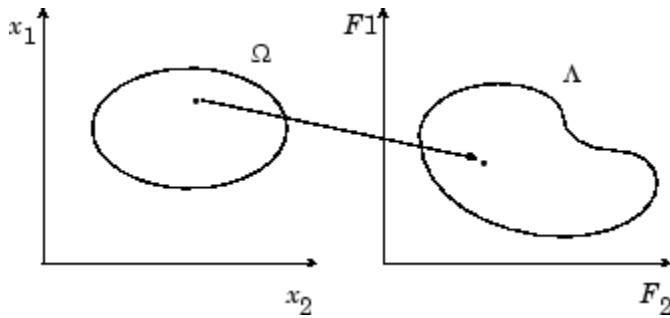


Figure 14-1, Mapping from Parameter Space into Objective Function Space

A noninferior solution point can now be defined.

Definition: Point $x^* \in \Omega$ is a noninferior solution if for some neighborhood of x^* there does not exist a Δx such that $(x^* + \Delta x) \in \Omega$ and

$$F_i(x^* + \Delta x) \leq F_i(x^*), \quad i = 1, \dots, m, \text{ and}$$

$$F_j(x^* + \Delta x) < F_j(x^*) \text{ for at least one } j.$$

In the two-dimensional representation of the figure “Figure 14-2, Set of Noninferior Solutions” on page 14-3, the set of noninferior solutions lies on the curve between C and D . Points A and B represent specific noninferior points.

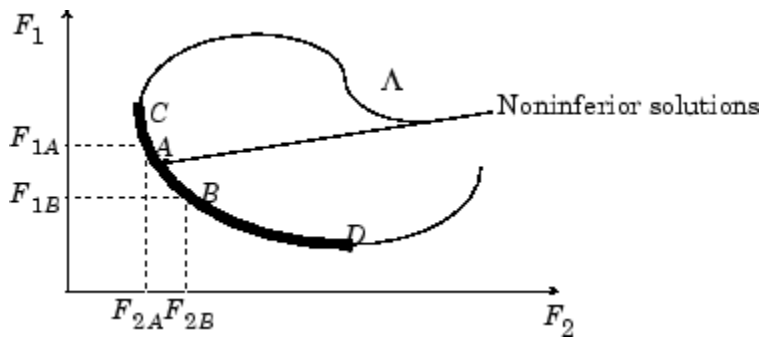


Figure 14-2, Set of Noninferior Solutions

A and B are clearly noninferior solution points because an improvement in one objective, F_1 , requires a degradation in the other objective, F_2 , that is, $F_{1B} < F_{1A}$, $F_{2B} > F_{2A}$.

Since any point in Ω that is an inferior point represents a point in which improvement can be attained in all the objectives, it is clear that such a point is of no value. Multiobjective optimization is, therefore, concerned with the generation and selection of noninferior solution points.

Noninferior solutions are also called Pareto optima. A general goal in multiobjective optimization is constructing the Pareto optima.

See Also

More About

- “gamultiobj Algorithm” on page 14-5
- “paretosearch Algorithm” on page 14-10
- “Pareto Front for Two Objectives” on page 14-19

gamultiobj Algorithm

In this section...

"Introduction" on page 14-5

"Multiobjective Terminology" on page 14-5

"Initialization" on page 14-7

"Iterations" on page 14-7

"Stopping Conditions" on page 14-7

"Integer and Linear Constraints" on page 14-8

"Bibliography" on page 14-8

Introduction

This section describes the algorithm that `gamultiobj` uses to create a set of points on the Pareto front. `gamultiobj` uses a controlled, elitist genetic algorithm (a variant of NSGA-II [3]). An elitist GA always favors individuals with better fitness value (rank). A controlled elitist GA also favors individuals that can help increase the diversity of the population even if they have a lower fitness value.

Multiobjective Terminology

Most of the terminology for the `gamultiobj` algorithm is the same as "Genetic Algorithm Terminology" on page 8-13. However, there are some additional terms, described in this section. For more details about the terminology and the algorithm, see Deb [3].

- **Dominance** — A point x dominates a point y for a vector-valued objective function f when:

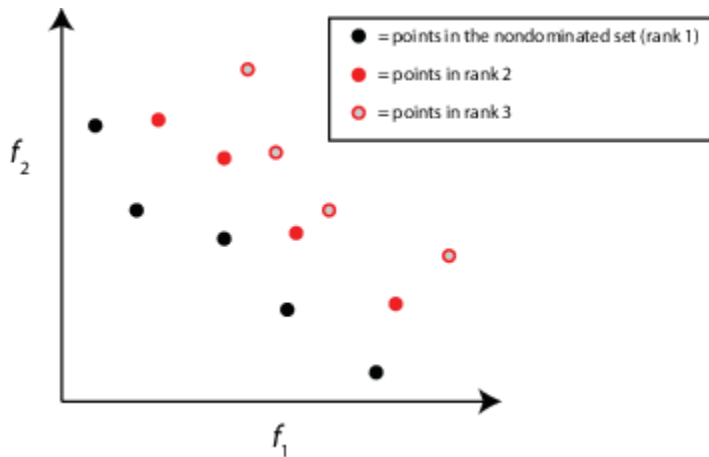
$$f_i(x) \leq f_i(y) \text{ for all } i.$$

$$f_j(x) < f_j(y) \text{ for some } j.$$

The term "dominate" is equivalent to the term "inferior:" x dominates y exactly when y is inferior to x .

A nondominated set among a set of points P is the set of points Q in P that are not dominated by any point in P .

- **Rank** — For feasible individuals, there is an iterative definition of the rank of an individual. Rank 1 individuals are not dominated by any other individuals. Rank 2 individuals are dominated only by rank 1 individuals. In general, rank k individuals are dominated only by individuals in rank $k - 1$ or lower.



Individuals with a lower rank have a higher chance of selection (lower rank is better).

All infeasible individuals have a worse rank than any feasible individual. Within the infeasible population, the rank is the order by sorted infeasibility measure, plus the highest rank for feasible members.

`gamultiobj` uses rank to select parents.

- *Crowding Distance* — The crowding distance is a measure of the closeness of an individual to its nearest neighbors. The `gamultiobj` algorithm measures distance among individuals of the same rank. By default, the algorithm measures distance in objective function space. However, you can measure the distance in decision variable space (also termed design variable space) by setting the `DistanceMeasureFcn` option to `{@distancecrowding, 'genotype'}`.

The algorithm sets the distance of individuals at the extreme positions to `Inf`. For the remaining individuals, the algorithm calculates distance as a sum over the dimensions of the normalized absolute distances between the individual's sorted neighbors. In other words, for dimension m and sorted, scaled individual i :

$$\text{distance}(i) = \sum_m (x(m, i+1) - x(m, i-1)).$$

The algorithm sorts each dimension separately, so the term neighbors means neighbors in each dimension.

Individuals of the same rank with a higher distance have a higher chance of selection (higher distance is better).

You can choose a different crowding distance measure than the default `@distancecrowding` function. See “Multiobjective Options” on page 17-38.

Crowding distance is one factor in the calculation of the spread, which is part of a stopping criterion. Crowding distance is also used as a tie-breaker in tournament selection, when two selected individuals have the same rank.

- *Spread* — The spread is a measure of the movement of the Pareto set. To calculate the spread, the `gamultiobj` algorithm first evaluates σ , the standard deviation of the crowding distance measure of points that are on the Pareto front with finite distance. Q is the number of these points, and d is the average distance measure among these points. The algorithm then evaluates μ , the sum over the k objective function indices of the norm of the difference between the current minimum-value Pareto point for that index and the minimum point for that index in the previous iteration. The spread is then

$$\text{spread} = (\mu + \sigma)/(\mu + Qd).$$

The spread is small when the extreme objective function values do not change much between iterations (that is, μ is small) and when the points on the Pareto front are spread evenly (that is, σ is small).

`gamultiobj` uses the spread in a stopping condition. Iterations halt when the spread does not change much, and the final spread is less than an average of recent spreads. See “Stopping Conditions” on page 14-7.

Initialization

The first step in the `gamultiobj` algorithm is creating an initial population. The algorithm creates the population, or you can give an initial population or a partial initial population by using the `InitialPopulationMatrix` option (see “Population Options” on page 17-26). The number of individuals in the population is set to the value of the `PopulationSize` option. By default, `gamultiobj` creates a population that is feasible with respect to bounds and linear constraints, but is not necessarily feasible with respect to nonlinear constraints. The default creation algorithm is `@gacreationuniform` when there are no constraints or only bound constraints, and `@gacreationlinearfeasible` when there are linear or nonlinear constraints.

`gamultiobj` evaluates the objective function and constraints for the population, and uses those values to create scores for the population.

Iterations

The main iteration of the `gamultiobj` algorithm proceeds as follows.

- 1 Select parents for the next generation using the selection function on the current population. The only built-in selection function available for `gamultiobj` is binary tournament. You can also use a custom selection function.
- 2 Create children from the selected parents by mutation and crossover.
- 3 Score the children by calculating their objective function values and feasibility.
- 4 Combine the current population and the children into one matrix, the extended population.
- 5 Compute the rank and crowding distance for all individuals in the extended population.
- 6 Trim the extended population to have `PopulationSize` individuals by retaining the appropriate number of individuals of each rank.

When the problem has integer or linear constraints (including bounds), the algorithm modifies the evolution of the population. See “Integer and Linear Constraints” on page 14-8.

Stopping Conditions

The following stopping conditions apply. Each stopping condition is associated with an exit flag.

exitflag Value	Stopping Condition
1	Geometric average of the relative change in value of the spread over <code>options.MaxStallGenerations</code> generations is less than <code>options.FunctionTolerance</code> , and the final spread is less than the mean spread over the past <code>options.MaxStallGenerations</code> generations
0	Maximum number of generations exceeded
-1	Optimization terminated by an output function or plot function
-2	No feasible point found
-5	Time limit exceeded

For exit flag 1, the geometric average of the relative change in spread has multiplier $\frac{1}{2}^k$ for the relative change in the k th previous generation.

Integer and Linear Constraints

When a problem has integer or linear constraints (including bounds), the algorithm modifies the evolution of the population.

- When the problem has both integer and linear constraints, the software modifies all generated individuals to be feasible with respect to those constraints. You can use any creation, mutation, or crossover function, and the entire population remains feasible with respect to integer and linear constraints.
- When the problem has only linear constraints, the software does not modify the individuals to be feasible with respect to those constraints. You must use creation, mutation, and crossover functions that maintain feasibility with respect to linear constraints. Otherwise, the population can become infeasible, and the result can be infeasible. The default operators maintain linear feasibility: `gacreationlinearfeasible` or `gacreationnonlinearfeasible` for creation, `mutationadaptfeasible` for mutation, and `crossoverintermediate` for crossover.

The internal algorithms for integer and linear feasibility are similar to those for `surrogateopt`. When a problem has integer and linear constraints, the algorithm first creates linearly feasible points. Then the algorithm tries to satisfy integer constraints by rounding linearly feasible points to integers using a heuristic that attempts to keep the points linearly feasible. When this process is unsuccessful in obtaining enough feasible points for constructing a population, the algorithm calls `intlinprog` to try to find more points that are feasible with respect to bounds, linear constraints, and integer constraints.

Later, when mutation or crossover creates new population members, the algorithms ensure that the new members are integer and linear feasible by taking similar steps. Each new member is modified, if necessary, to be as close as possible to its original value, while also satisfying the integer and linear constraints and bounds.

Bibliography

- [1] Censor, Y. "Pareto Optimality in Multiobjective Problems," *Appl. Math. Optimiz.*, Vol. 4, pp 41-59, 1977.
- [2] Da Cunha, N. O. and E. Polak. "Constrained Minimization Under Vector-Valued Criteria in Finite Dimensional Spaces," *J. Math. Anal. Appl.*, Vol. 19, pp 103-124, 1967.

- [3] Deb, Kalyanmoy. "Multi-Objective Optimization using Evolutionary Algorithms," John Wiley & Sons, Ltd, Chichester, England, 2001.
- [4] Zadeh, L. A. "Optimality and Nonscalar-Valued Performance Criteria," *IEEE Trans. Automat. Contr.*, Vol. AC-8, p. 1, 1963.

See Also

gamultiobj

More About

- "What Is Multiobjective Optimization?" on page 14-2
- "Genetic Algorithm Options" on page 17-23
- "gamultiobj Options and Syntax: Differences from ga" on page 14-18

paretosearch Algorithm

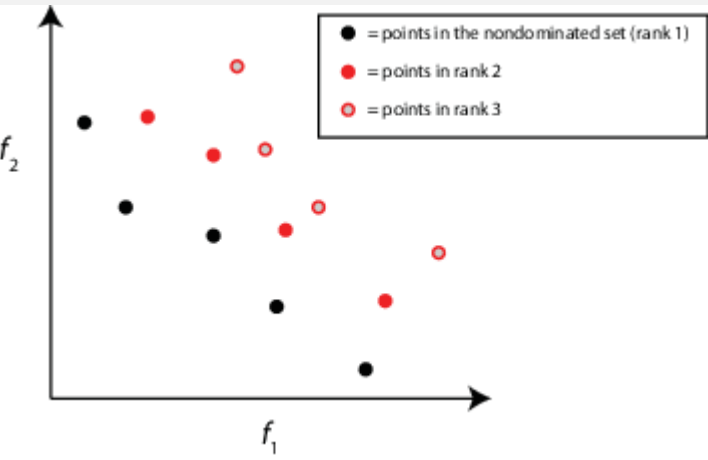
paretosearch Algorithm Overview

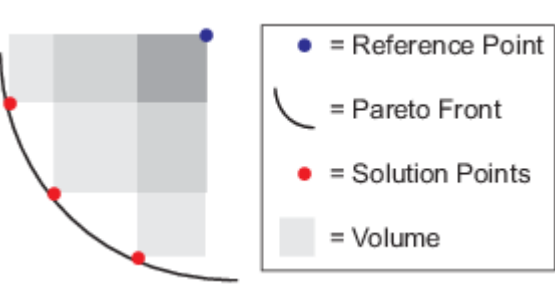
The paretosearch algorithm uses pattern search on a set of points to search iteratively for nondominated points. See “Multiobjective Terminology” on page 14-5. The pattern search satisfies all bounds and linear constraints at each iteration.

Theoretically, the algorithm converges to points near the true Pareto front. For a discussion and proof of convergence, see Custòdio et al. [1], whose proof applies to problems with Lipschitz continuous objectives and constraints.

Definitions for paretosearch Algorithm

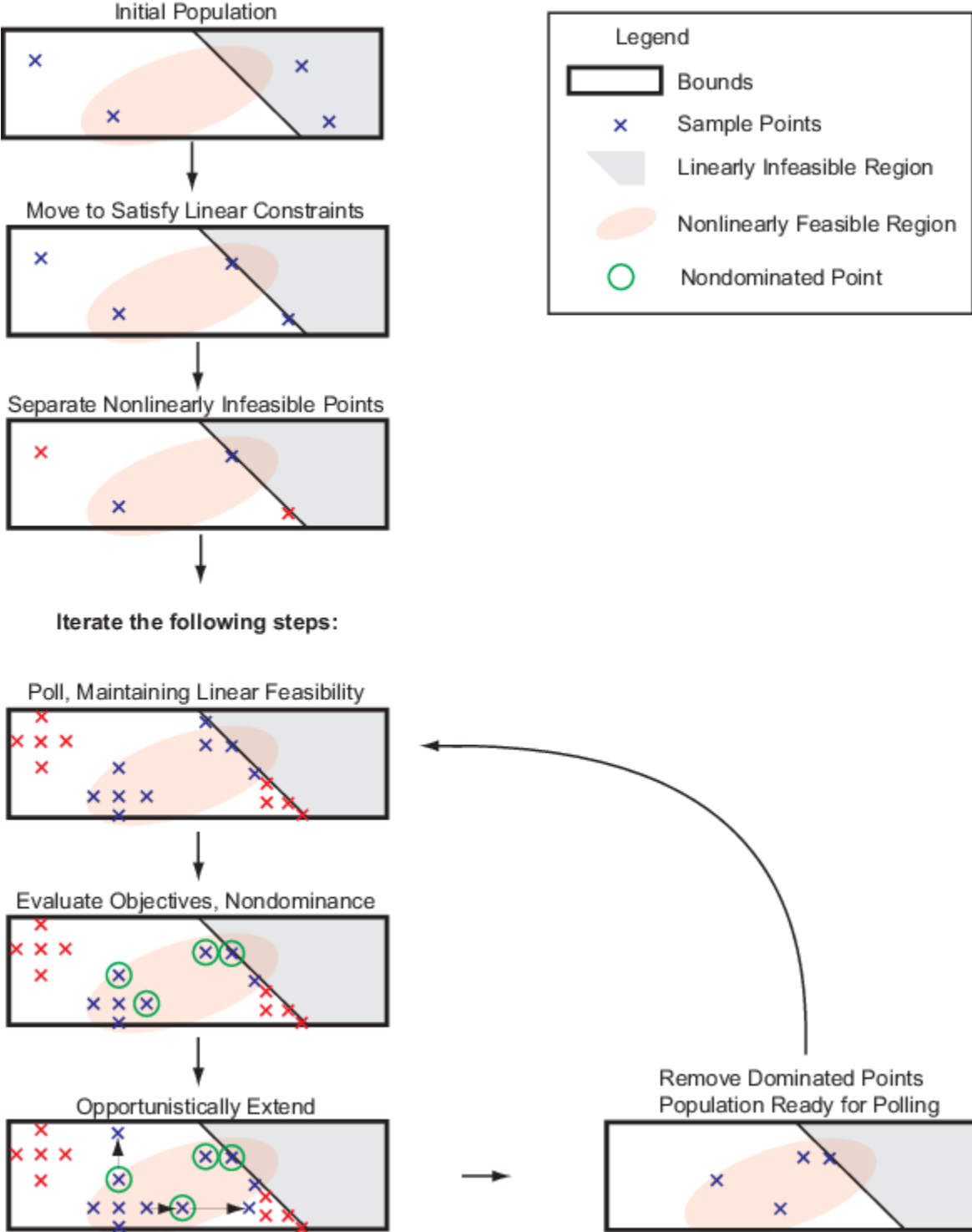
paretosearch uses a number of intermediate quantities and tolerances in its algorithm.

Quantity	Definition
Rank	<p>The rank of a point has an iterative definition.</p> <ul style="list-style-type: none"> • Nondominated points have rank 1. • For any integer $k > 1$, a point has rank k when the only points that dominate it have rank strictly less than k. 

Quantity	Definition
Volume	<p>Hypervolume of the set of points \mathbf{p} in objective function space that satisfy the inequality, for every index j,</p> $f_i(j) < \mathbf{p}_i < M_i,$ <p>where $f_i(j)$ is the ith component of the jth objective function value in the Pareto set, and M_i is an upper bound for the ith component for all points in the Pareto set. In this figure, M is called the Reference Point. The shades of gray in the figure denote portions of the volume that some calculation algorithms use as part of an inclusion-exclusion calculation.</p>  <p>For details, see Fleischer [3].</p> <p>paretosearch calculates the volume only when the number of nondominated points exceeds the number of objectives. paretosearch uses the reference point $M = \max(\text{pts}, [], 1) + 1$. Here, pts is a matrix whose rows are the points.</p> <p>Volume change is one factor in stopping the algorithm. For details, see “Stopping Conditions” on page 14-15.</p>
Distance	<p>Distance is a measure of the closeness of an individual to its nearest neighbors. The paretosearch algorithm measures distance among individuals of the same rank. The algorithm measures distance in objective function space.</p> <p>The algorithm sets the distance of individuals at the extreme positions to Inf. For the remaining individuals, the algorithm calculates distance as a sum over the dimensions of the normalized absolute distances between the individual's sorted neighbors. In other words, for dimension m and sorted, scaled individual i:</p> $\text{distance}(i) = \sum_m (x(m, i+1) - x(m, i-1)).$ <p>The algorithm sorts each dimension separately, so the term neighbors means neighbors in each dimension.</p> <p>Individuals of the same rank with a higher distance have a higher chance of selection (higher distance is better).</p> <p>Distance is one factor in the calculation of the spread, which is part of a stopping criterion. For details, see “Stopping Conditions” on page 14-15.</p>

Quantity	Definition
Spread	<p>Spread is a measure of the movement of the Pareto set. To calculate the spread, the <code>paretosearch</code> algorithm first evaluates σ, the standard deviation of the crowding distance measure of points on the Pareto front with finite distance. Q is the number of these points, and d is the average distance measure among these points. The algorithm then evaluates μ, the sum over the k objective function indices of the norm of the difference between the current minimum-value Pareto point for that index and the minimum point for that index in the previous iteration. The spread is then</p> $\text{spread} = (\mu + \sigma)/(\mu + Qd).$ <p>The spread is small when the extreme objective function values do not change much between iterations (that is, μ is small) and when the points on the Pareto front are spread evenly (that is, σ is small).</p> <p><code>paretosearch</code> uses the spread in a stopping condition. Iterations halt when the spread does not change much. For details, see “Stopping Conditions” on page 14-15.</p>
ParetoSetChangeTolerance	<p>Stopping condition for the search. <code>paretosearch</code> stops when the volume, spread, or distance does not change by more than <code>ParetoSetChangeTolerance</code> over a window of iterations. For details, see “Stopping Conditions” on page 14-15.</p>
MinPollFraction	<p>Minimum fraction of locations to poll during an iteration. <code>paretosearch</code> polls at least <code>MinPollFraction</code>*(number of points in pattern) locations. If the number of polled points gives a nondominated point, the poll is considered a success. Otherwise, <code>paretosearch</code> continues to poll until it either finds a nondominated point or runs out of points in the pattern.</p> <p>This option does not apply when the <code>UseVectorized</code> option is <code>true</code>. In that case, <code>paretosearch</code> polls all pattern points.</p>

Sketch of paretosearch Algorithm



Initialize Search

To create the initial set of points, `paretosearch` generates `options.ParetoSetSize` points from a quasirandom sample based on the problem bounds, by default. For details, see Bratley and Fox [2]. When the problem has over 500 dimensions, `paretosearch` uses Latin hypercube sampling to generate the initial points.

If a component has no bounds, `paretosearch` uses an artificial lower bound of -10 and an artificial upper bound of 10 .

If a component has only one bound, `paretosearch` uses that bound as an endpoint of an interval of width $20 + 2 \cdot \text{abs}(\text{bound})$. For example, if there is no upper bound for a component and there is a lower bound of 15 , `paretosearch` uses an interval width of $20 + 2 \cdot 15 = 55$, so uses an artificial upper bound of $15 + 55 = 70$.

If you pass some initial points in `options.InitialPoints`, then `paretosearch` uses those points as the initial points. `paretosearch` generates more points, if necessary, to obtain at least `options.ParetoSetSize` initial points.

`paretosearch` then checks the initial points to ensure that they are feasible with respect to the bounds and linear constraints. If necessary, `paretosearch` projects the initial points onto the linear subspace of linearly feasible points by solving a linear programming problem. This process can cause some points to coincide, in which case `paretosearch` removes any duplicate points. `paretosearch` does not alter initial points for artificial bounds, only for specified bounds and linear constraints.

After moving the points to satisfy linear constraints, if necessary, `paretosearch` checks whether the points satisfy the nonlinear constraints. `paretosearch` gives a penalty value of `Inf` to any point that does not satisfy all nonlinear constraints. Then `paretosearch` calculates any missing objective function values of the remaining feasible points.

Note Currently, `paretosearch` does not support nonlinear equality constraints $\text{ceq}(x) = 0$.

Create Archive and Incumbents

`paretosearch` maintains two sets of points:

- `archive` — A structure that contains nondominated points associated with a mesh size below `options.MeshTolerance` and satisfying all constraints to within `options.ConstraintTolerance`. The `archive` structure contains no more than $2 \cdot \text{options.ParetoSetSize}$ points and is initially empty. Each point in `archive` contains an associated mesh size, which is the mesh size at which the point was generated.
- `iterates` — A structure containing nondominated points and possibly some dominated points associated with larger mesh sizes or infeasibility. Each point in `iterates` contains an associated mesh size. `iterates` contains no more than `options.ParetoSetSize` points.

Poll to Find Better Points

`paretosearch` polls points from `iterates`, with the polled points inheriting the associated mesh size from the point in `iterates`. The `paretosearch` algorithm uses a poll that maintains feasibility with respect to bounds and all linear constraints.

If the problem has nonlinear constraints, `paretosearch` computes the feasibility of each poll point. `paretosearch` keeps the score of infeasible points separately from the score of feasible points. The score of a feasible point is the vector of objective function values of that point. The score of an infeasible point is the sum of the nonlinear infeasibilities.

`paretosearch` polls at least `MinPollFraction*(number of points in pattern)` locations for each point in `iterates`. If the polled points give at least one nondominated point with respect to the incumbent (original) point, the poll is considered a success. Otherwise, `paretosearch` continues to poll until it either finds a nondominated point or runs out of points in the pattern. If `paretosearch` runs out of points and does not produce a nondominated point, `paretosearch` declares the poll unsuccessful and halves the mesh size.

If the poll finds nondominated points, `paretosearch` extends the poll in the successful directions repeatedly, doubling the mesh size each time, until the extension produces a dominated point. During this extension, if the mesh size exceeds `options.MaxMeshSize` (default value: `Inf`), the poll stops. If the objective function values decrease to `-Inf`, `paretosearch` declares the problem unbounded and stops.

Update archive and iterates Structures

After polling all the points in `iterates`, the algorithm examines the new points together with the points in the `iterates` and archive structures. `paretosearch` computes the rank, or Pareto front number, of each point and then does the following.

- Mark for removal all points that do not have rank 1 in archive.
- Mark new rank 1 points for insertion into `iterates`.
- Mark feasible points in `iterates` whose associated mesh size is less than `options.MeshTolerance` for transfer to archive.
- Mark dominated points in `iterates` for removal only if they prevent new nondominated points from being added to `iterates`.

`paretosearch` then computes the volume and distance measures for each point. If archive will overflow as a result of marked points being included, then the points with the largest volume occupy archive, and the others leave. Similarly, the new points marked for addition to `iterates` enter `iterates` in order of their volumes.

If `iterates` is full and has no dominated points, then `paretosearch` adds no points to `iterates` and declares the iteration to be unsuccessful. `paretosearch` multiplies the mesh sizes in `iterates` by 1/2.

Stopping Conditions

For three or fewer objective functions, `paretosearch` uses volume and spread as stopping measures. For four or more objectives, `paretosearch` uses distance and spread as stopping measures. In the remainder of this discussion, the two measures that `paretosearch` uses are denoted the applicable measures.

The algorithm maintains vectors of the last eight values of the applicable measures. After eight iterations, the algorithm checks the values of the two applicable measures at the beginning of each iteration, where `tol = options.ParetoSetChangeTolerance`:

- `spreadConverged = abs(spread(end - 1) - spread(end)) <= tol*max(1,spread(end - 1));`
- `volumeConverged = abs(volume(end - 1) - volume(end)) <= tol*max(1,volume(end - 1));`
- `distanceConverged = abs(distance(end - 1) - distance(end)) <= tol*max(1,distance(end - 1));`

If either applicable test is `true`, the algorithm stops. Otherwise, the algorithm computes the max of squared terms of the Fourier transforms of the applicable measures minus the first term. The algorithm then compares the maxima to their deleted terms (the DC components of the transforms). If either deleted term is larger than $100 * tol * (\text{max of all other terms})$, then the algorithm stops. This test essentially determines that the sequence of measures is not fluctuating, and therefore has converged.

Additionally, a plot function or output function can stop the algorithm, or the algorithm can stop because it exceeds a time limit or function evaluation limit.

Returned Values

The algorithm returns the points on the Pareto front as follows.

- `paretosearch` combines the points in `archive` and `iterates` into one set.
- When there are three or fewer objective functions, `paretosearch` returns the points from the largest volume to the smallest, up to at most `ParetoSetSize` points.
- When there are four or more objective functions, `paretosearch` returns the points from the largest distance to the smallest, up to at most `ParetoSetSize` points.

Modifications for Parallel Computation and Vectorized Function Evaluation

When `paretosearch` computes objective function values in parallel or in a vectorized fashion (`UseParallel` is `true` or `UseVectorized` is `true`), there are some changes to the algorithm.

- When `UseVectorized` is `true`, `paretosearch` ignores the `MinPollFraction` option and evaluates all poll points in the pattern.
- When computing in parallel, `paretosearch` sequentially examines each point in `iterates` and performs a parallel poll from each point. After returning `MinPollFraction` fraction of the poll points, `paretosearch` determines if any poll points dominate the base point. If so, the poll is deemed successful, and any other parallel evaluations halt. If not, polling continues until a dominating point appears or the poll is done.
- `paretosearch` performs objective function evaluations either on workers or in a vectorized fashion, but not both. If you set both `UseParallel` and `UseVectorized` to `true`, `paretosearch` calculates objective function values in parallel on workers, but not in a vectorized fashion. In this case, `paretosearch` ignores the `MinPollFraction` option and evaluates all poll points in the pattern.

Run paretosearch Quickly

The fastest way to run `paretosearch` depends on several factors.

- If objective function evaluations are slow, then it is usually fastest to use parallel computing. The overhead in parallel computing can be substantial when objective function evaluations are fast, but when they are slow, it is usually best to use more computing power.

Note Parallel computing requires a Parallel Computing Toolbox license.

- If objective function evaluations are not very time consuming, then it is usually fastest to use vectorized evaluation. However, this is not always the case, because vectorized computations evaluate an entire pattern, whereas serial evaluations can take just a small fraction of a pattern. In high dimensions especially, this reduction in evaluations can cause serial evaluation to be faster for some problems.
- To use vectorized computing, your objective function must accept a matrix with an arbitrary number of rows. Each row represents one point to evaluate. The objective function must return a matrix of objective function values with the same number of rows as it accepts, with one column for each objective function. For a single-objective discussion, see “Vectorize the Fitness Function” on page 8-103 (ga) or “Vectorized Objective Function” on page 6-83 (patternsearch).

References

- [1] Custódio, A. L., J. F. A. Madeira, A. I. F. Vaz, and L. N. Vicente. *Direct Multisearch for Multiobjective Optimization*. SIAM J. Optim., 21(3), 2011, pp. 1109–1140. Preprint available at <https://estudogeral.sib.uc.pt/bitstream/10316/13698/1/Direct%20multisearch%20for%20multiobjective%20optimization.pdf>.
- [2] Bratley, P., and B. L. Fox. *Algorithm 659: Implementing Sobol’s quasirandom sequence generator*. ACM Trans. Math. Software 14, 1988, pp. 88–100.
- [3] Fleischer, M. *The Measure of Pareto Optima: Applications to Multi-Objective Metaheuristics*. In “Proceedings of the Second International Conference on Evolutionary Multi-Criterion Optimization—EMO” April 2003 in Faro, Portugal. Published by Springer-Verlag in the Lecture Notes in Computer Science series, Vol. 2632, pp. 519–533. Preprint available at <https://apps.dtic.mil/dtic/tr/fulltext/u2/a441037.pdf>.

See Also

paretosearch

More About

- “Multiobjective Optimization”

gamultiobj Options and Syntax: Differences from ga

The syntax and options for `gamultiobj` are similar to those for `ga`, with the following differences:

- `gamultiobj` uses only the 'penalty' algorithm for nonlinear constraints. See “Nonlinear Constraint Solver Algorithms for Genetic Algorithm” on page 8-56.
- `gamultiobj` takes an option `DistanceMeasureFcn`, a function that assigns a distance measure to each individual with respect to its neighbors.
- `gamultiobj` takes an option `ParetoFraction`, a number between 0 and 1 that specifies the fraction of the population on the best Pareto frontier to be kept during the optimization. (If there are too few individuals of other ranks in step 6 of “Iterations” on page 14-7, then the fraction of the population on the best Pareto frontier can exceed `ParetoFraction`.)
- `gamultiobj` uses only the Tournament selection function.
- `gamultiobj` uses elite individuals differently than `ga`. It sorts noninferior individuals above inferior ones, so it uses elite individuals automatically.
- `gamultiobj` has only one hybrid function, `fgoalattain`.
- `gamultiobj` does not have a stall time limit.
- `gamultiobj` has different plot functions available.
- `gamultiobj` does not have a choice of scaling function.

See Also

More About

- “What Is Multiobjective Optimization?” on page 14-2
- “gamultiobj Algorithm” on page 14-5
- “Genetic Algorithm Options” on page 17-23

Pareto Front for Two Objectives

In this section...

“Multiobjective Optimization with Two Objectives” on page 14-19

“Find Pareto Set Using Optimize Live Editor Task” on page 14-19

“Find Pareto Set at the Command Line” on page 14-24

“Alternate Views” on page 14-25

Multiobjective Optimization with Two Objectives

This example shows how to find a Pareto set for a two-objective function of two variables. The example presents two approaches for minimizing: using the **Optimize** Live Editor task and working at the command line.

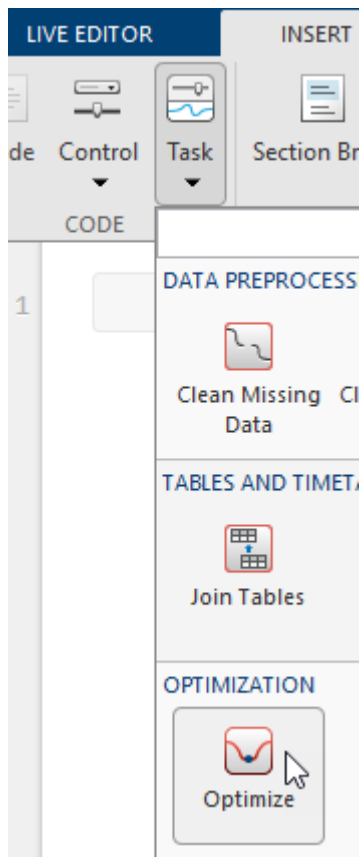
The two-objective function $f(x)$, where x is also two-dimensional, is

$$f_1(x) = x_1^4 + x_2^4 + x_1x_2 - (x_1 - 10x_1^2)$$

$$f_2(x) = x_1^4 + x_2^4 + x_1x_2 - (x_1).$$

Find Pareto Set Using Optimize Live Editor Task

- 1 Create a new live script by clicking the **New Live Script** button in the **File** section on the **Home** tab.
- 2 Insert an **Optimize** Live Editor task. Click the **Insert** tab and then, in the **Code** section, select **Task > Optimize**.



Optimize

Solve an optimization problem or system of equations

Select approach

Problem-based (recommended)

- Easier to define problem
- Represents problem inputs symbolically
- Built-in automatic differentiation

Solver-based

- Start with a solver
- Represents inputs as matrices/functions
- Allows specialized solution methods


3 Click the **Solver-based** task.

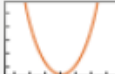
Optimize

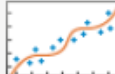
Minimize a function with or without constraints


Specify problem type


Objective


 Linear


 Quadratic



 Least squares



 Nonlinear



 Nonsmooth


Select an objective type to see example functions


Constraints


 Unconstrained


 Lower bounds


 Upper bounds

 Linear inequality

 Linear equality

 Second-order cone

 Nonlinear

 Integer

Select constraint types to see example formulas

Solver fmincon - Constrained nonlinear minimization (recommended) ?

Select problem data

Objective function From file ▼ Browse... New... ?

Initial point (x0) select ▼

Specify solver options

Display progress

Text display Final output ▼

Plot

Current point

Evaluation count

Objective value and feasibility

Objective value

Max constraint violation

Step size

Optimality measure

- 4 For use in entering problem data, insert a new section by clicking the **Section Break** button on the **Insert** tab. New sections appear above and below the task.
- 5 In the new section above the task, enter the following code to define the number of variables and lower and upper bounds.

```
nvar = 2;
lb = [0 -5];
ub = [5 0];
```

- 6 To place these variables into the workspace, run the section by pressing **Ctrl+Enter**.
- 7 **Specify Problem Type**

In the **Specify problem type** section of the task, click the **Objective > Nonlinear** button.

- 8 Click the **Constraints > Lower bounds** and **Upper bounds** buttons.
- 9 Select **Solver > gamultiobj - Multiobjective optimization using genetic algorithm**.

Specify problem type

Objective

Linear Quadratic Least squares **Nonlinear** Nonsmooth

Examples: $f(x, y) = x/y$, $f(x) = \cos(x)$, $f(x) = \log(x)$, $f(x) = e^x$, $f(x) = x^3$, Solve $F(x) = 0$, ...

Unconstrained Lower bounds Upper bounds Linear inequality

Constraints Linear equality Second-order cone Nonlinear Integer

Examples: $x \geq 0$, $x \leq 2$

Solver gamultiobj - Multiobjective optimization using genetic algorithm

10 Select Problem Data

In the **Select problem data** section, select **Objective function > Local function**, and then click the **New** button. The function appears in a new section below the task.

11 Edit the resulting function definition to contain the following code.

```
function f = mymulti1(x)

f(2) = x(1)^4 + x(2)^4 + x(1)*x(2) - (x(1)*x(2))^2;
f(1) = f(2) - 10*x(1)^2;
end
```

12 In the **Select problem data** section, select the **Local function > mymulti1** function.

13 Select **Number of variables > nvar**.

14 Select **Lower bounds > From workspace > lb** and **Upper bounds > From workspace > ub**.

15 **Specify Solver Options**

Expand the **Specify solver options** section of the task, and then click the **Add** button. To have a denser, more connected Pareto front, specify a larger-than-default populations by selecting **Population settings > Population size > 60**.

16 To have more of the population on the Pareto front than the default settings, click the + button. In the resulting options, select **Algorithm > Pareto set fraction > 0.7**.

17 **Set Display Options**

In the **Display progress** section of the task, select the **Pareto front** plot function.

Solver gamultiobj - Multiobjective optimization using genetic algorithm ?

Select problem data

Objective function Local function mymulti1 New... ?

Number of variables nvar

Constraints Lower bounds From workspace lb $\leq x$

Upper bounds From workspace ub $\geq x$

Specify solver options

? Population settings Population size 60 - +

Algorithm settings Pareto set fraction 0.7 - +

Display progress

Text display Final output

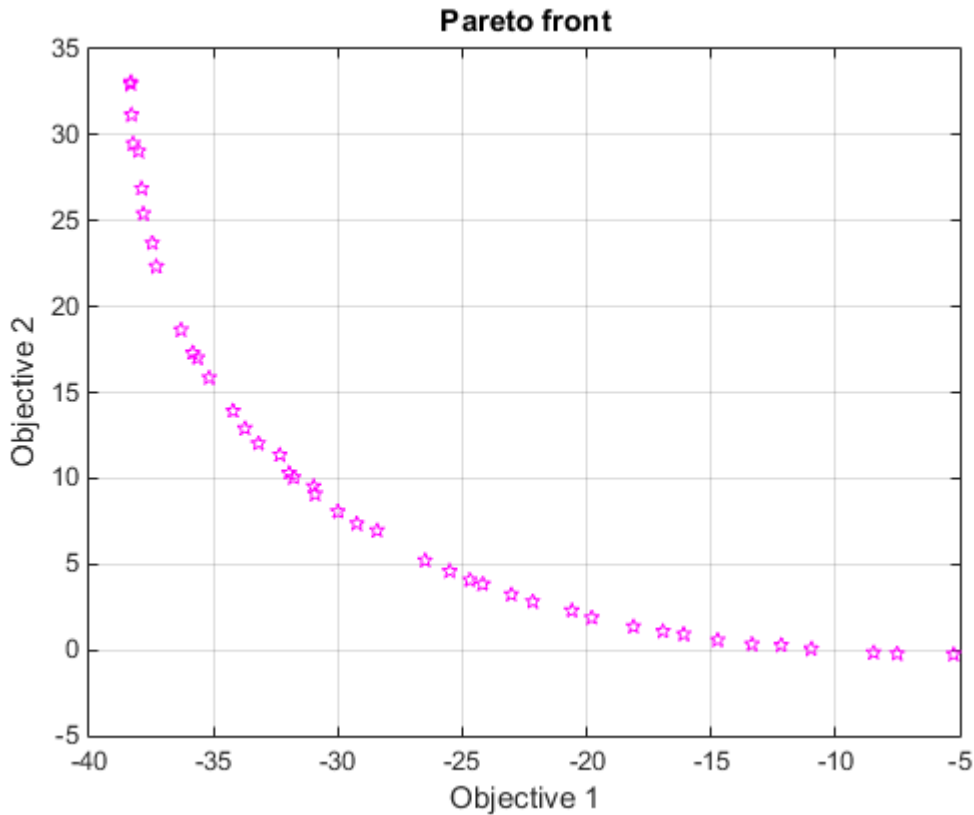
Plot Distance Genealogy Selection Score diversity

Scores Stopping criteria Max constraint violation Rank histogram

Pareto front Average Pareto spread Average Pareto distance

18 Run Solver and Examine Results

To run the solver, click the options button : at the top right of the task window, and select **Run Section**. The plot appears in a separate figure window and in the task output area.



The plot shows the tradeoff between the two components of f , which is plotted in objective function space. For details, see the figure “Figure 14-2, Set of Noninferior Solutions” on page 14-3.

Find Pareto Set at the Command Line

To perform the same optimization at the command line, complete the following steps.

- 1 Create the `mymulti1` objective function file on your MATLAB path.

```
function f = mymulti1(x)
    f(2) = x(1)^4 + x(2)^4 + x(1)*x(2) - (x(1)*x(2))^2;
    f(1) = f(2) - 10*x(1)^2;
end
```

- 2 Set the options and bounds.

```
options = optimoptions('gamultiobj','PopulationSize',60,...
    'ParetoFraction',0.7,'PlotFcn',@gaplotpareto);
lb = [0 -5];
ub = [5 0];
```

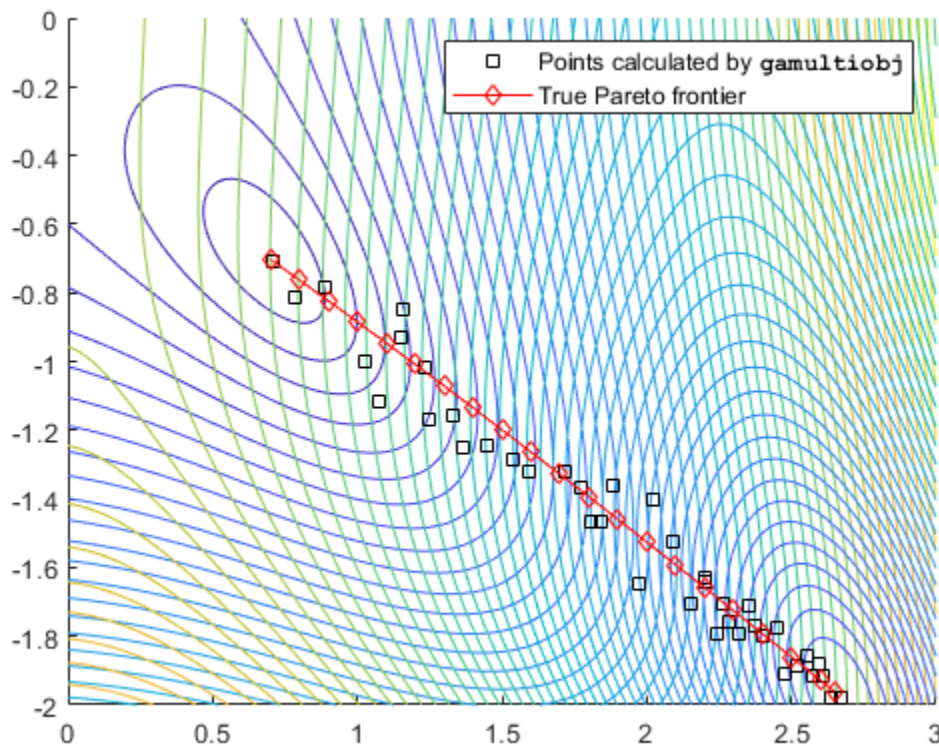
- 3 Run the optimization using the options.

```
[solution,ObjectiveValue] = gamultiobj(@mymulti1,2,...
    [],[],[],[],lb,ub,options);
```


Both the **Optimize** Live Editor task and the command line allow you to formulate and solve problems, and they give identical results. The command line is more streamlined, but provides less help for choosing a solver, setting up the problem, and choosing options such as plot functions. You can also start a problem using **Optimize**, and then generate code for command line use, as in “Constrained Nonlinear Problem Using Optimize Live Editor Task or Solver”.

Alternate Views

You can view this problem in other ways. The following figure contains a plot of the level curves of the two objective functions, the Pareto frontier calculated by `gamultiobj` (boxes), and the x-values of the true Pareto frontier (diamonds connected by a nearly straight line). The true Pareto frontier points are where the level curves of the objective functions are parallel. The algorithm calculates these points by finding where the gradients of the objective functions are parallel. The figure is plotted in parameter space; see “Figure 14-1, Mapping from Parameter Space into Objective Function Space” on page 14-3.



Contours of objective functions, and Pareto frontier

`gamultiobj` finds the ends of the line segment, meaning it finds the full extent of the Pareto frontier.

See Also

`gamultiobj` | `paretosearch`

More About

- “Multiobjective Optimization”
- “Add Interactive Tasks to a Live Script”

Compare paretosearch and gamultiobj

This example shows how to create a set of points on the Pareto front using both `paretosearch` and `gamultiobj`. The objective function has two objectives and a two-dimensional control variable x . The objective function `mymulti3` is available in your MATLAB® session when you click the button to edit or try this example. Alternatively, copy the `mymulti3` code to your session. For speed of calculation, the function is vectorized.

```
type mymulti3
```

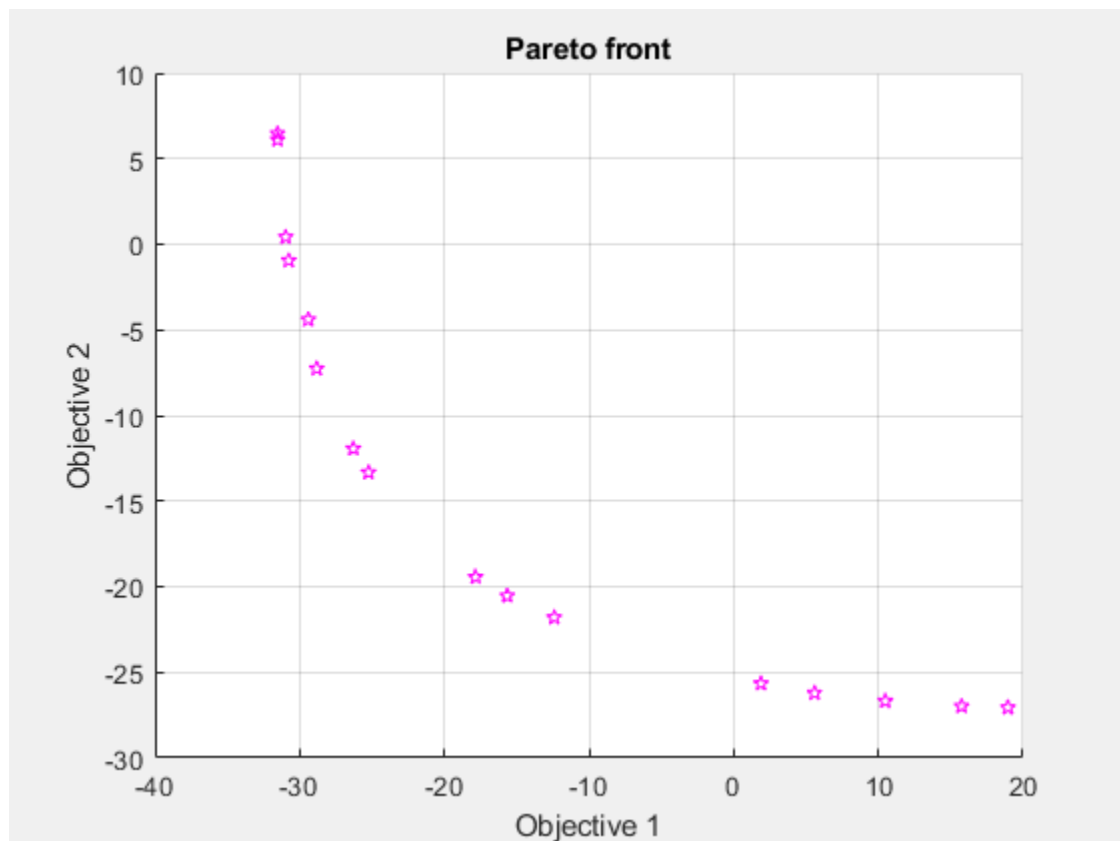
```
function f = mymulti3(x)
%
f(:,1) = x(:,1).^4 + x(:,2).^4 + x(:,1).*x(:,2) - (x(:,1).^2).*(x(:,2).^2) - 9*x(:,1).^2;
f(:,2) = x(:,2).^4 + x(:,1).^4 + x(:,1).*x(:,2) - (x(:,1).^2).*(x(:,2).^2) + 3*x(:,2).^3;
```

Basic Example and Plots

Find Pareto sets for the objective functions using `paretosearch` and `gamultiobj`. Set the `UseVectorized` option to `true` for added speed. Include a plot function to visualize the Pareto set.

```
rng default
nvars = 2;
opts = optimoptions(@gamultiobj, 'UseVectorized', true, 'PlotFcn', 'gaplotpareto');
[xga, fvalga, ~, gaoutput] = gamultiobj(@(x)mymulti3(x), nvars, [], [], [], [], [], [], [], [], opts);
```

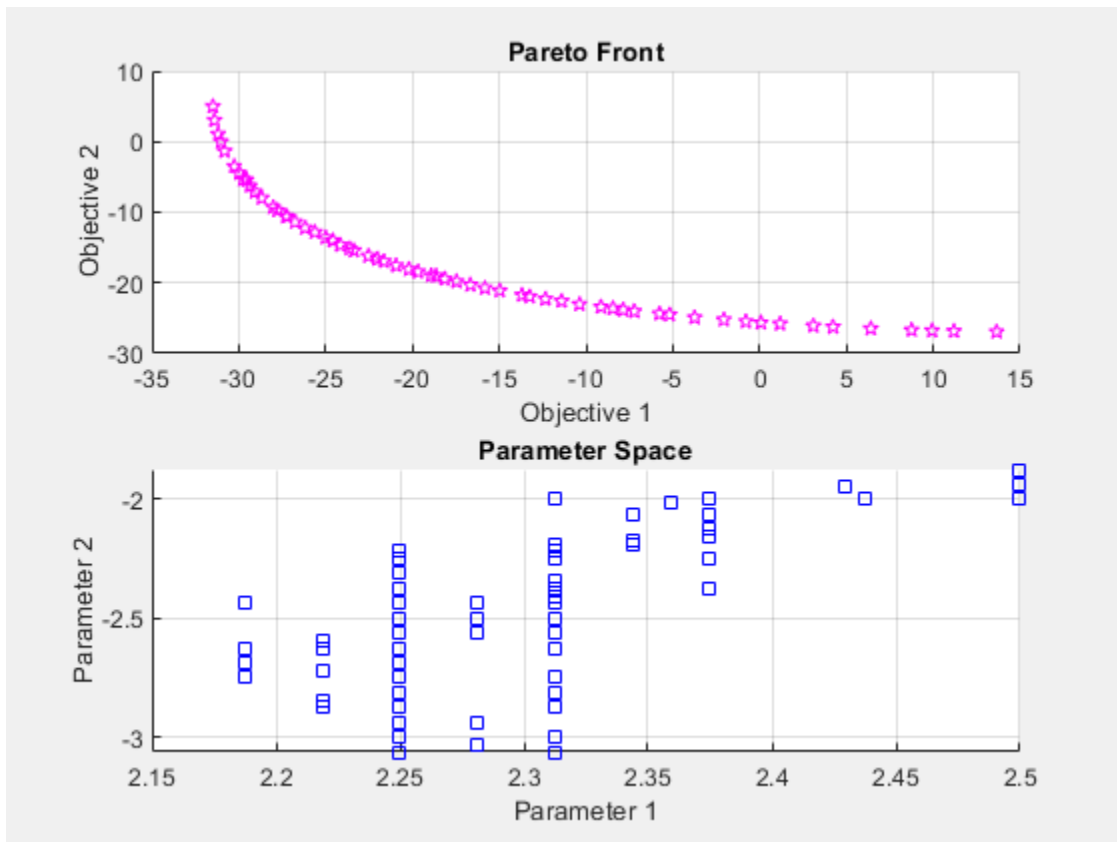
Optimization terminated: average change in the spread of Pareto solutions less than options.Function



```
optsp = optimoptions('paretosearch','UseVectorized',true,'PlotFcn',{'psplotparetof' 'psplotparetof'});
[xp,fvalp,~,psoutput] = paretosearch(@(x)mymulti3(x),nvars,[],[],[],[],[],[],[],[],optsp);
```

Pareto set found that satisfies the constraints.

Optimization completed because the relative change in the volume of the Pareto set is less than 'options.ParetoSetChangeTolerance' and constraints are satisfied to within 'options.ConstraintTolerance'.



Compute theoretically exact points on the Pareto front by using `mymulti4`. The `mymulti4` function is available in your MATLAB session when you click the button to edit or try this example.

```
type mymulti4
```

```
function mout = mymulti4(x)
%
gg = [4*x(1)^3+x(2)-2*x(1)*(x(2)^2) - 18*x(1);
      x(1)+4*x(2)^3-2*(x(1)^2)*x(2)];
gf = gg + [18*x(1);9*x(2)^2];

mout = gf(1)*gg(2) - gf(2)*gg(1);
```

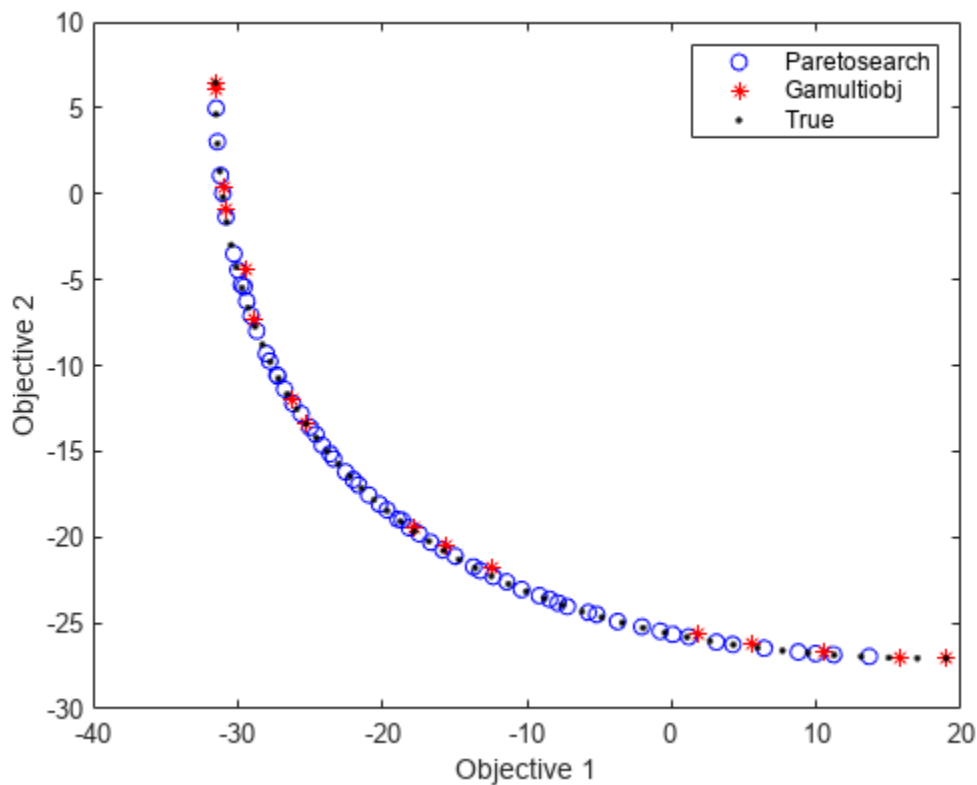
The `mymulti4` function evaluates the gradients of the two objective functions. Next, for a range of values of $x(2)$, use `fzero` to locate the point where the gradients are exactly parallel, which is where the output `mout = 0`.

```
a = [fzero(@(t)mymulti4([t,-3.15]),[2,3]),-3.15];
for jj = linspace(-3.125,-1.89,50)
```

```

    a = [a;[fzero(@(t)mymulti4([t,jj]),[2,3]),jj]];
end
figure
plot(fvalp(:,1),fvalp(:,2),'bo');
hold on
fs = mymulti3(a);
plot(fvalga(:,1),fvalga(:,2),'r*');
plot(fs(:,1),fs(:,2),'k.')
legend('Paretosearch','Gamultiobj','True')
xlabel('Objective 1')
ylabel('Objective 2')
hold off

```



`gamultiobj` finds points with a slightly wider spread in objective function space. Plot the solutions in decision variable space, along with the theoretical optimal Pareto curve and a contour plot of the two objective functions.

```

[x,y] = meshgrid(1.9:.01:3.1,-3.2:.01:-1.8);
mydata = mymulti3([x(:),y(:)]);
myff = sqrt(mydata(:,1) + 39);% Spaces the contours better
mygg = sqrt(mydata(:,2) + 28);% Spaces the contours better
myff = reshape(myff,size(x));
mygg = reshape(mygg,size(x));

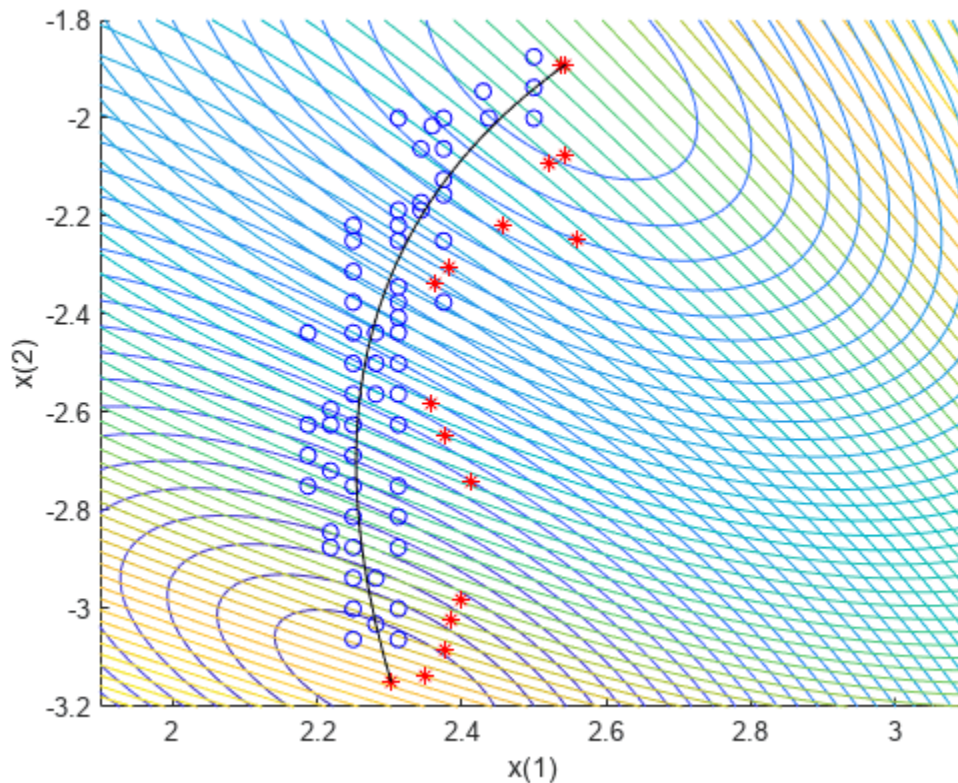
figure;
hold on
contour(x,y,mygg,50)

```

```

contour(x,y,myff,50)
plot(xp(:,1),xp(:,2),'bo')
plot(xga(:,1),xga(:,2),'r*')
plot(a(:,1),a(:,2),'-k')
xlabel('x(1)')
ylabel('x(2)')
hold off

```



Unlike the `paretosearch` solution, the `gamultiobj` solution has points at the extreme ends of the range in objective function space. However, the `paretosearch` solution has more points that are closer to the true solution in both objective function space and decision variable space. The number of points on the Pareto front is different for each solver when you use the default options.

Shifted Problem

What happens if the solution to your problem has control variables that are large? Examine this case by shifting the problem variables. For an unconstrained problem, `gamultiobj` can fail, while `paretosearch` is more robust to such shifts.

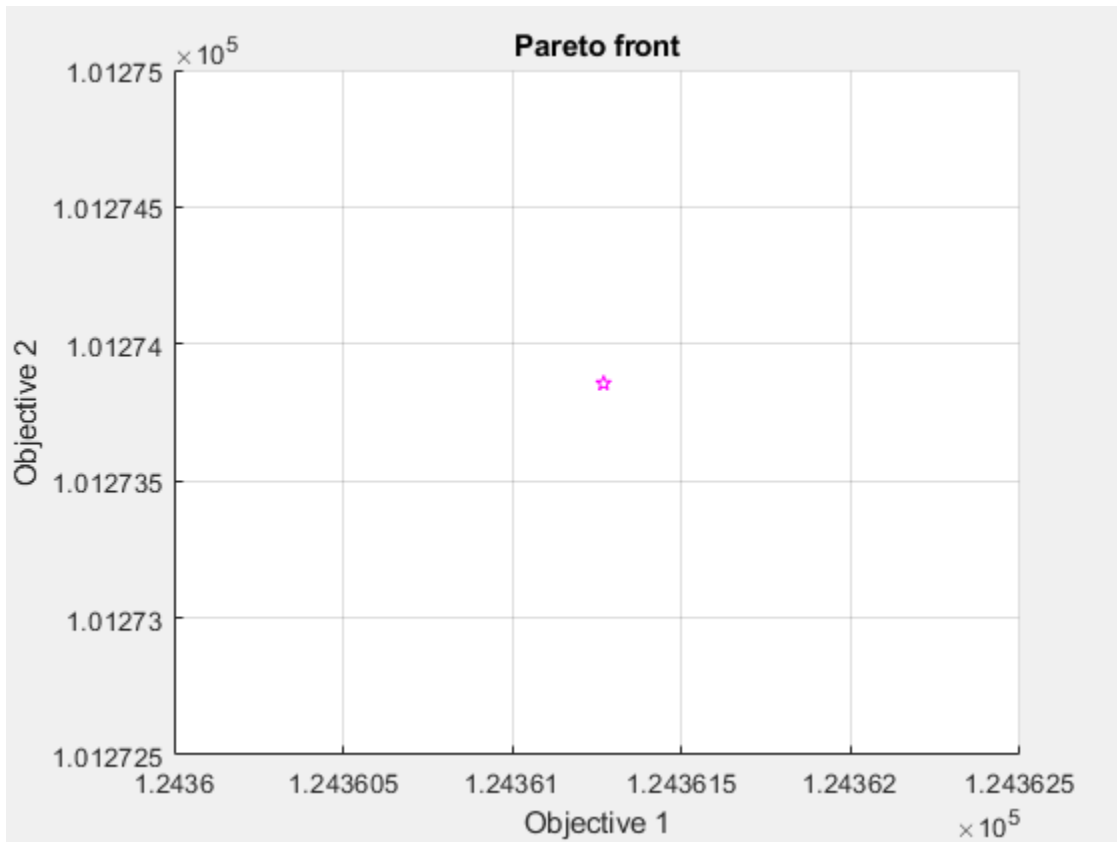
For easier comparison, specify 35 points on the Pareto front for each solver.

```

shift = [20, -30];
fun = @(x)mymulti3(x+shift);
opts.PopulationSize = 100; % opts.ParetoFraction = 35
[xgash,fvalgash,~,gashoutput] = gamultiobj(fun,nvars,[],[],[],[],[],[],opts);

```

Optimization terminated: average change in the spread of Pareto solutions less than options.FuncOpt



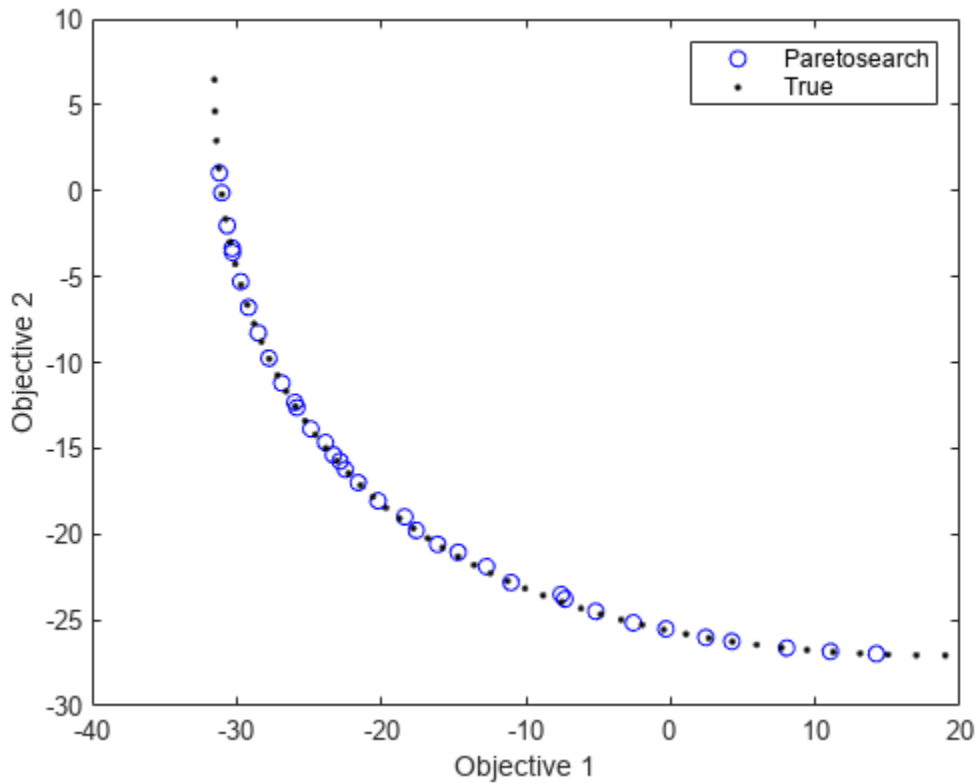
gamultiobj fails to find a useful Pareto set.

```
optsp.PlotFcn = [];
optsp.ParetoSetSize = 35;
[xpsh,fvalpsh,~,pshoutput] = paretosearch(fun,nvars,[],[],[],[],[],[],[],[],optsp);
```

Pareto set found that satisfies the constraints.

Optimization completed because the relative change in the volume of the Pareto set is less than 'options.ParetoSetChangeTolerance' and constraints are satisfied to within 'options.ConstraintTolerance'.

```
figure
plot(fvalpsh(:,1),fvalpsh(:,2),'bo');
hold on
plot(fs(:,1),fs(:,2),'k.')
legend('Paretosearch','True')
xlabel('Objective 1')
ylabel('Objective 2')
hold off
```



paretosearch finds solution points spread evenly over nearly the entire possible range.

Adding bounds, even fairly loose ones, helps both `gamultiobj` and `paretosearch` to find appropriate solutions. Set lower bounds of -50 in each component, and upper bounds of 50.

```
opts.PlotFcn = [];
optsp.PlotFcn = [];
lb = [-50,-50];
ub = -lb;
[xgash,fvalgash,~,gashoutput] = gamultiobj(fun,nvars,[],[],[],[],lb,ub,opts);
```

Optimization terminated: average change in the spread of Pareto solutions less than options.FunctionTolerance

```
[xpsh2,fvalpsh2,~,pshoutput2] = paretosearch(fun,nvars,[],[],[],[],lb,ub,[],optsp);
```

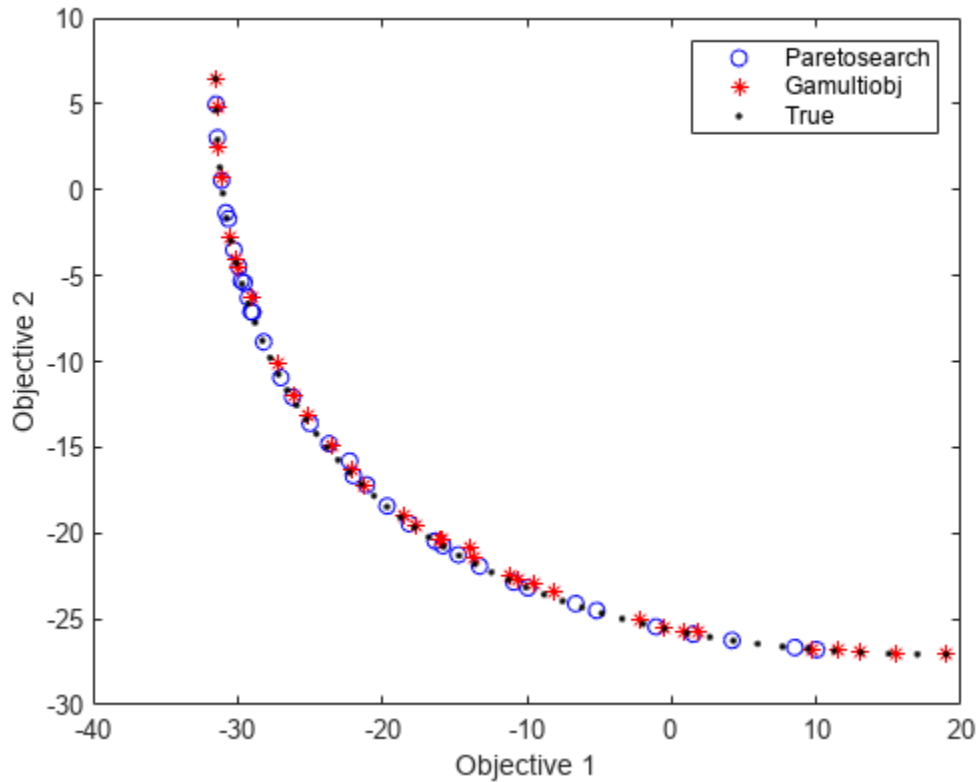
Pareto set found that satisfies the constraints.

Optimization completed because the relative change in the volume of the Pareto set is less than 'options.ParetoSetChangeTolerance' and constraints are satisfied to within 'options.ConstraintTolerance'.

```
figure
plot(fvalpsh2(:,1),fvalpsh2(:,2),'bo');
hold on
plot(fvalgash(:,1),fvalgash(:,2),'r*');
plot(fs(:,1),fs(:,2),'k.')
legend('Paretosearch','Gamultiobj','True')
xlabel('Objective 1')
```



```
ylabel('Objective 2')
hold off
```



In this case, both solvers find good solutions.

Start paretosearch from gamultiobj Solution

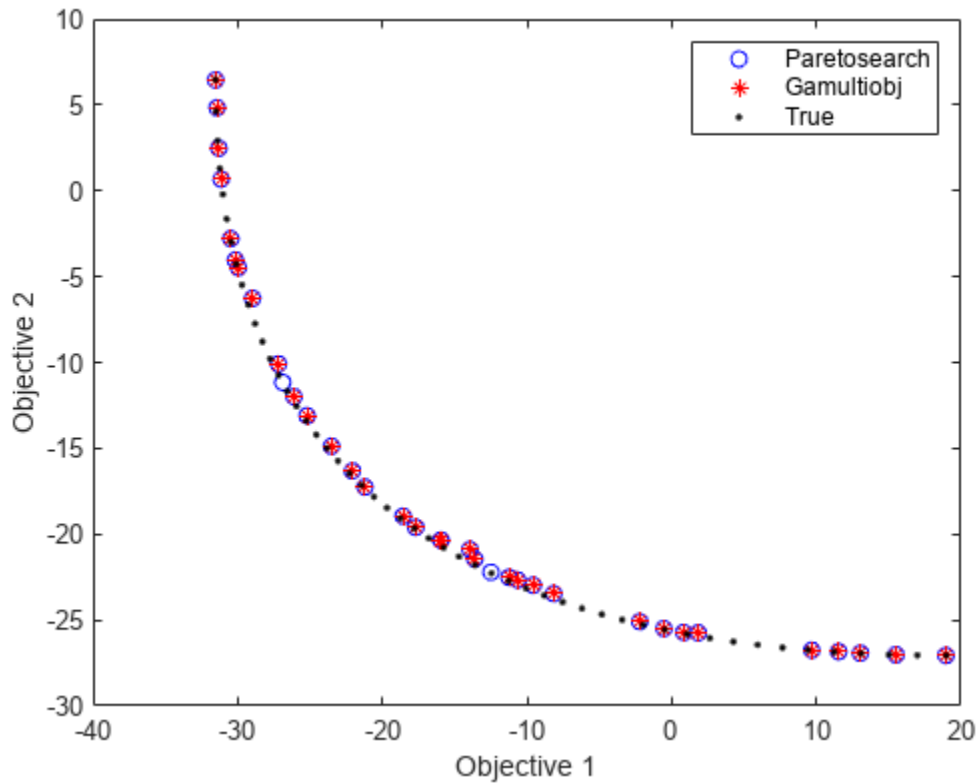
Obtain a similar range of solutions from the solvers by starting paretosearch from the gamultiobj solution.

```
optsp.InitialPoints = xgash;
[xpsh3,fvalpsh3,~,pshoutput3] = paretosearch(fun,nvars,[],[],[],[],lb,ub,[],optsp);
```

Pareto set found that satisfies the constraints.

Optimization completed because the relative change in the volume of the Pareto set is less than 'options.ParetoSetChangeTolerance' and constraints are satisfied to within 'options.ConstraintTolerance'.

```
figure
plot(fvalpsh3(:,1),fvalpsh3(:,2),'bo');
hold on
plot(fvalgash(:,1),fvalgash(:,2),'r*');
plot(fs(:,1),fs(:,2),'k.')
legend('Paretosearch','Gamultiobj','True')
xlabel('Objective 1')
ylabel('Objective 2')
hold off
```



Now the paretosearch solution is similar to the gamultiobj solution, although some of the solution points differ.

Start from Single-Objective Solutions

Another way of obtaining a good solution is to start from the points that minimize each objective function separately.

From the multiobjective function, create a single-objective function that chooses each objective in turn. Use the shifted function from the previous section. Because you are giving good start points to the solvers, you do not need to specify bounds.

```
nobj = 2; % Number of objectives

x0 = -shift; % Initial point for single-objective minimization
uncmin = cell(nobj,1); % Cell array to hold the single-objective minima
allfuns = zeros(nobj,2); % Hold the objective function values
eflag = zeros(nobj,1);
fopts = optimoptions('patternsearch','Display','off'); % Use an appropriate solver here
for i = 1:nobj
    indi = zeros(nobj,1); % Choose the objective to minimize
    indi(i) = 1;
    funi = @(x)dot(fun(x),indi);
    [uncmin{i},~,eflag(i)] = patternsearch(funi,x0,[],[],[],[],[],[],[],fopts); % Minimize objective
    allfuns(i,:) = fun(uncmin{i});
end
uncmin = cell2mat(uncmin); % Matrix of start points
```

Start `paretosearch` from the single-objective minimum points and note that it has a full range in its solutions. `paretosearch` adds random initial points to the supplied ones in order to have a population of at least `options.ParetoSetSize` individuals. Similarly, `gamultiobj` adds random points to the supplied ones to obtain a population of at least $(\text{options.PopulationSize}) * (\text{options.ParetoFraction})$ individuals.

```
optsp = optimoptions(optsp, 'InitialPoints', uncmin);
[xpinit, fvalpinit, ~, outputpinit] = paretosearch(fun, nvars, [], [], [], [], [], [], [], [], optsp);
```

Pareto set found that satisfies the constraints.

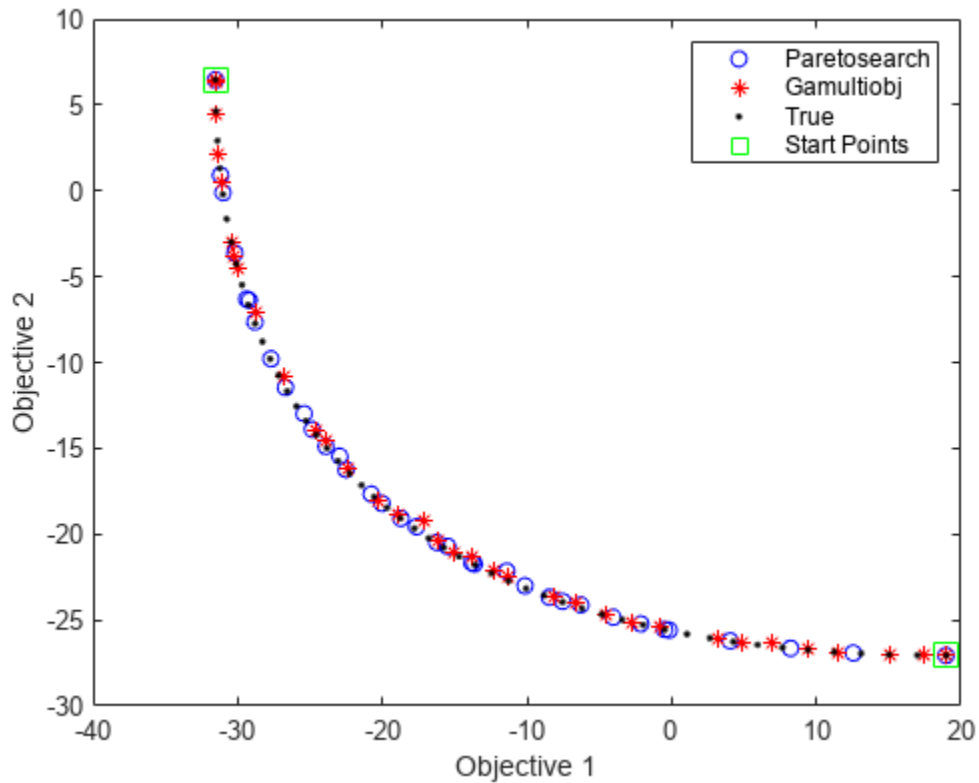
Optimization completed because the relative change in the volume of the Pareto set is less than `'options.ParetoSetChangeTolerance'` and constraints are satisfied to within `'options.ConstraintTolerance'`.

Now solve the problem using `gamultiobj` starting from the initial points.

```
opts = optimoptions(opts, 'InitialPopulationMatrix', uncmin);
[xgash2, fvalgash2, ~, gashoutput2] = gamultiobj(fun, nvars, [], [], [], [], [], [], [], opts);
```

Optimization terminated: average change in the spread of Pareto solutions less than `options.Function`

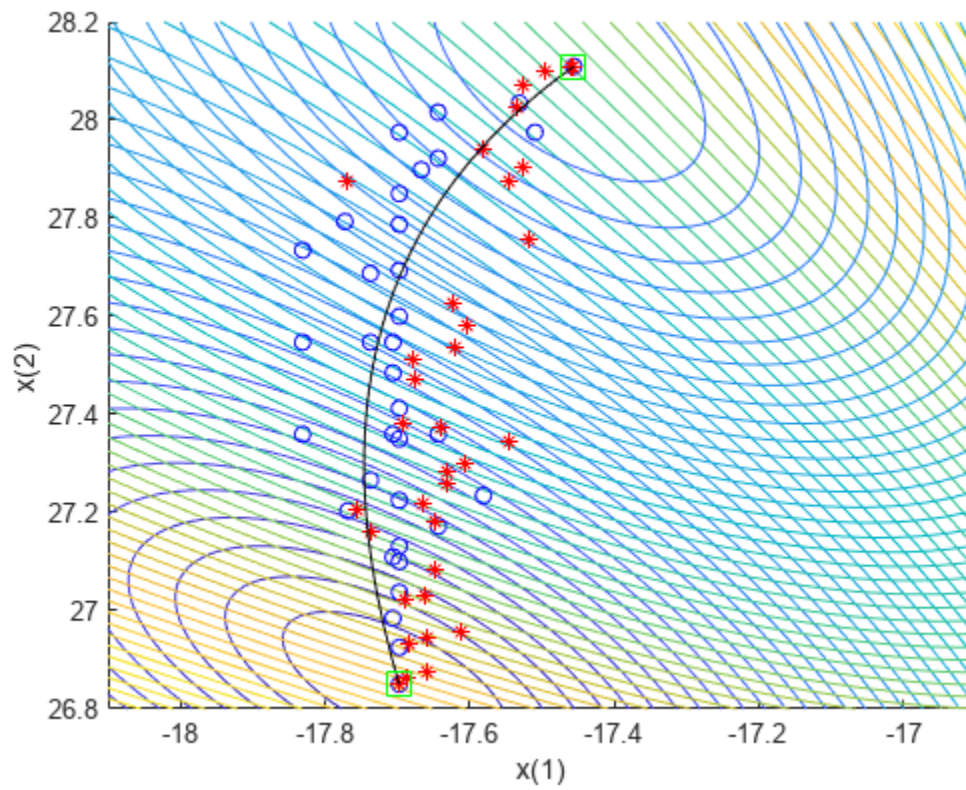
```
figure
plot(fvalpinit(:,1), fvalpinit(:,2), 'bo');
hold on
plot(fvalgash2(:,1), fvalgash2(:,2), 'r*');
plot(fs(:,1), fs(:,2), 'k.')
plot(allfuns(:,1), allfuns(:,2), 'gs', 'MarkerSize', 12)
legend('Paretosearch', 'Gamultiobj', 'True', 'Start Points')
xlabel('Objective 1')
ylabel('Objective 2')
hold off
```



Both solvers fill in the Pareto front between the extreme points, with reasonably accurate and well-spaced solutions.

View the final points in decision variable space.

```
figure;
hold on
xx = x - shift(1);
yy = y - shift(2);
contour(xx,yy,mygg,50)
contour(xx,yy,myff,50)
plot(xpinit(:,1),xpinit(:,2), 'bo')
plot(xgash2(:,1),xgash2(:,2), 'r*')
ashift = a - shift;
plot(ashift(:,1),ashift(:,2), '-k')
plot(uncmin(:,1),uncmin(:,2), 'gs', 'MarkerSize', 12);
xlabel('x(1)')
ylabel('x(2)')
hold off
```



See Also

gamultiobj | paretosearch

More About

- “Multiobjective Optimization”

Plot 3-D Pareto Front

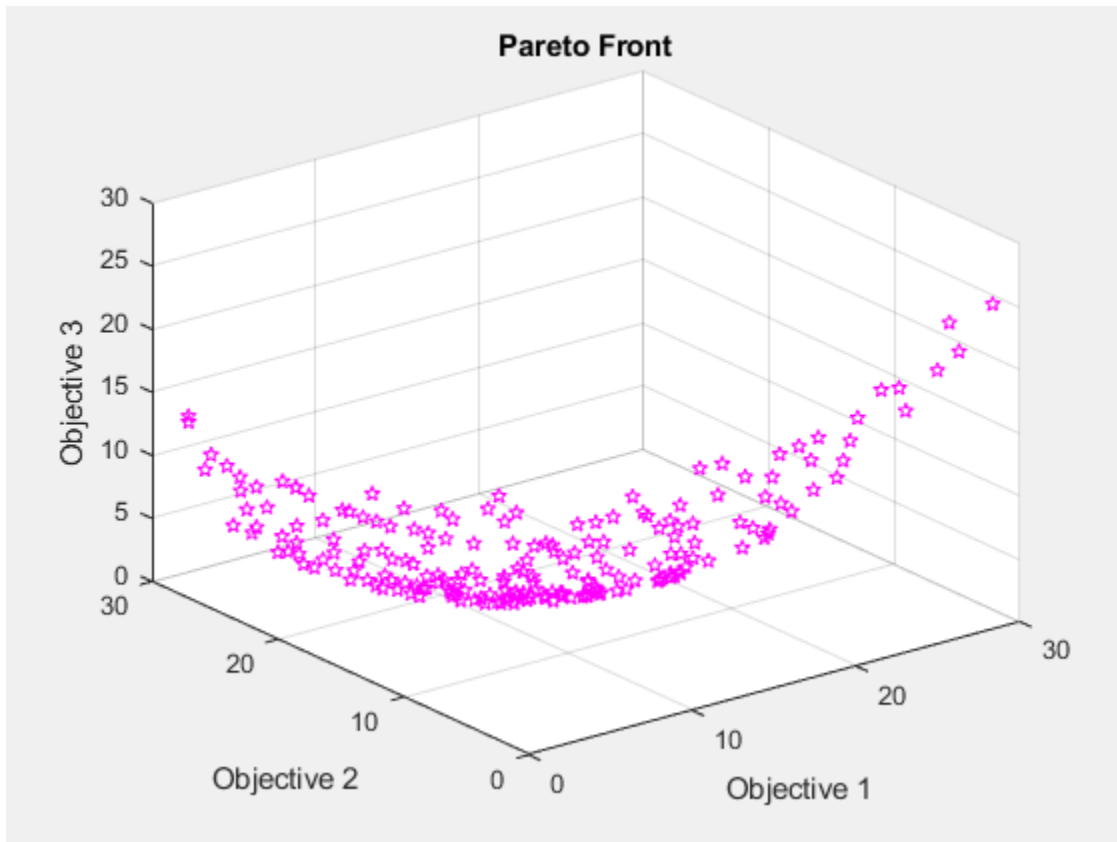
This example shows how to plot a Pareto front for three objectives. Each objective function is the squared distance from a particular 3-D point. For speed of calculation, write each objective function in vectorized fashion as a dot product. To obtain a dense solution set, use 200 points on the Pareto front.

The example first shows how to obtain the plot using the built-in 'psplotparetof' plot function. Then solve the same problem and obtain the plot using gamultiobj, which requires slightly different option settings. The example shows how to obtain solution variables for a particular point in the Pareto plot. Then the example shows how to plot the points directly, without using a plot function, and shows how to plot an interpolated surface instead of Pareto points.

```
fun = @(x)[dot(x - [1,2,3],x - [1,2,3],2), ...  
          dot(x - [-1,3,-2],x - [-1,3,-2],2), ...  
          dot(x - [0,-1,1],x - [0,-1,1],2)];  
options = optimoptions('paretosearch','UseVectorized',true,'ParetoSetSize',200,...  
                      'PlotFcn','psplotparetof');  
lb = -5*ones(1,3);  
ub = -lb;  
rng default % For reproducibility  
[x,f] = paretosearch(fun,3,[],[],[],[],lb,ub,[],options);
```

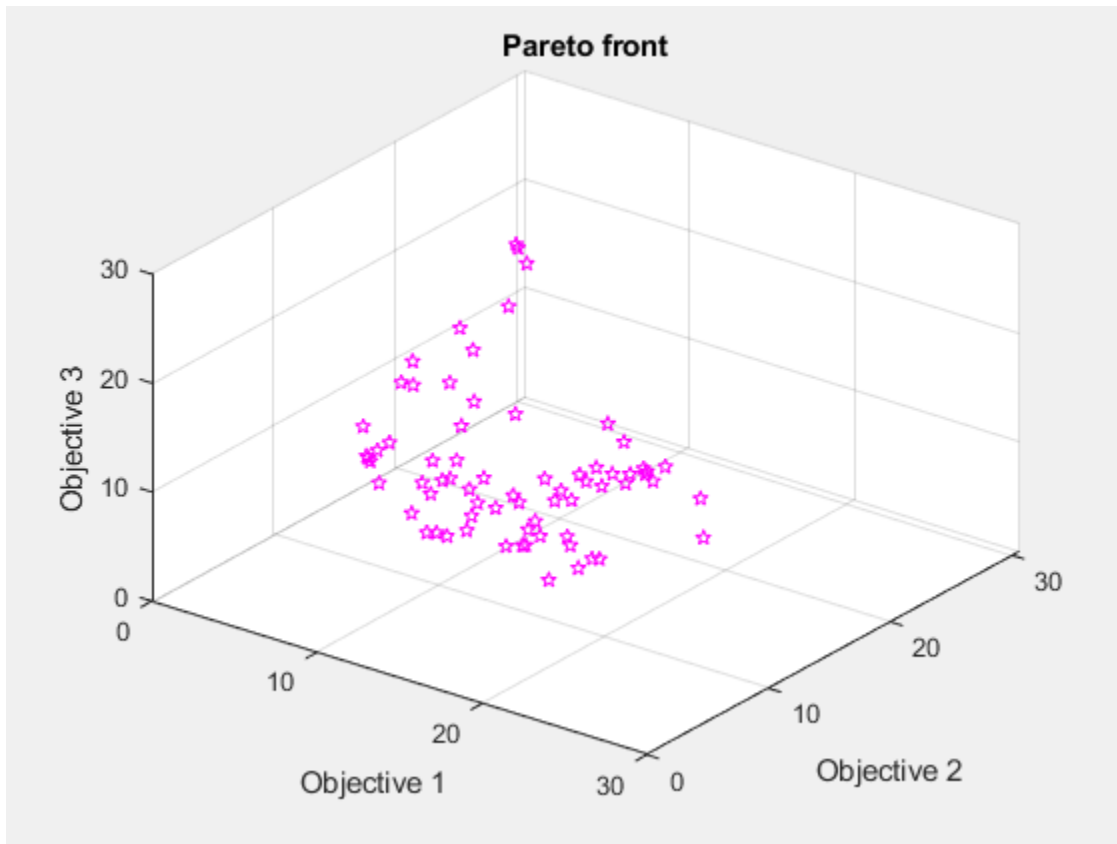
Pareto set found that satisfies the constraints.

Optimization completed because the relative change in the volume of the Pareto set is less than 'options.ParetoSetChangeTolerance' and constraints are satisfied to within 'options.ConstraintTolerance'.



```
opts = optimoptions('gamultiobj','PlotFcn','gaplotpareto','PopulationSize',200);  
[xg,fg] = gamultiobj(fun,3,[],[],[],[],[],lb,ub,[],opts);
```

Optimization terminated: average change in the spread of Pareto solutions less than options.Function



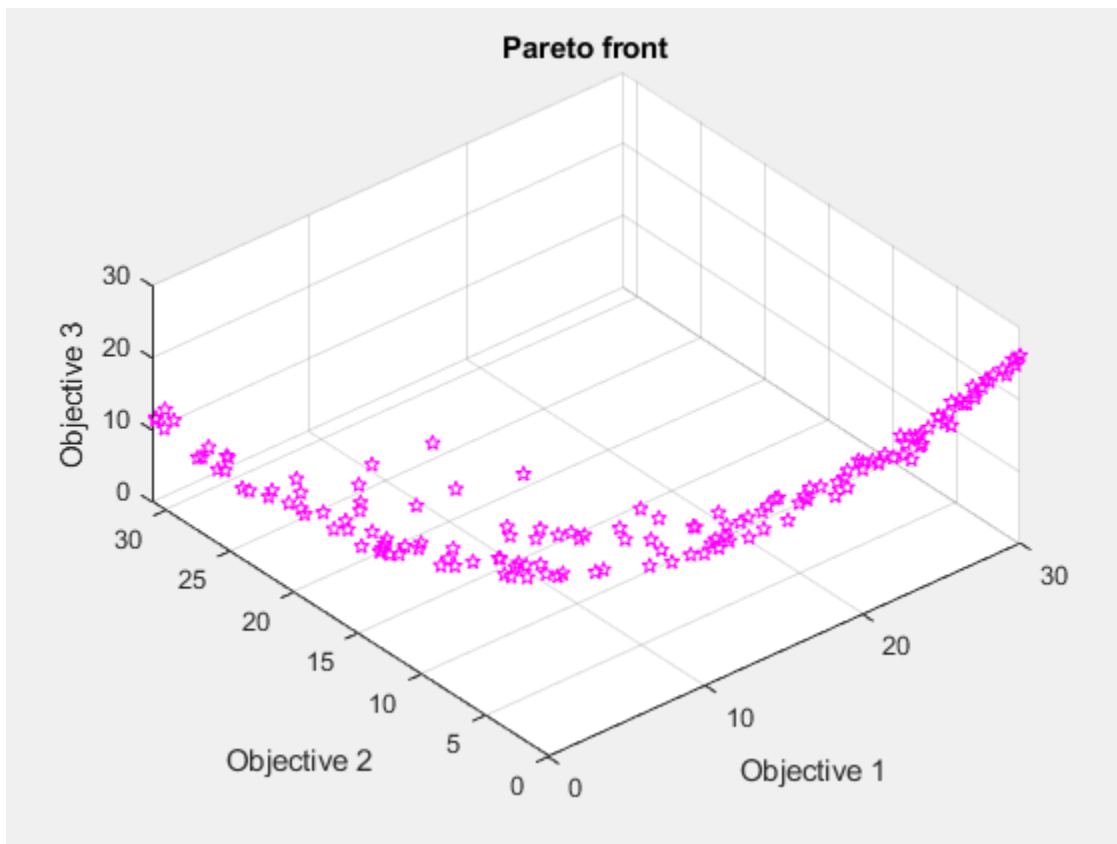
This plot shows many fewer points than the `paretosearch` plot. Solve the problem again using a larger population.

```
opts.PopulationSize = 400;  
[xg,fg] = gamultiobj(fun,3,[],[],[],[],lb,ub,[],opts);
```

Optimization terminated: average change in the spread of Pareto solutions less than options.Function

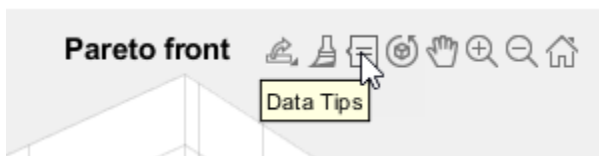
Change the viewing angle to better match the `psplotpareto` plot.

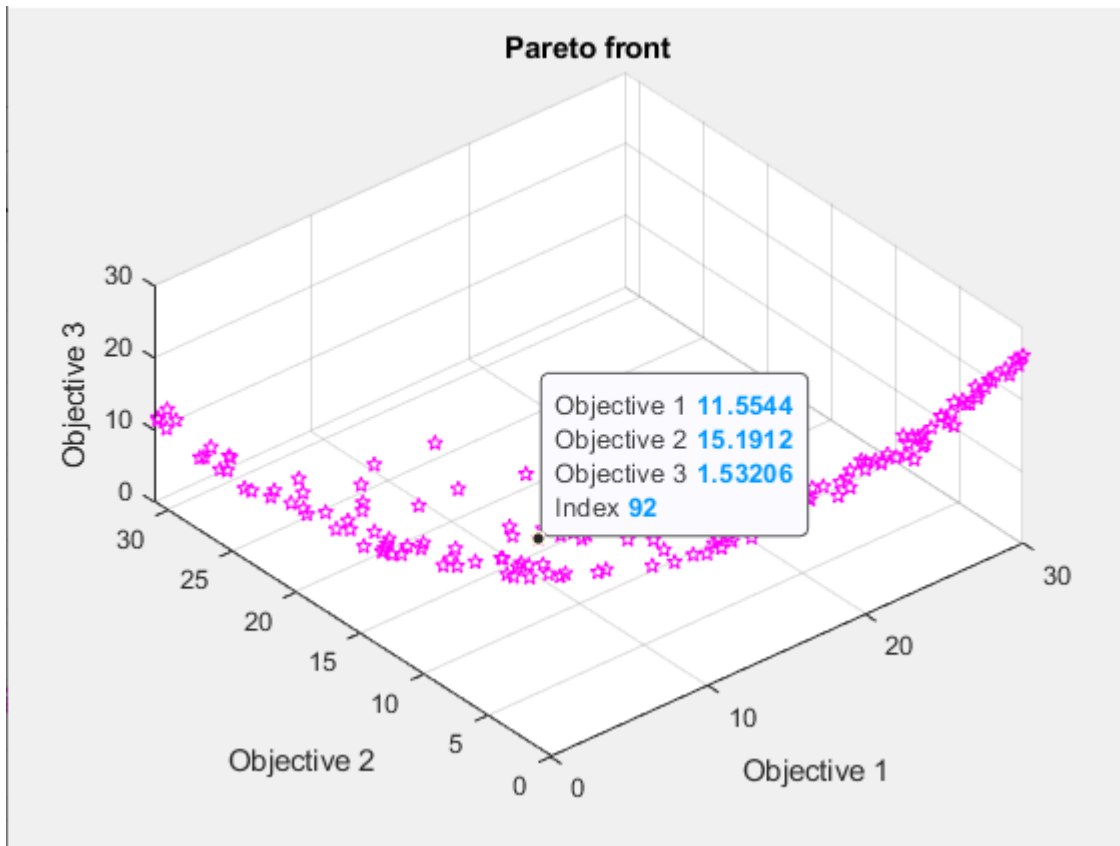
```
view(-40,57)
```

Find Solution Point Using Tool Tips

Select a point in the plot by using the Data Tips tool.





The pictured point has index 92. Display the point `xg(92, :)` that contains the solution variables associated with the pictured point.

```
disp(xg(92, :))
-0.2889    0.0939    0.4980
```

Evaluate the objective functions at this point to see that it matches the displayed values.

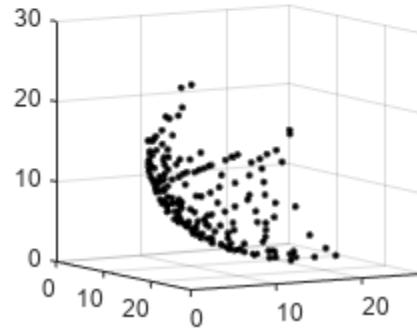
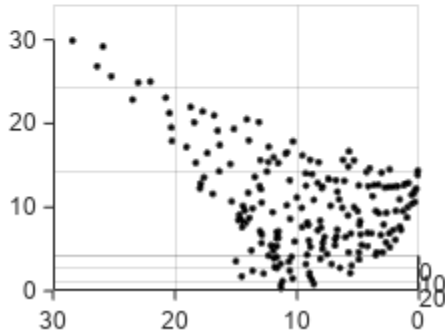
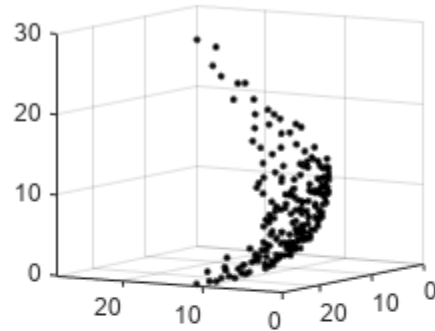
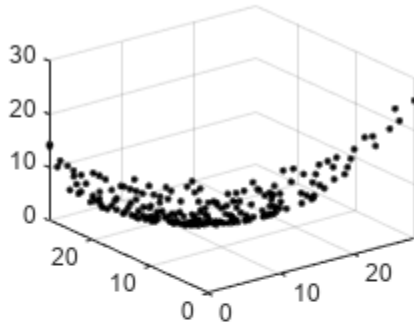
```
disp(fun(xg(92, :)))
11.5544    15.1912    1.5321
```

Create 3-D Scatter Plot

Plot points on the Pareto front by using `scatter3`.

```
figure
subplot(2,2,1)
scatter3(f(:,1),f(:,2),f(:,3), 'k. ');
subplot(2,2,2)
scatter3(f(:,1),f(:,2),f(:,3), 'k. ');
view(-148,8)
subplot(2,2,3)
scatter3(f(:,1),f(:,2),f(:,3), 'k. ');
view(-180,8)
subplot(2,2,4)
```

```
scatter3(f(:,1),f(:,2),f(:,3),'k. ');
view(-300,8)
```



By rotating the plot interactively, you get a better view of its structure.

Interpolated Surface Plot

To see the Pareto front as a surface, create a scattered interpolant.

```
figure
F = scatteredInterpolant(f(:,1),f(:,2),f(:,3),'linear','none');
```

To plot the resulting surface, create a mesh in x-y space from the smallest to the largest values. Then plot the interpolated surface.

```
sgr = linspace(min(f(:,1)),max(f(:,1)));
ygr = linspace(min(f(:,2)),max(f(:,2)));
[XX,YY] = meshgrid(sgr,ygr);
ZZ = F(XX,YY);
```

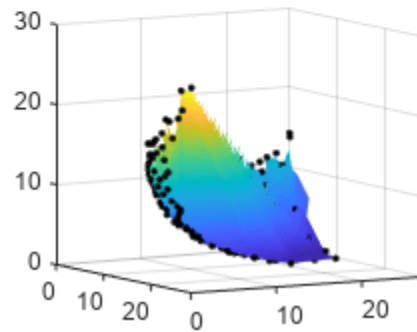
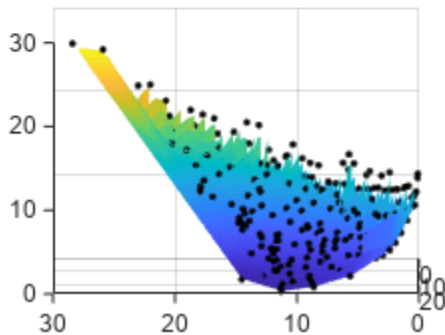
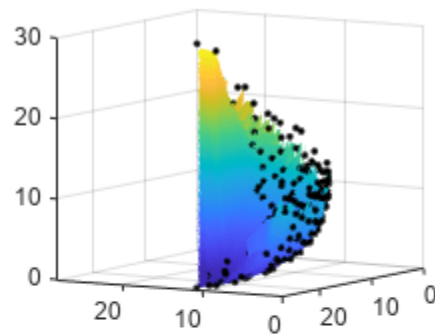
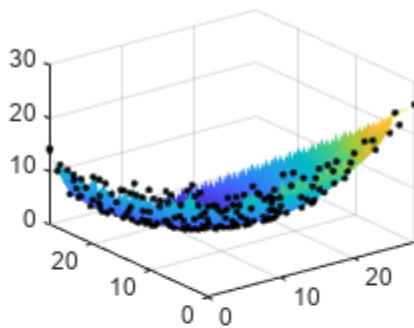
Plot the Pareto points and surface together.

```
figure
subplot(2,2,1)
surf(XX,YY,ZZ,'LineStyle','none')
hold on
scatter3(f(:,1),f(:,2),f(:,3),'k. ');
hold off
```

```

subplot(2,2,2)
surf(XX,YY,ZZ, 'LineStyle', 'none')
hold on
scatter3(f(:,1),f(:,2),f(:,3), 'k. ');
hold off
view(-148,8)
subplot(2,2,3)
surf(XX,YY,ZZ, 'LineStyle', 'none')
hold on
scatter3(f(:,1),f(:,2),f(:,3), 'k. ');
hold off
view(-180,8)
subplot(2,2,4)
surf(XX,YY,ZZ, 'LineStyle', 'none')
hold on
scatter3(f(:,1),f(:,2),f(:,3), 'k. ');
hold off
view(-300,8)

```



By rotating the plot interactively, you get a better view of its structure.

Plot Pareto Set in Control Variable Space

You can obtain a plot of the points on the Pareto set by using the 'psplotparetox' plot function.

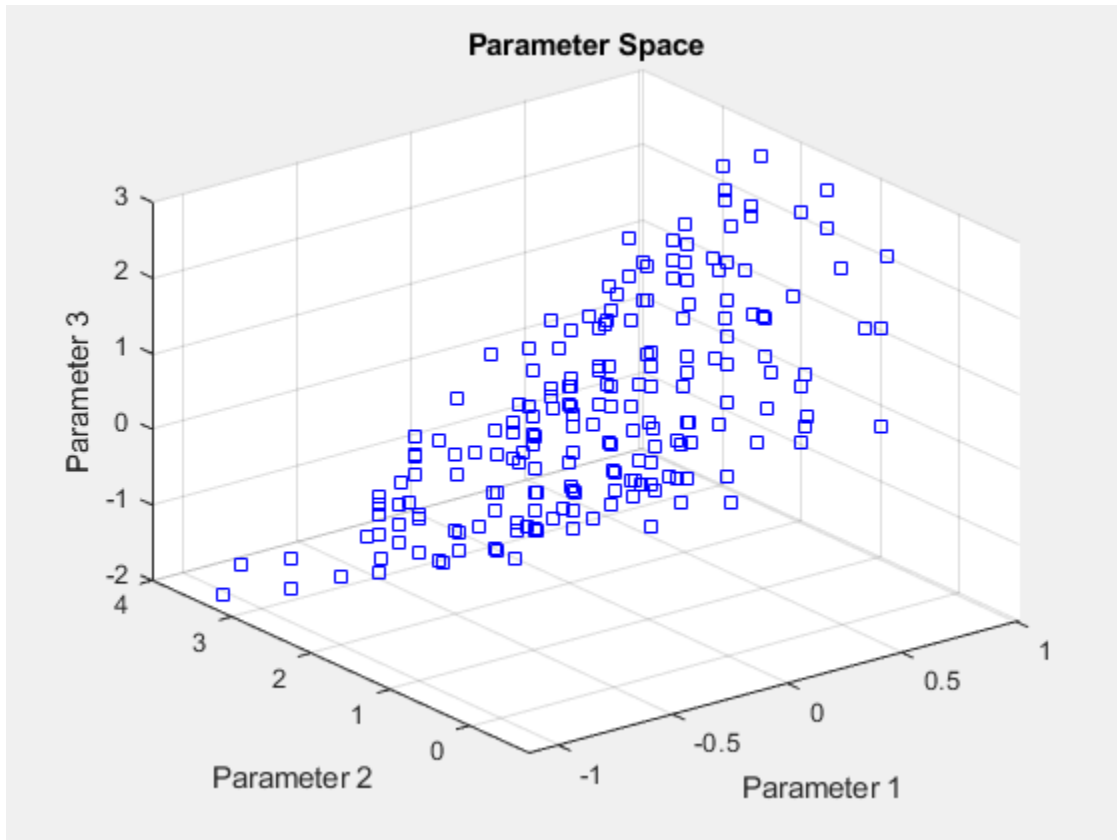
```

options.PlotFcn = 'psplotparetox';
[x,f] = paretosearch(fun,3,[],[],[],[],lb,ub,[],options);

```

Pareto set found that satisfies the constraints.

Optimization completed because the relative change in the volume of the Pareto set is less than 'options.ParetoSetChangeTolerance' and constraints are satisfied to within 'options.ConstraintTolerance'.



Alternatively, create a scatter plot of the x-values in the Pareto set.

```
figure
subplot(2,2,1)
scatter3(x(:,1),x(:,2),x(:,3),'k.');
```

subplot(2,2,2)

```
scatter3(x(:,1),x(:,2),x(:,3),'k.');
```

view(-148,8)

subplot(2,2,3)

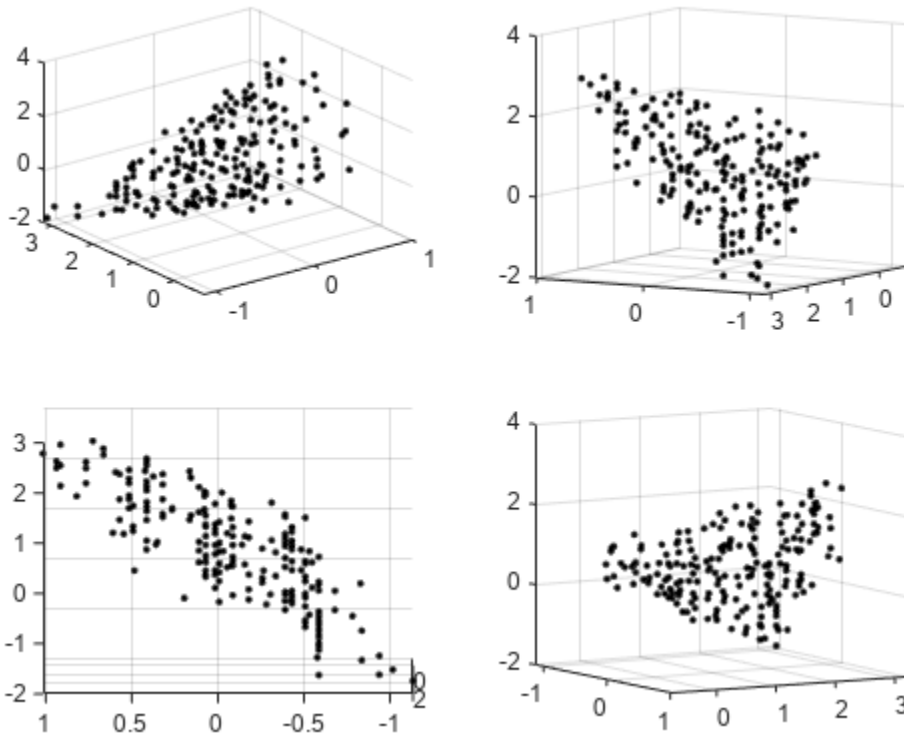
```
scatter3(x(:,1),x(:,2),x(:,3),'k.');
```

view(-180,8)

subplot(2,2,4)

```
scatter3(x(:,1),x(:,2),x(:,3),'k.');
```

view(-300,8)



This set does not have a clear surface. By rotating the plot interactively, you get a better view of its structure.

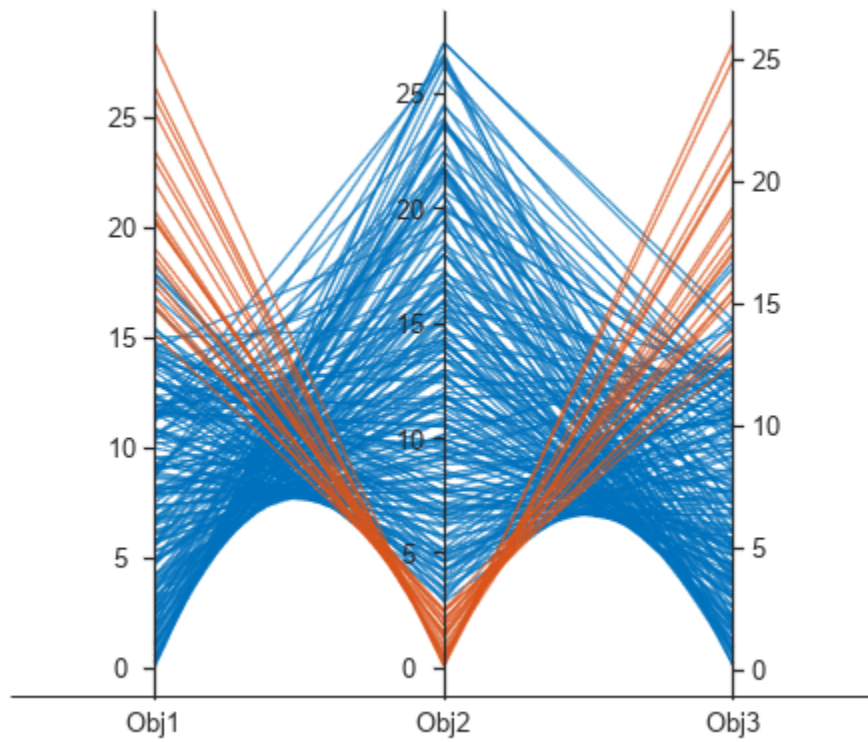
Parallel Plot

You can plot the Pareto set using a parallel coordinates plot. You can use a parallel coordinates plot for any number of dimensions. In the plot, each colored line represents one Pareto point, and each coordinate variable plots to an associated vertical line. Plot the objective function values using `parallelplot`.

```
figure
p = parallelplot(f);
p.CoordinateTickLabels = ["Obj1";"Obj2";"Obj3"];
```

Color the Pareto points in the lowest tenth of the values of `Obj2`.

```
minObj2 = min(f(:,2));
maxObj2 = max(f(:,2));
grpRng = minObj2 + 0.1*(maxObj2-minObj2);
grpData = f(:,2) <= grpRng;
p.GroupData = grpData;
p.LegendVisible = "off";
```

**See Also**

`gamultiobj` | `paretosearch`

More About

- “Multiobjective Optimization”

Performing a Multiobjective Optimization Using the Genetic Algorithm

This example shows how to perform a multiobjective optimization using multiobjective genetic algorithm function `gamultiobj` in Global Optimization Toolbox.

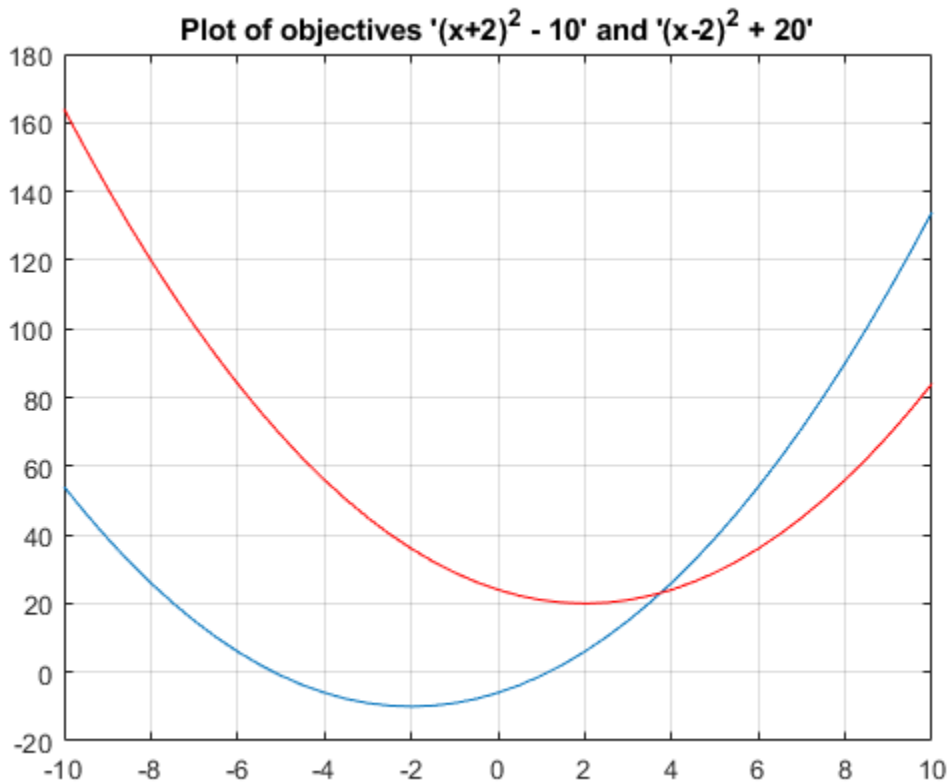
Simple Multiobjective Optimization Problem

`gamultiobj` can be used to solve multiobjective optimization problem in several variables. Here we want to minimize two objectives, each having one decision variable.

```
min F(x) = [objective1(x); objective2(x)]
           x

where, objective1(x) = (x+2)^2 - 10, and
      objective2(x) = (x-2)^2 + 20

% Plot two objective functions on the same axis
x = -10:0.5:10;
f1 = (x+2).^2 - 10;
f2 = (x-2).^2 + 20;
plot(x,f1);
hold on;
plot(x,f2,'r');
grid on;
title('Plot of objectives ''(x+2)^2 - 10'' and ''(x-2)^2 + 20''');
```



The two objectives have their minima at $x = -2$ and $x = +2$ respectively. However, in a multiobjective problem, $x = -2$, $x = 2$, and any solution in the range $-2 \leq x \leq 2$ is equally optimal. There is no single solution to this multiobjective problem. The goal of the multiobjective genetic algorithm is to find a set of solutions in that range (ideally with a good spread). The set of solutions is also known as a Pareto front. All solutions on the Pareto front are optimal.

Coding the Fitness Function

We create a MATLAB® file named `simple_multiobjective.m`:

```
function y = simple_multiobjective(x)
y(1) = (x+2)^2 - 10;
y(2) = (x-2)^2 + 20;
```

The Genetic Algorithm solver assumes the fitness function will take one input x , where x is a row vector with as many elements as the number of variables in the problem. The fitness function computes the value of each objective function and returns these values in a single vector output y .

Minimizing Using `gamultiobj`

To use the `gamultiobj` function, we need to provide at least two input arguments, a fitness function, and the number of variables in the problem. The first two output arguments returned by `gamultiobj` are X , the points on Pareto front, and $FVAL$, the objective function values at the values X . A third output argument, `exitFlag`, tells you the reason why `gamultiobj` stopped. A fourth argument, `OUTPUT`, contains information about the performance of the solver. `gamultiobj` can also return a fifth argument, `POPULATION`, that contains the population when `gamultiobj` terminated and a sixth argument, `SCORE`, that contains the function values of all objectives for `POPULATION` when `gamultiobj` terminated.

```
FitnessFunction = @simple_multiobjective;
numberOfVariables = 1;
[x,fval] = gamultiobj(FitnessFunction,numberOfVariables);
```

Optimization terminated: maximum number of generations exceeded.

The X returned by the solver is a matrix in which each row is the point on the Pareto front for the objective functions. The $FVAL$ is a matrix in which each row contains the value of the objective functions evaluated at the corresponding point in X .

```
size(x)
size(fval)
```

```
ans =
    18     1
```

```
ans =
    18     2
```

Constrained Multiobjective Optimization Problem

`gamultiobj` can handle optimization problems with linear inequality, equality, and simple bound constraints. Here we want to add bound constraints on simple multiobjective problem solved previously.

```
min F(x) = [objective1(x); objective2(x)]
x

subject to -1.5 <= x <= 0 (bound constraints)

where, objective1(x) = (x+2)^2 - 10, and
      objective2(x) = (x-2)^2 + 20
```

`gamultiobj` accepts linear inequality constraints in the form $A*x \leq b$ and linear equality constraints in the form $Aeq*x = beq$ and bound constraints in the form $lb \leq x \leq ub$. We pass A and Aeq as matrices and b , beq , lb , and ub as vectors. Since we have no linear constraints in this example, we pass `[]` for those inputs.

```
A = []; b = [];
Aeq = []; beq = [];
lb = -1.5;
ub = 0;
x = gamultiobj(FitnessFunction,numberOfVariables,A,b,Aeq,beq,lb,ub);
```

Optimization terminated: maximum number of generations exceeded.

All solutions in X (each row) will satisfy all linear and bound constraints within the tolerance specified in `options.ConstraintTolerance`. However, if you use your own crossover or mutation function, ensure that the new individuals are feasible with respect to linear and simple bound constraints.

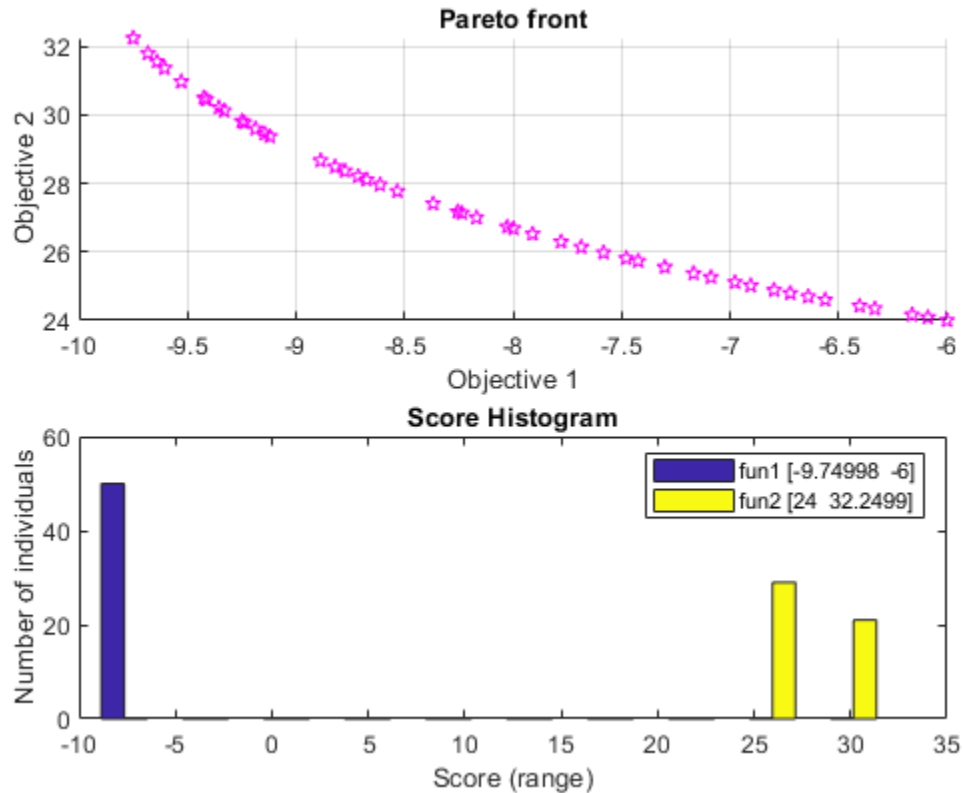
Adding Visualization

`gamultiobj` can accept one or more plot functions through the options argument. This feature is useful for visualizing the performance of the solver at run time. Plot functions can be selected using `optimoptions`.

Here we use `optimoptions` to select two plot functions. The first plot function is `gaplotpareto`, which plots the Pareto front (limited to any three objectives) at every generation. The second plot function is `gaplotscorediversity`, which plots the score diversity for each objective. The options are passed as the last argument to the solver.

```
options = optimoptions(@gamultiobj,'PlotFcn',{@gaplotpareto,@gaplotscorediversity});
gamultiobj(FitnessFunction,numberOfVariables,[],[],[],[],lb,ub,options);
```

Optimization terminated: maximum number of generations exceeded.



Vectorizing Your Fitness Function

Consider the previous fitness functions again:

$$\begin{aligned} \text{objective1}(x) &= (x+2)^2 - 10, \text{ and} \\ \text{objective2}(x) &= (x-2)^2 + 20 \end{aligned}$$

By default, the `gamultiobj` solver only passes in one point at a time to the fitness function. However, if the fitness function is vectorized to accept a set of points and returns a set of function values you can speed up your solution.

For example, if the solver needs to evaluate five points in one call to this fitness function, then it will call the function with a matrix of size 5-by-1, i.e., 5 rows and 1 column (recall that 1 is the number of variables).

Create a MATLAB file called `vectorized_multiobjective.m`:

```
function scores = vectorized_multiobjective(pop)
    popSize = size(pop,1); % Population size
    numObj = 2; % Number of objectives
    % initialize scores
    scores = zeros(popSize, numObj);
    % Compute first objective
    scores(:,1) = (pop + 2).^2 - 10;
    % Compute second objective
    scores(:,2) = (pop - 2).^2 + 20;
```

This vectorized version of the fitness function takes a matrix `pop` with an arbitrary number of points, the rows of `pop`, and returns a matrix of size `populationSize-by-numberOfObjectives`.

We need to specify that the fitness function is vectorized using the options created using `optimoptions`. The options are passed in as the ninth argument.

```
FitnessFunction = @(x) vectorized_multiobjective(x);  
options = optimoptions(@gamultiobj, 'UseVectorized', true);  
gamultiobj(FitnessFunction, numberOfVariables, [], [], [], [], lb, ub, options);
```

Optimization terminated: average change in the spread of Pareto solutions less than options.Function

See Also

More About

- “Vectorize the Fitness Function” on page 8-103
- “Genetic Algorithm Options” on page 17-23

Effects of Multiobjective Genetic Algorithm Options

This example shows some of the effects of multiobjective genetic algorithm options. You create and change options for `gamultiobj` using the `optimoptions` function.

Setting Up a Problem for `gamultiobj`

`gamultiobj` finds a local Pareto front for multiple objective functions using the genetic algorithm. For this example, use `gamultiobj` to obtain a Pareto front for two objective functions described in the MATLAB® file `kur_multiobjective.m`. This file represents a real-valued function that consists of two objectives, each of three decision variables. Also impose bound constraints on the decision variables $-5 \leq x(i) \leq 5$, $i = 1, 2, 3$.

```
type kur_multiobjective.m
```

```
function y = kur_multiobjective(x)
%KUR_MULTIOBJECTIVE Objective function for a multiobjective problem.
% The Pareto-optimal set for this two-objective problem is nonconvex as
% well as disconnected. The function KUR_MULTIOBJECTIVE computes two
% objectives and returns a vector y of size 2-by-1.
%
% Reference: Kalyanmoy Deb, "Multi-Objective Optimization using
% Evolutionary Algorithms", John Wiley & Sons ISBN 047187339
%
% Copyright 2007 The MathWorks, Inc.

% Initialize for two objectives
y = zeros(2,1);

% Compute first objective
for i = 1:2
    y(1) = y(1) - 10*exp(-0.2*sqrt(x(i)^2 + x(i+1)^2));
end

% Compute second objective
for i = 1:3
    y(2) = y(2) + abs(x(i))^0.8 + 5*sin(x(i)^3);
end
```

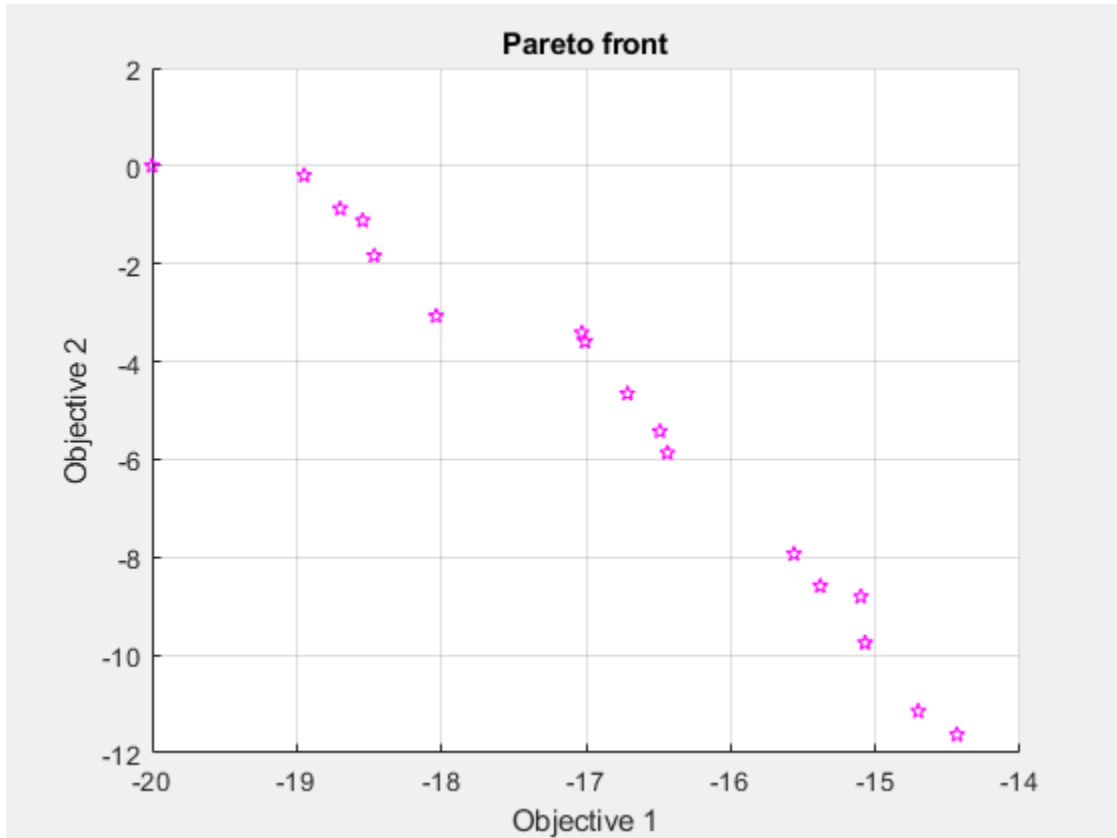
To observe the solver progress, plot the Pareto front in every generation using the plot function `@gaplotpareto`. Specify this plot function in the options by using the `optimoptions` function.

```
FitnessFunction = @kur_multiobjective; % Function handle to the fitness function
numberOfVariables = 3; % Number of decision variables
lb = [-5 -5 -5]; % Lower bound
ub = [5 5 5]; % Upper bound
A = []; % No linear inequality constraints
b = []; % No linear inequality constraints
Aeq = []; % No linear equality constraints
beq = []; % No linear equality constraints
options = optimoptions(@gamultiobj, 'PlotFcn', @gaplotpareto);
```

Run the `gamultiobj` solver and display the number of solutions found on the Pareto front and the number of generations.

```
[x,Fval,exitFlag,Output] = gamultiobj(FitnessFunction,numberOfVariables,A, ...
    b,Aeq,beq,lb,ub,options);
```

Optimization terminated: average change in the spread of Pareto solutions less than options.Function



```
fprintf('The number of points on the Pareto front was: %d\n', size(x,1));
```

The number of points on the Pareto front was: 18

```
fprintf('The number of generations was : %d\n', Output.generations);
```

The number of generations was : 317

The Pareto plot displays two competing objectives. For this problem, the Pareto front is known to be disconnected. The solution from `gamultiobj` can capture the Pareto front even if it is disconnected. Note that when you run this example, your result may be different from the results shown because `gamultiobj` uses random number generators.

Elitist Multiobjective Genetic Algorithm

The multiobjective genetic algorithm (`gamultiobj`) works on a population using a set of operators that are applied to the population. A population is a set of points in the design space. The initial population is generated randomly by default. The next generation of the population is computed using the non-dominated rank and a distance measure of the individuals in the current generation.

A non-dominated rank is assigned to each individual using the relative fitness. Individual 'p' dominates 'q' ('p' has a lower rank than 'q') if 'p' is strictly better than 'q' in at least one objective and

'p' is no worse than 'q' in all objectives. This is the same as saying 'q' is dominated by 'p' or 'p' is non-inferior to 'q'. Two individuals 'p' and 'q' are considered to have equal ranks if neither dominates the other. The distance measure of an individual is used to compare individuals with equal rank. It is a measure of how far an individual is from the other individuals with the same rank.

The multiobjective GA function `gamultiobj` uses a controlled elitist genetic algorithm (a variant of NSGA-II [1]). An elitist GA always favors individuals with better fitness value (rank) whereas, a controlled elitist GA also favors individuals that can help increase the diversity of the population even if they have a lower fitness value. It is very important to maintain the diversity of population for convergence to an optimal Pareto front. This is done by controlling the elite members of the population as the algorithm progresses. Two options 'ParetoFraction' and 'DistanceFcn' are used to control the elitism. The Pareto fraction option limits the number of individuals on the Pareto front (elite members) and the distance function helps to maintain diversity on a front by favoring individuals that are relatively far away on the front.

Specifying Multiobjective GA Options

We can choose the default distance measure function, `distancecrowding`, that is provided in the toolbox or write our own function to calculate the distance measure of an individual. The crowding distance measure function in the toolbox takes an optional argument to calculate distance either in function space (phenotype) or design space (genotype). If 'genotype' is chosen, then the diversity on a Pareto front is based on the design space. The default choice is 'phenotype' and, in that case, the diversity is based on the function space. Here we choose 'genotype' for our distance function. We will directly modify the value of the parameter `DistanceMeasureFcn`.

```
options.DistanceMeasureFcn = {@distancecrowding,'genotype'};
```

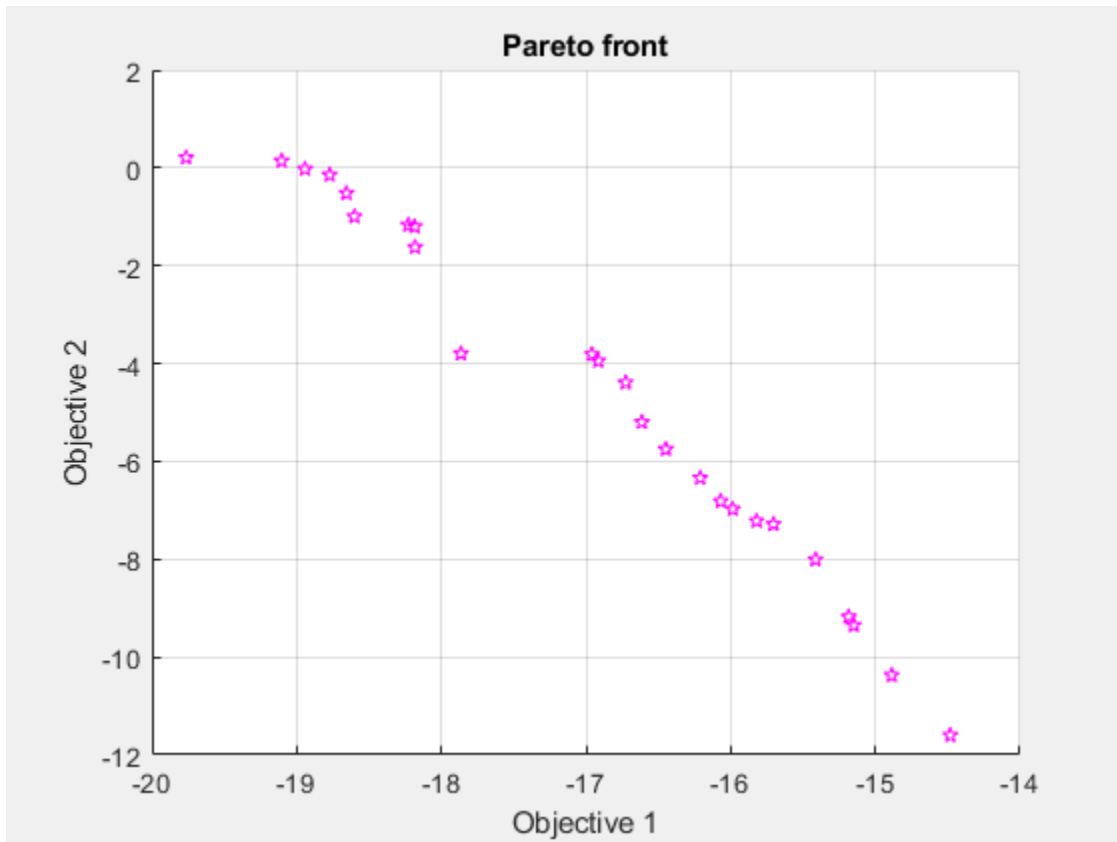
The Pareto fraction has a default value of 0.35 i.e., the solver will try to limit the number of individuals in the current population that are on the Pareto front to 35 percent of the population size. Here we set the Pareto fraction to 0.5.

```
options = optimoptions(options,'ParetoFraction',0.5);
```

Run the `gamultiobj` solver and display the number of solutions found on the Pareto front and the average distance measure of solutions.

```
[x,Fval,exitFlag,Output] = gamultiobj(FitnessFunction,numberOfVariables,A, ...
                                     b,Aeq,beq,lb,ub,options);
```

```
Optimization terminated: average change in the spread of Pareto solutions less than options.Func
```



```
fprintf('The number of points on the Pareto front was: %d\n', size(x,1));
```

The number of points on the Pareto front was: 25

```
fprintf('The average distance measure of the solutions on the Pareto front was: %g\n', Output.averageDistance);
```

The average distance measure of the solutions on the Pareto front was: 0.051005

```
fprintf('The spread measure of the Pareto front was: %g\n', Output.spread);
```

The spread measure of the Pareto front was: 0.181192

A smaller average distance measure indicates that the solutions on the Pareto front are evenly distributed. However, if the Pareto front is disconnected, then the distance measure will not indicate the true spread of solutions.

Modifying the Stopping Criteria

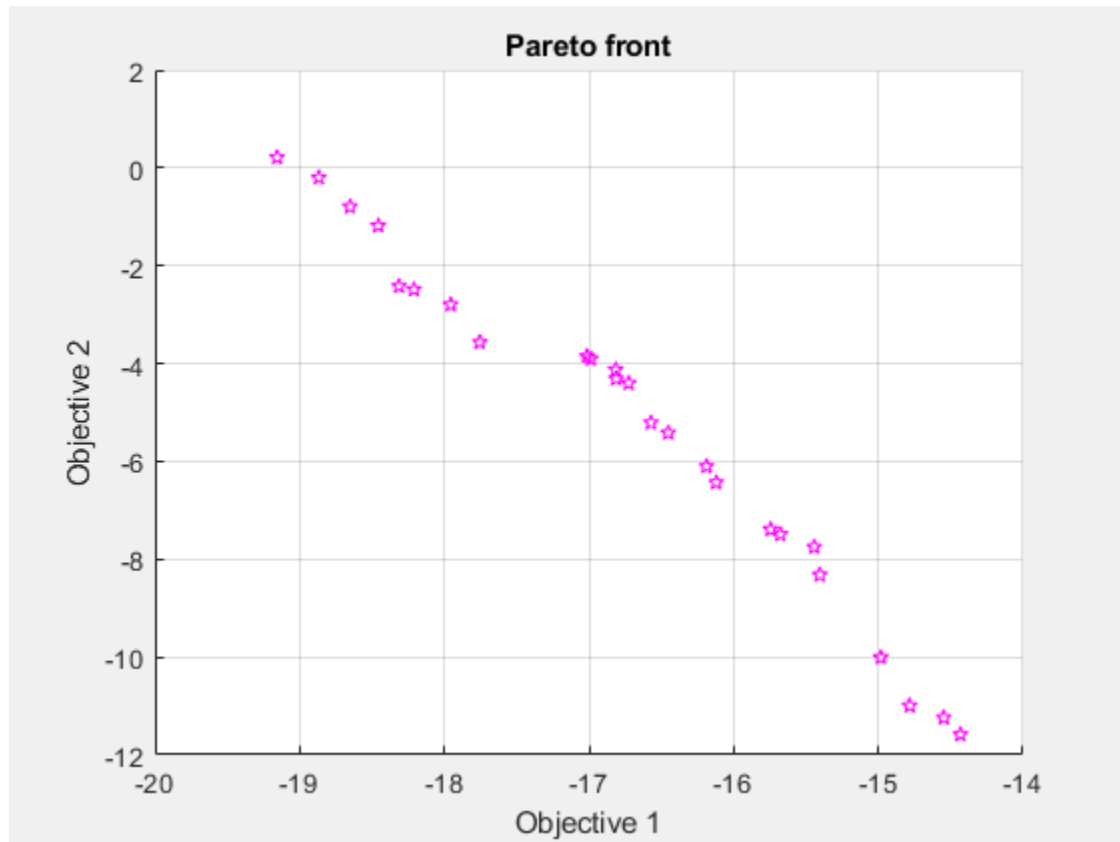
`gamultiobj` uses three different criteria to determine when to stop the solver. The solver stops when any one of the stopping criteria is met. It stops when the maximum number of generations is reached; by default this number is `'200*numberOfVariables'`. `gamultiobj` also stops if the average change in the spread of the Pareto front over the `MaxStallGenerations` generations (default is 100) is less than the tolerance specified in `options.FunctionTolerance`. The third criterion is the maximum time limit in seconds (default is `Inf`). Here we modify the stopping criteria to change the `FunctionTolerance` from `1e-4` to `1e-3` and increase `MaxStallGenerations` to 150.

```
options = optimoptions(options, 'FunctionTolerance', 1e-3, 'MaxStallGenerations', 150);
```


Run the `gamultiobj` solver and display the number of solutions found on the Pareto front and the number of generations.

```
[x,Fval,exitFlag,Output] = gamultiobj(FitnessFunction,numberOfVariables,A, ...
    b,Aeq,beq,lb,ub,options);
```

Optimization terminated: average change in the spread of Pareto solutions less than options.Function



```
fprintf('The number of points on the Pareto front was: %d\n', size(x,1));
```

The number of points on the Pareto front was: 25

```
fprintf('The number of generations was : %d\n', Output.generations);
```

The number of generations was : 152

Multiobjective GA Hybrid Function

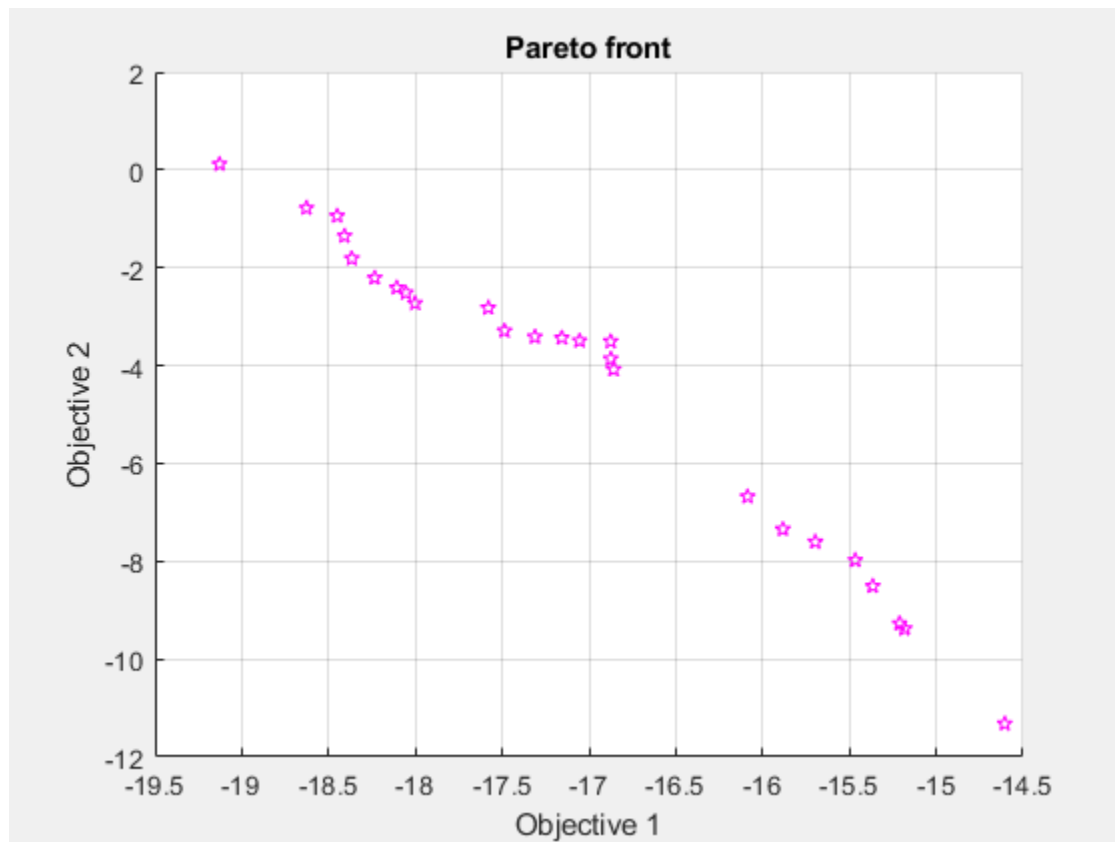
We will use a hybrid scheme to find an optimal Pareto front for our multiobjective problem. `gamultiobj` can reach the region near an optimal Pareto front relatively quickly, but it can take many function evaluations to achieve convergence. A commonly used technique is to run `gamultiobj` for a small number of generations to get near an optimum front. Then the solution from `gamultiobj` is used as an initial point for another optimization solver that is faster and more efficient for a local search. We use `fgoalattain` as the hybrid solver with `gamultiobj`. `fgoalattain` solves the goal attainment problem, which is one formulation for minimizing a multiobjective optimization problem.

The hybrid functionality in multiobjective function `gamultiobj` is slightly different from that of the single objective function GA. In single objective GA the hybrid function starts at the best point returned by GA. However, in `gamultiobj` the hybrid solver will start at all the points on the Pareto front returned by `gamultiobj`. The new individuals returned by the hybrid solver are combined with the existing population and a new Pareto front is obtained. It may be useful to see the syntax of `fgoalattain` function to better understand how the output from `gamultiobj` is internally converted to the input of `fgoalattain` function. `gamultiobj` estimates the pseudo weights (required input for `fgoalattain`) for each point on the Pareto front and runs the hybrid solver starting from each point on the Pareto front. Another required input, `goal`, is a vector specifying the goal for each objective. `gamultiobj` provides this input as the extreme points from the Pareto front found so far.

Here we run `gamultiobj` without the hybrid function.

```
[x,Fval,exitFlag,Output] = gamultiobj(FitnessFunction,numberOfVariables,A, ...
    b,Aeq,beq,lb,ub,options);
```

Optimization terminated: average change in the spread of Pareto solutions less than options.Function



```
fprintf('The number of points on the Pareto front was: %d\n', size(x,1));
```

The number of points on the Pareto front was: 25

```
fprintf('The average distance measure of the solutions on the Pareto front was: %g\n', Output.averageDistance);
```

The average distance measure of the solutions on the Pareto front was: 0.0434477

```
fprintf('The spread measure of the Pareto front was: %g\n', Output.spread);
```

The spread measure of the Pareto front was: 0.17833

Here we use `fgoalattain` as the hybrid function. We also reset the random number generators so that we can compare the results with the previous run (without the hybrid function).

```
options = optimoptions(options, 'HybridFcn', @fgoalattain);
```

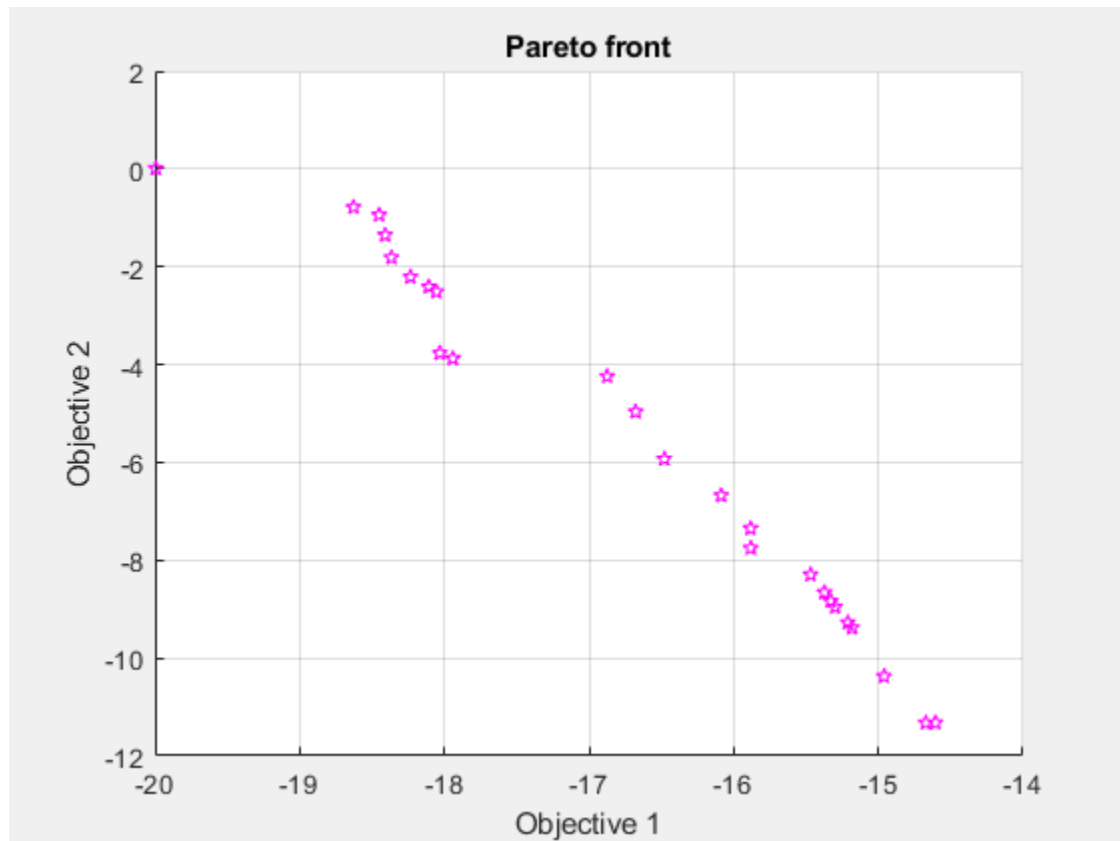
Reset the random state (only to compare with previous run)

```
strm = RandStream.getGlobalStream;
strm.State = Output.rngstate.State;
```

Run the GAMULTIOBJ solver with hybrid function.

```
[x,Fval,exitFlag,Output,Population,Score] = gamultiobj(FitnessFunction,numberOfVariables,A, ...
    b,Aeq,beq,lb,ub,options);
```

Optimization terminated: average change in the spread of Pareto solutions less than options.Function



```
fprintf('The number of points on the Pareto front was: %d\n', size(x,1));
```

The number of points on the Pareto front was: 25

```
fprintf('The average distance measure of the solutions on the Pareto front was: %g\n', Output.averageDistance);
```

The average distance measure of the solutions on the Pareto front was: 0.127964

```
fprintf('The spread measure of the Pareto front was: %g\n', Output.spread);
```

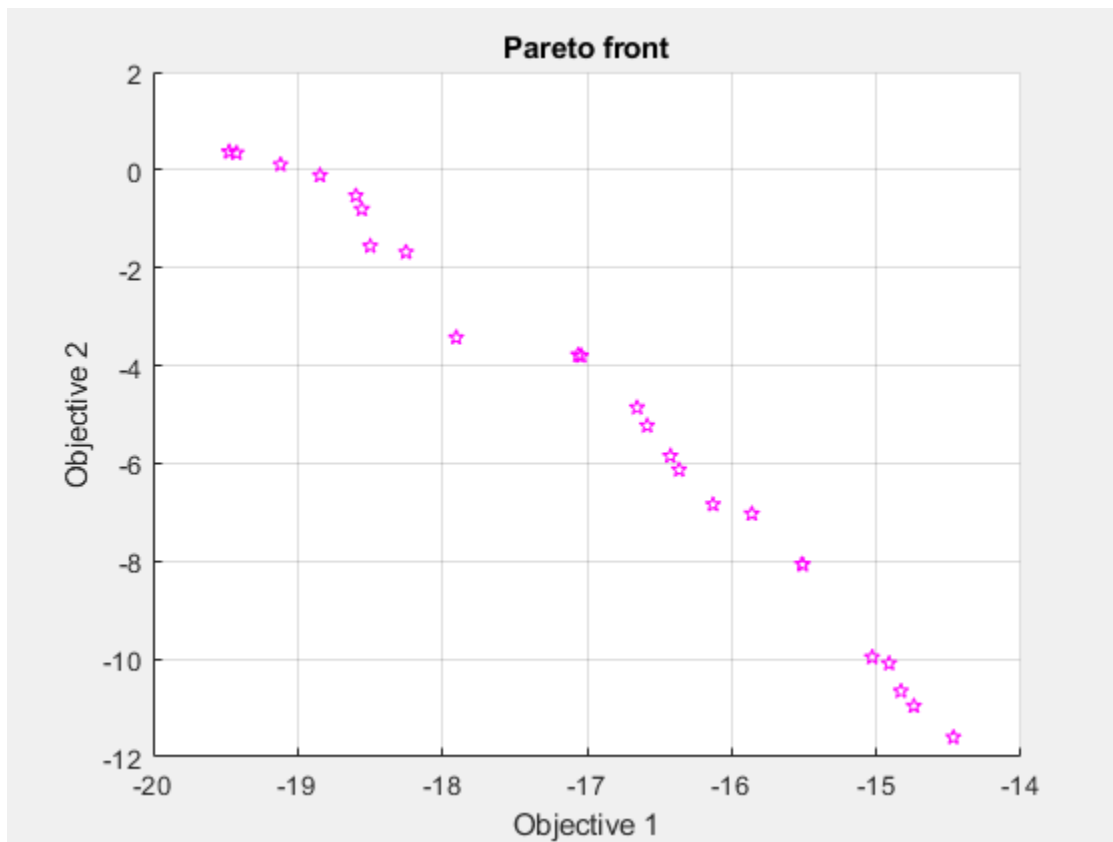
The spread measure of the Pareto front was: 0.421195

If the Pareto fronts obtained by `gamultiobj` alone and by using the hybrid function are close, we can compare them using the spread and the average distance measures. The average distance of the solutions on the Pareto front can be improved by using a hybrid function. The spread is a measure of the change in two fronts and that can be higher when hybrid function is used. This indicates that the front has changed considerably from that obtained by `gamultiobj` with no hybrid function.

It is certain that using the hybrid function will result in an optimal Pareto front but we may lose the diversity of the solution (because `fgoalattain` does not try to preserve the diversity). This can be indicated by a higher value of the average distance measure and the spread of the front. We can further improve the average distance measure of the solutions and the spread of the Pareto front by running `gamultiobj` again with the final population returned in the last run. Here, we should remove the hybrid function.

```
options = optimoptions(options,'HybridFcn',[]); % No hybrid function
% Provide initial population and scores
options = optimoptions(options,'InitialPopulationMatrix',Population,'InitialScoresMatrix',Score)
% Run the GAMULTIOBJ solver with hybrid function.
[x,Fval,exitFlag,Output,Population,Score] = gamultiobj(FitnessFunction,numberOfVariables,A, ...
    b,Aeq,beq,lb,ub,options);
```

Optimization terminated: average change in the spread of Pareto solutions less than options.Function



```
fprintf('The number of points on the Pareto front was: %d\n', size(x,1));
```

The number of points on the Pareto front was: 25

```
fprintf('The average distance measure of the solutions on the Pareto front was: %g\n', Output.averageDistance);
```

The average distance measure of the solutions on the Pareto front was: 0.0518031

```
fprintf('The spread measure of the Pareto front was: %g\n', Output.spread);
```

The spread measure of the Pareto front was: 0.306712

References

[1] Kalyanmoy Deb, "Multi-Objective Optimization using Evolutionary Algorithms", John Wiley & Sons ISBN 047187339.

See Also

More About

- "Genetic Algorithm Options" on page 17-23
- "Hybrid Scheme in the Genetic Algorithm" on page 8-95

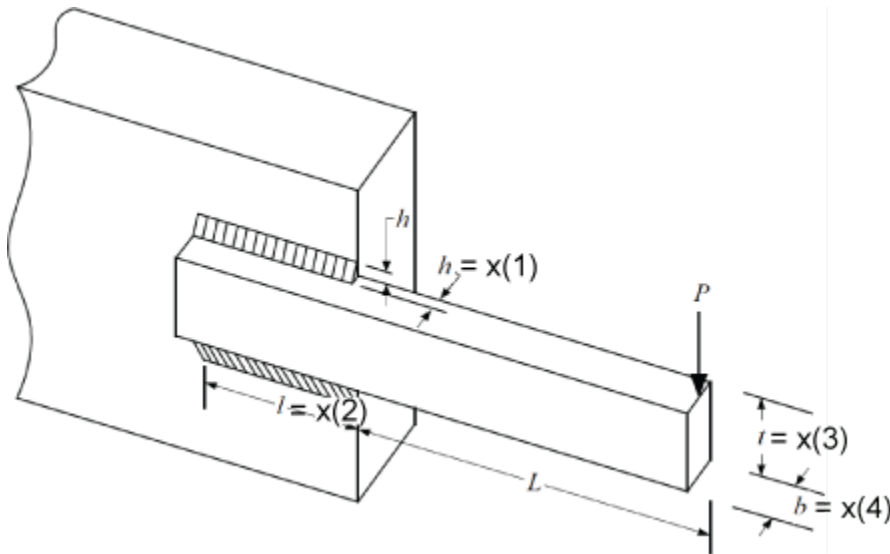
Design Optimization of a Welded Beam

This example shows how to examine the tradeoff between the strength and cost of a beam. Several publications use this example as a test problem for various multiobjective algorithms, including Deb et al. [1] and Ray and Liew [2].

For a video overview of this example, see Pareto Sets for Multiobjective Optimization.

Problem Description

The following sketch is adapted from Ray and Liew [2].



This sketch represents a beam welded onto a substrate. The beam supports a load P at a distance L from the substrate. The beam is welded onto the substrate with upper and lower welds, each of length l and thickness h . The beam has a rectangular cross-section, width b , and height t . The material of the beam is steel.

The two objectives are the fabrication cost of the beam and the deflection of the end of the beam under the applied load P . The load P is fixed at 6,000 lbs, and the distance L is fixed at 14 in.

The design variables are:

- $x(1) = h$, the thickness of the welds
- $x(2) = l$, the length of the welds
- $x(3) = t$, the height of the beam
- $x(4) = b$, the width of the beam

The fabrication cost of the beam is proportional to the amount of material in the beam, $(l + L)tb$, plus the amount of material in the welds, lh^2 . Using the proportionality constants from the cited papers, the first objective is

$$F_1(x) = 1.10471x_1^2x_2 + 0.04811x_3x_4(14 + x_2).$$

The deflection of the beam is proportional to P and inversely proportional to bt^3 . Again, using the proportionality constants from the cited papers, the second objective is

$$F_2(x) = \frac{P}{x_4 x_3^3} C, \text{ where } C = \frac{4(14)^3}{30 \times 10^6} \approx 3.6587 \times 10^{-4} \text{ and } P = 6,000.$$

The problem has several constraints.

- The weld thickness cannot exceed the beam width. In symbols, $x(1) \leq x(4)$. In toolbox syntax:

```
Aineq = [1,0,0,-1];
bineq = 0;
```

- The shear stress $\tau(x)$ on the welds cannot exceed 13,600 psi. To calculate the shear stress, first calculate preliminary expressions:

$$\tau_1 = \frac{1}{\sqrt{2}x_1x_2}$$

$$R = \sqrt{x_2^2 + (x_1 + x_3)^2}$$

$$\tau_2 = \frac{(L + x_2/2)R}{\sqrt{2}x_1x_3(x_2^2/3 + (x_1 + x_3)^2)}$$

$$\tau(x) = P\sqrt{\tau_1^2 + \tau_2^2 + \frac{2\tau_1\tau_2x_2}{R}}.$$

In summary, the shear stress on the welds has the constraint $\tau(x) \leq 13600$.

- The normal stress $\sigma(x)$ on the welds cannot exceed 30,000 psi. The normal stress is

$$P\frac{6L}{x_4x_3^2} \leq 30 \times 10^3.$$

- The buckling load capacity in the vertical direction must exceed the applied load of 6,000 lbs. Using the values of Young's modulus $E = 30 \times 10^6$ psi and $G = 12 \times 10^6$ psi, the buckling load

$$\text{constraint is } \frac{4.013Ex_3x_4^3}{6L^2} \left(1 - \frac{x_3}{2L}\sqrt{\frac{E}{4G}}\right) \geq 6000. \text{ Numerically, this becomes the inequality}$$

$$64,746.022(1 - 0.0282346x_3)x_3x_4^3 \geq 6000.$$

- The bounds on the variables are $0.125 \leq x(1) \leq 5$, $0.1 \leq x(2) \leq 10$, $0.1 \leq x(3) \leq 10$, and $0.125 \leq x(4) \leq 5$. In toolbox syntax:

```
lb = [0.125,0.1,0.1,0.125];
ub = [5,10,10,5];
```

The objective functions appear at the end of this example in the function `objval(x)`. The nonlinear constraints appear at the end of this example in the function `nonlcon(x)`.

Multiobjective Problem Formulation and paretosearch Solution

You can optimize this problem in several ways:

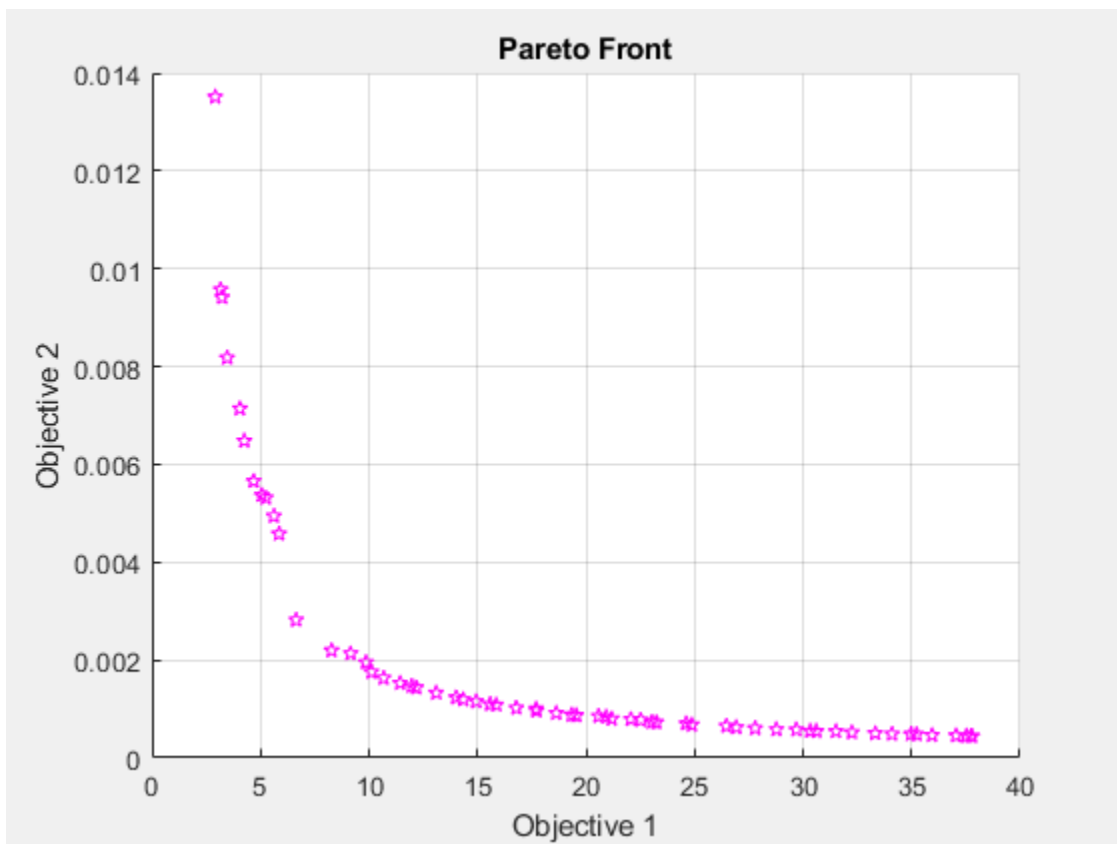
- Set a maximum deflection, and find a single-objective minimal fabrication cost over designs that satisfy the maximum deflection constraint.
- Set a maximum fabrication cost, and find a single-objective minimal deflection over designs that satisfy the fabrication cost constraint.
- Solve a multiobjective problem, visualizing the tradeoff between the two objectives.

To use the multiobjective approach, which gives more information about the problem, set the objective function and nonlinear constraint function.

```
fun = @objval;
nlcon = @nonlcon;
```

Solve the problem using `paretosearch` with the `'psplotparetof'` plot function. To reduce the amount of diagnostic display information, set the `Display` option to `'off'`.

```
opts_ps = optimoptions('paretosearch', 'Display', 'off', 'PlotFcn', 'psplotparetof');
rng default % For reproducibility
[x_psl, fval_psl, ~, psoutput1] = paretosearch(fun, 4, Aineq, bineq, [], [], lb, ub, nlcon, opts_ps);
```

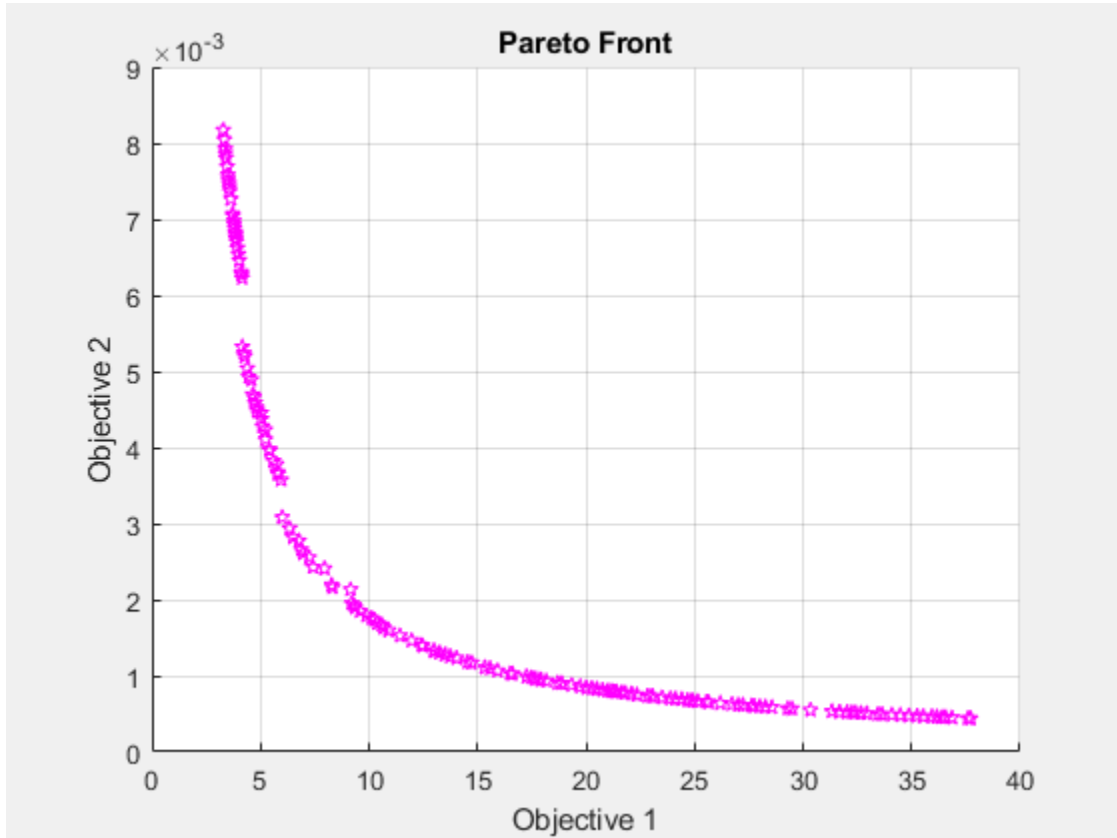


```
disp("Total Function Count: " + psoutput1.funccount);
```

Total Function Count: 1870

For a smoother Pareto front, try using more points.


```
npts = 160; % The default is 60
opts_ps.ParetoSetSize = npts;
[x_ps2,fval_ps2,~,psoutput2] = paretosearch(fun,4,Aineq,bineq,[],[],lb,ub,nlcon,opts_ps);
```



```
disp("Total Function Count: " + psoutput2.funccount);
```

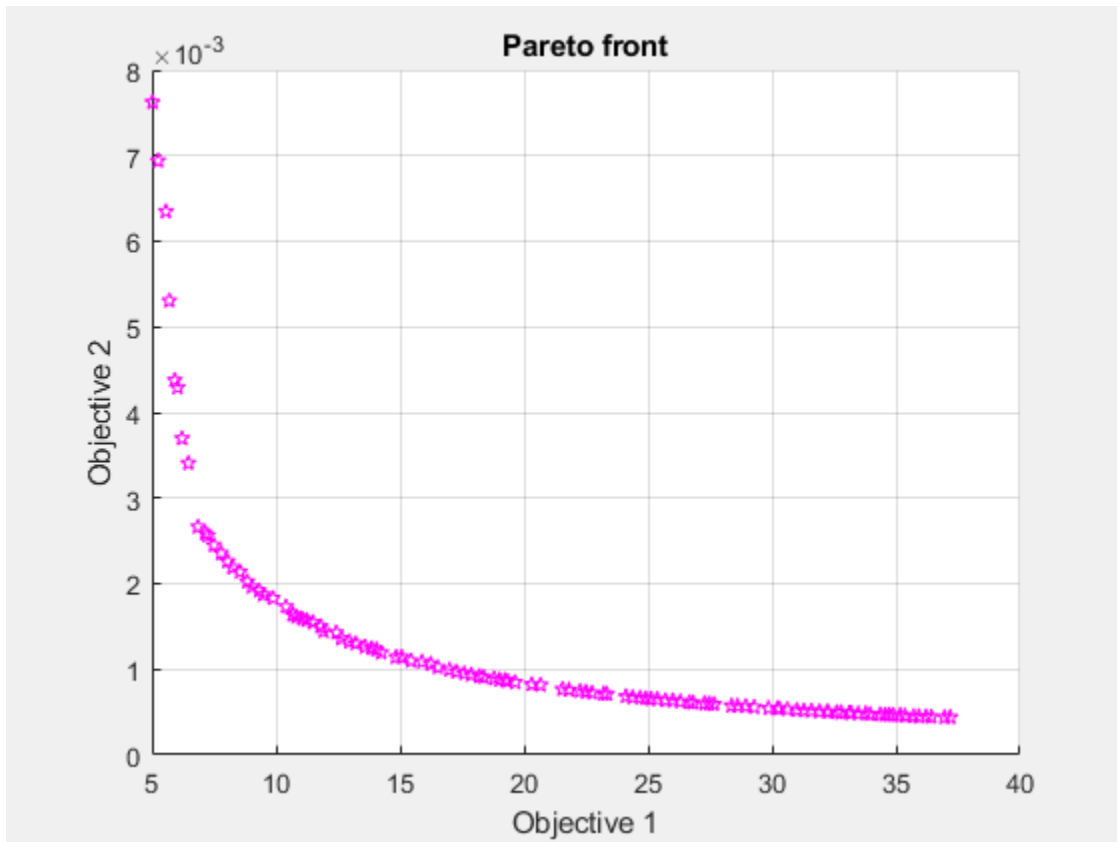
Total Function Count: 6254

This solution looks like a smoother curve, but it has a smaller extent of Objective 2. The solver takes over three times as many function evaluations when using 160 Pareto points instead of 60.

gamultiobj Solution

To see if the solver makes a difference, try the `gamultiobj` solver on the problem. Set equivalent options as in the previous solution. Because the `gamultiobj` solver keeps fewer than half of its solutions on the best Pareto front, use two times as many points as before.

```
opts_ga = optimoptions('gamultiobj','Display','off','PlotFcn','gaplotpareto','PopulationSize',2*
[x_gal,fval_gal,~,gaoutput1] = gamultiobj(fun,4,Aineq,bineq,[],[],lb,ub,nlcon,opts_ga);
```



```
disp("Total Function Count: " + gaoutput1.funccount);
```

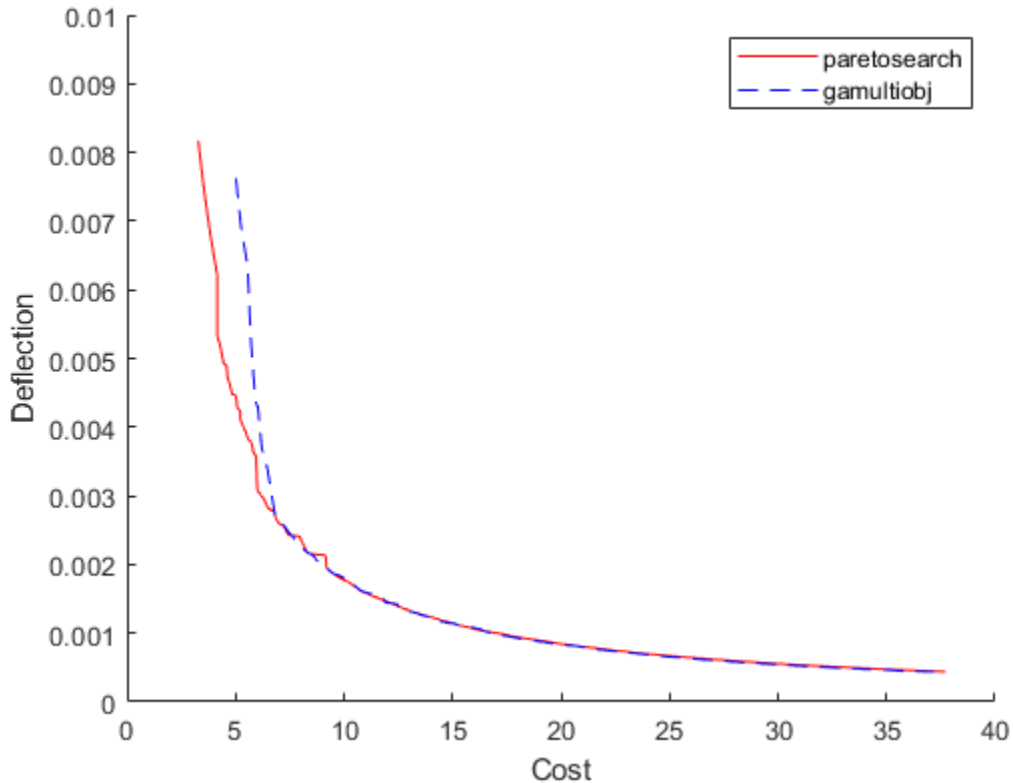
```
Total Function Count: 38401
```

`gamultiobj` takes tens of thousands of function evaluations, whereas `paretosearch` takes only thousands.

Compare Solutions

The `gamultiobj` solution seems to differ from the `paretosearch` solution, although it is difficult to tell because the plotted scales differ. Plot the two solutions on the same plot, using the same scale.

```
fps2 = sortrows(fval_ps2,1,'ascend');
figure
hold on
plot(fps2(:,1),fps2(:,2),'r-')
fga = sortrows(fval_ga1,1,'ascend');
plot(fga(:,1),fga(:,2),'b--')
xlim([0,40])
ylim([0,1e-2])
legend('paretosearch','gamultiobj')
xlabel 'Cost'
ylabel 'Deflection'
hold off
```



The `gamultiobj` solution is better in the rightmost portion of the plot, whereas the `paretosearch` solution is better in the leftmost portion. `paretosearch` uses many fewer function evaluations to obtain its solution.

Typically, when the problem has no nonlinear constraints, `paretosearch` is at least as accurate as `gamultiobj`. However, the resulting Pareto sets can have somewhat different ranges. In this case, the presence of a nonlinear constraint causes the `paretosearch` solution to be less accurate over part of the range.

One of the main advantages of `paretosearch` is that it usually takes many fewer function evaluations.

Start from Single-Objective Solutions

To help the solvers find better solutions, start them from points that are the solutions to minimizing the individual objective functions. The `pickindex` function returns a single objective from the `objval` function. Use `fmincon` to find single-objective optima. Then use those solutions as initial points for a multiobjective search.

```
x0 = zeros(2,4);
x0f = (lb + ub)/2;
opts_fmc = optimoptions('fmincon','Display','off','MaxFunctionEvaluations',1e4);
x0(1,:) = fmincon(@(x)pickindex(x,1),x0f,Aineq,bineq,[],[],lb,ub,@nonlcon,opts_fmc);
x0(2,:) = fmincon(@(x)pickindex(x,2),x0f,Aineq,bineq,[],[],lb,ub,@nonlcon,opts_fmc);
```

Examine the single-objective optima.

```
objval(x0(1,:))
ans = 1x2
    2.3810    0.0158
```

```
objval(x0(2,:))
ans = 1x2
    76.7188    0.0004
```

The minimum cost is 2.381, which gives a deflection of 0.158. The minimum deflection is 0.0004, which has a cost of 76.7253. The plotted curves are quite steep near the ends of their ranges, meaning you get much less deflection if you take a cost a bit above its minimum, or much less cost if you take a deflection a bit above its minimum.

Start `paretosearch` from the single-objective solutions. Because you will plot the solutions later on the same plot, remove the `paretosearch` plot function.

```
opts_ps.InitialPoints = x0;
opts_ps.PlotFcn = [];
[x_ps,x0,fval_ps1x0,~,psoutput1x0] = paretosearch(fun,4,Aineq,bineq,[],[],lb,ub,nlcon,opts_ps);
disp("Total Function Count: " + psoutput1x0.funccount);
```

```
Total Function Count: 4839
```

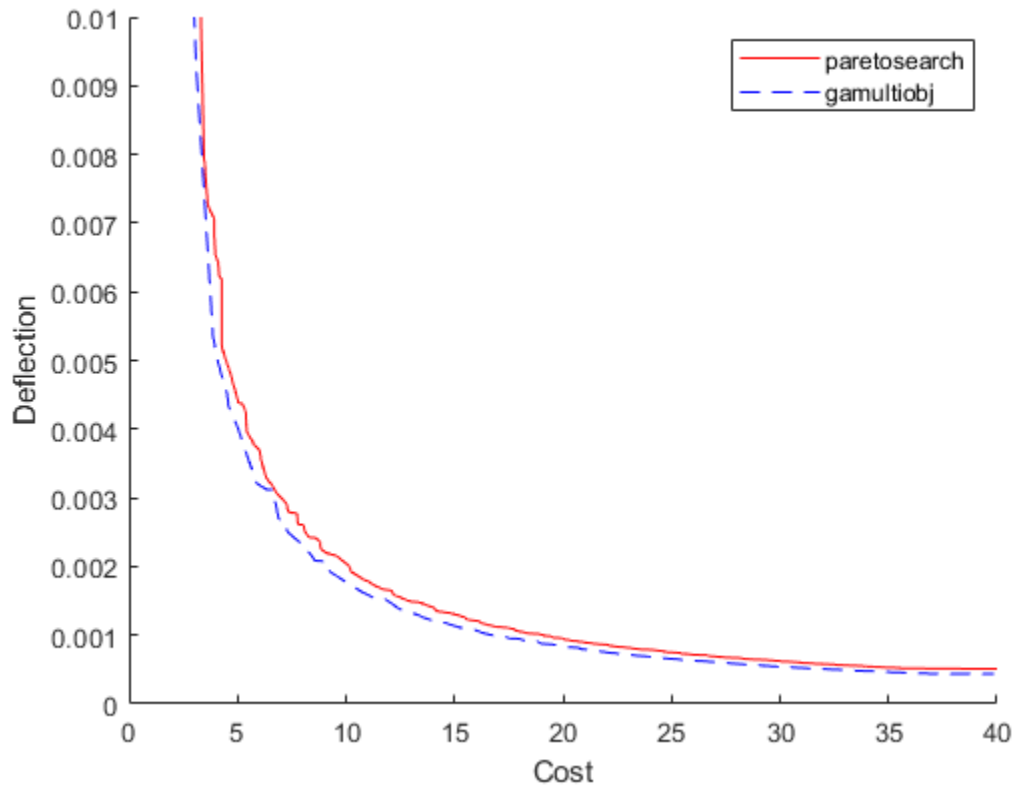
Start `ga` from the same initial points, and remove its plot function.

```
opts_ga.InitialPopulationMatrix = x0;
opts_ga.PlotFcn = [];
[~,fval_ga,~,gaoutput] = gamultiobj(fun,4,Aineq,bineq,[],[],lb,ub,nlcon,opts_ga);
disp("Total Function Count: " + gaoutput.funccount);
```

```
Total Function Count: 37441
```

Plot the solutions on the same axes.

```
fps = sortrows(fval_ps1x0,1,'ascend');
figure
hold on
plot(fps(:,1),fps(:,2),'r-')
fga = sortrows(fval_ga,1,'ascend');
plot(fga(:,1),fga(:,2),'b--')
xlim([0,40])
ylim([0,1e-2])
legend('paretosearch','gamultiobj')
xlabel 'Cost'
ylabel 'Deflection'
hold off
```



By starting from the single-objective solutions, the `gamultiobj` solution is slightly better than the `paretosearch` solution throughout the plotted range. However, `gamultiobj` takes almost ten times as many function evaluations to reach its solution.

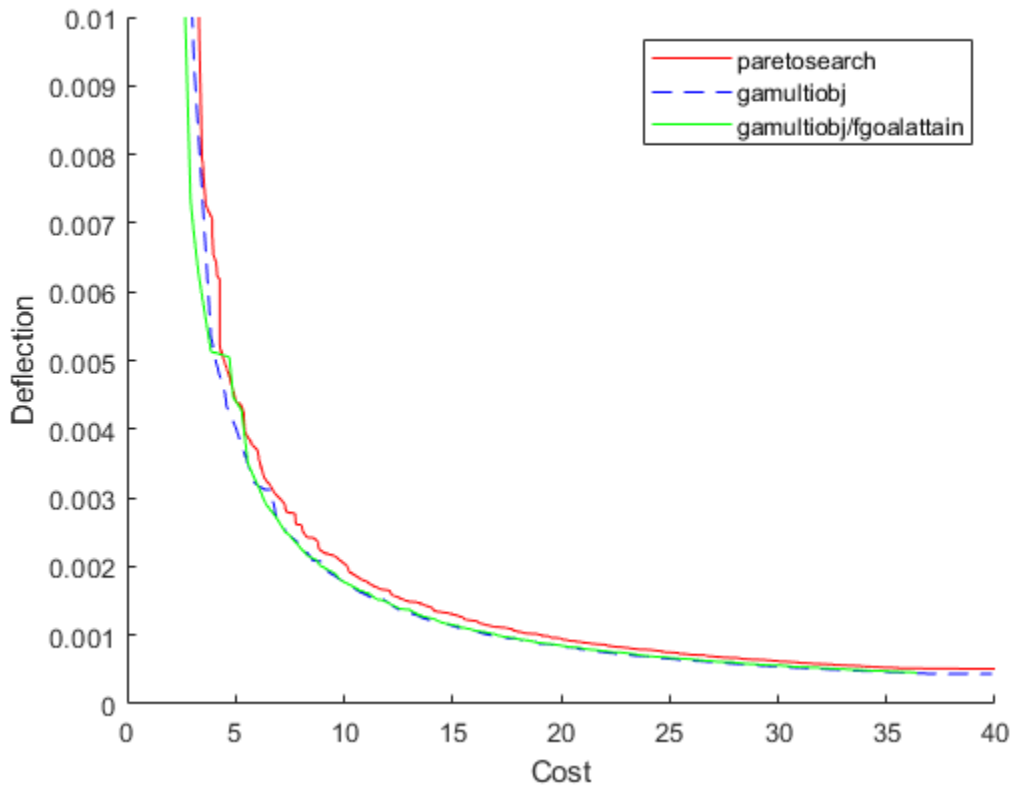
Hybrid Function

`gamultiobj` can call the hybrid function `fgoalattain` automatically to attempt to reach a more accurate solution. See whether the hybrid function improves the solution.

```
opts_ga.HybridFcn = 'fgoalattain';
[xgah,fval_gah,~,gaoutputh] = gamultiobj(fun,4,Aineq,bineq,[],[],lb,ub,nlcon,opts_ga);
disp("Total Function Count: " + gaoutputh.funccount);
```

Total Function Count: 57478

```
fgah = sortrows(fval_gah,1,'ascend');
figure
hold on
plot(fps(:,1),fps(:,2),'r-')
plot(fga(:,1),fga(:,2),'b--')
plot(fgah(:,1),fgah(:,2),'g-')
xlim([0,40])
ylim([0,1e-2])
legend('paretosearch','gamultiobj','gamultiobj/fgoalattain')
xlabel 'Cost'
ylabel 'Deflection'
hold off
```



The hybrid function provides a slight improvement on the `gamultiobj` solution, mainly in the leftmost part of the plot.

Run `fgoalattain` Manually from `paretosearch` Solution Points

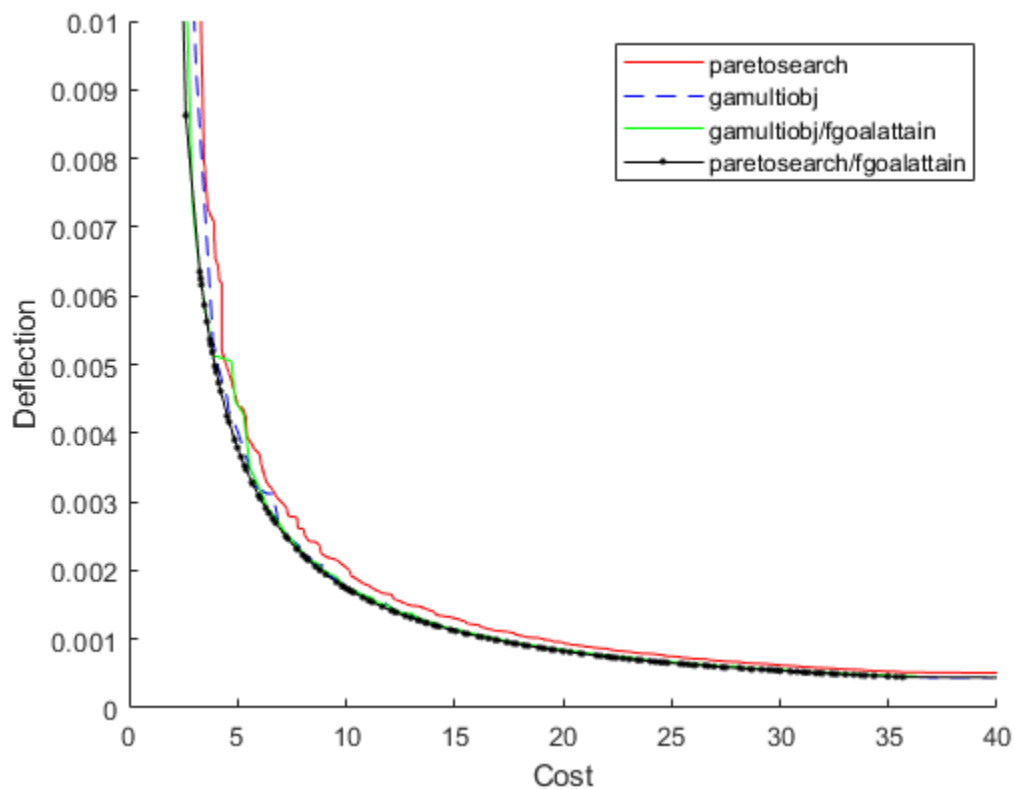
Although `paretosearch` has no built-in hybrid function, you can improve the `paretosearch` solution by running `fgoalattain` from the `paretosearch` solution points. Create a goal and weights for `fgoalattain` by using the same setup for `fgoalattain` as described in “`gamultiobj` Hybrid Function” on page 17-40.

```
Fmax = max(fval_pslx0);
nobj = numel(Fmax);
Fmin = min(fval_pslx0);
w = sum((Fmax - fval_pslx0)./(1 + Fmax - Fmin),2);
p = w.*((Fmax - fval_pslx0)./(1 + Fmax - Fmin));
xnew = zeros(size(x_psx0));
nsol = size(xnew,1);
fvalnew = zeros(nsol,nobj);
opts_fg = optimoptions('fgoalattain','Display','off');
nfv = 0;
for ii = 1:nsol
    [xnew(ii,:),fvalnew(ii,:),~,~,output] = fgoalattain(fun,x_psx0(ii,:),fval_pslx0(ii,:),p(ii,:),
        Aineq,bineq,[],[],lb,ub,nlcon,opts_fg);
    nfv = nfv + output.funcCount;
end
disp("fgoalattain Function Count: " + nfv)
fgoalattain Function Count: 14049
```

```

fnew = sortrows(fvalnew,1,'ascend');
figure
hold on
plot(fps(:,1),fps(:,2),'r-')
plot(fga(:,1),fga(:,2),'b--')
plot(fgah(:,1),fgah(:,2),'g-')
plot(fnew(:,1),fnew(:,2),'k.-')
xlim([0,40])
ylim([0,1e-2])
legend('paretosearch','gamultiobj','gamultiobj/fgoalattain','paretosearch/fgoalattain')
xlabel 'Cost'
ylabel 'Deflection'

```

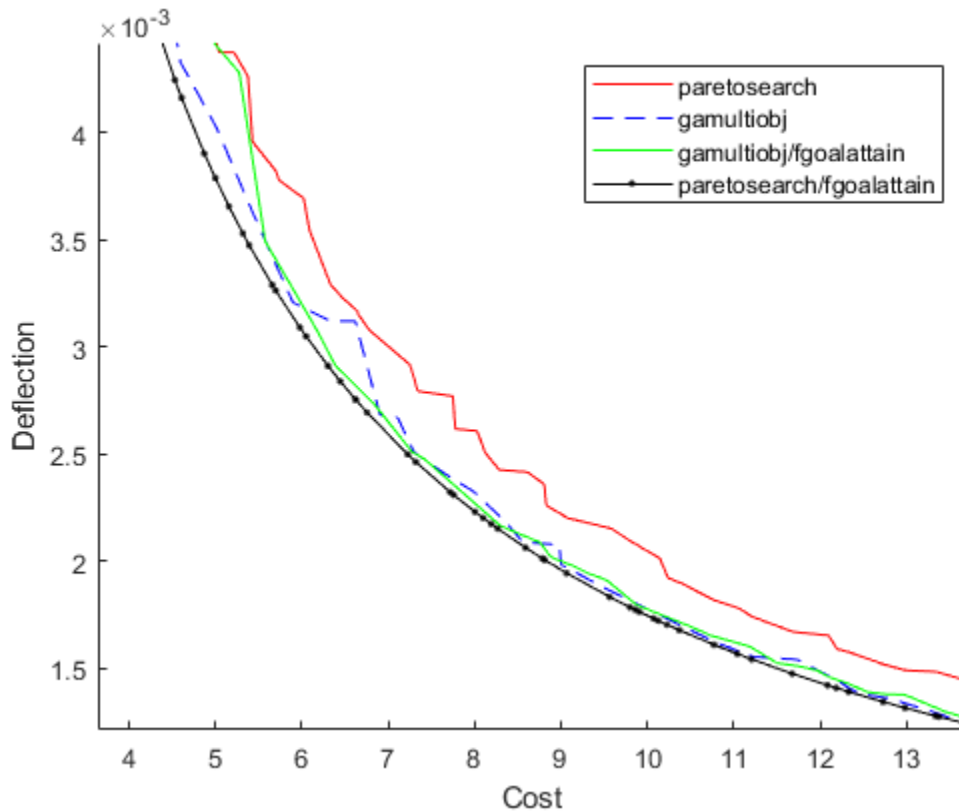


The combination of paretosearch and fgoalattain creates the most accurate Pareto front. Zoom in to see.

```

xlim([3.64 13.69])
ylim([0.00121 0.00442])
hold off

```



Even with the extra `fgoalattain` computations, the total function count for the combination is less than half of the function count for the `gamultiobj` solution alone.

```
fprintf("Total function count for gamultiobj alone is %d.\n" + ...
       "For paretosearch and fgoalattain together it is %d.\n",...
       gaoutput.funccount,nfv + psoutput1x0.funccount)
```

```
Total function count for gamultiobj alone is 37441.
For paretosearch and fgoalattain together it is 18888.
```

Find Good Parameters from Plot

The plotted points show the best values in function space. To determine which parameters achieve these function values, find the size of the beam and size of the weld in order to get a particular cost/deflection point. For example, the plot of `paretosearch` followed by `fgoalattain` shows points with a cost of about 6 and a deflection of about 3.5×10^{-3} . Determine the sizes of the beam and weld that achieve these points.

```
whichgood = find(fvalnew(:,1) <= 6 & fvalnew(:,2) <= 3.5e-3);
goodpoints = table(xnew(whichgood,:),fvalnew(whichgood,:), 'VariableNames', {'Parameters' 'Objectives'})
```

```
goodpoints=4x2 table
```

Parameters			Objectives		
0.63457	1.5187	10	0.67262	5.6974	0.0032637
0.61635	1.5708	10	0.63165	5.391	0.0034753
0.63228	1.5251	10	0.6674	5.6584	0.0032892

0.65077 1.4751 10 0.70999 5.976 0.0030919

Four sets of parameters achieve a cost of less than 6 and a deflection of less than $3.5e-3$:

- Weld thickness slightly over 0.6
- Weld length about 1.5
- Beam height 10 (the upper bound)
- Beam width between 0.63 and 0.71

Objective and Nonlinear Constraints

```
function [Cineq,Ceq] = nonlcon(x)
sigma = 5.04e5 ./ (x(:,3).^2 .* x(:,4));
P_c = 64746.022*(1 - 0.028236*x(:,3)).*x(:,3).*x(:,4).^3;
tp = 6e3./sqrt(2)./(x(:,1).*x(:,2));
tpp = 6e3./sqrt(2) .* (14+0.5*x(:,2)).*sqrt(0.25*(x(:,2).^2 + (x(:,1) + x(:,3)).^2)) ./ (x(:,1) .* x(:,2));
tau = sqrt(tp.^2 + tpp.^2 + (x(:,2).*tp.*tpp)./sqrt(0.25*(x(:,2).^2 + (x(:,1) + x(:,3)).^2)));
Cineq = [tau - 13600,sigma - 3e4,6e3 - P_c];
Ceq = [];
end
```

```
function F = objval(x)
f1 = 1.10471*x(:,1).^2.*x(:,2) + 0.04811*x(:,3).*x(:,4).*(14.0+x(:,2));
f2 = 2.1952./(x(:,3).^3 .* x(:,4));
```

```
F = [f1,f2];
end
```

```
function z = pickindex(x,k)
    z = objval(x); % evaluate both objectives
    z = z(k); % return objective k
end
```

References

[1] Deb, Kalyanmoy, J. Sundar, Udaya Bhaskara Rao N, and Shamik Chaudhuri. *Reference Point Based Multi-Objective Optimization Using Evolutionary Algorithms*. International Journal of Computational Intelligence Research, Vol. 2, No. 3, 2006, pp. 273-286. Available at <https://www.softcomputing.net/ijcir/vol2-issu3-paper4.pdf>

[2] Ray, T., and K. M. Liew. *A Swarm Metaphor for Multiobjective Design Optimization*. Engineering Optimization 34, 2002, pp.141-153.

See Also

More About

- “Multiobjective Optimization”
- Pareto Sets for Multiobjective Optimization

Problem-Based Multiobjective Optimization

- “Steps for Problem-Based Multiobjective Optimization” on page 15-2
- “Pareto Front for Multiobjective Optimization, Problem-Based” on page 15-5
- “Plan Nuclear Fuel Disposal Using Multiobjective Optimization” on page 15-10

Steps for Problem-Based Multiobjective Optimization

In this section...

“Specify Multiple Objective Functions” on page 15-2
 “Specify Multiple Objective Senses (Maximize or Minimize)” on page 15-2
 “Data Format of Multiobjective Solutions” on page 15-3
 “Supply Initial Points for Multiobjective Problem” on page 15-3
 “Hybrid Function” on page 15-3
 “View Pareto Set” on page 15-3

This topic shows how to set up a multiobjective optimization in the problem-based approach, and details the format of results and initial points. For an example, see “Pareto Front for Multiobjective Optimization, Problem-Based” on page 15-5.

Specify Multiple Objective Functions

Specify multiple objective functions in one of two ways:

- Optimization expression — Give an optimization expression that has vector or array values. For example, this objective function returns a vector of three values:

```
prob.Objective = [sin(x),cos(x),1 - x.^2];
```

- Structure — Give a structure of optimization expressions, each of which evaluates to a scalar. For example, this objective function returns a structure with three objective components:

```
prob.Objective.sin = sin(x);
prob.Objective.cos = cos(x);
prob.Objective.quad = 1 - x^2;
```

Specify Multiple Objective Senses (Maximize or Minimize)

Specify an objective function sense, meaning maximize or minimize, depending on how you specify the objective function.

- Objective is an optimization expression — All objectives in the problem have the same objective sense. For example,

```
prob.ObjectiveSense = "max";
```

- Objective is a structure — Each objective function can have its own sense. The `prob.ObjectiveSense` structure has the same fields as the `prob.Objective` structure. For example,

```
prob.ObjectiveSense.sin = "minimize";
prob.ObjectiveSense.cos = "maximize";
prob.ObjectiveSense.quad = "max";
```

The default sense is minimize.

Data Format of Multiobjective Solutions

The returned `sol` output is a vector of `OptimizationValues` objects. Each object contains the values of the optimization variables and the objective functions at one point on the Pareto front. If the problem has nonlinear constraints, `sol` also contains the nonlinear constraint violations at each solution point.

The returned `fval` output is a matrix where each row represents one solution point and each column represents one objective function. The `fval` output is numeric, unlike the `sol` output. You can obtain the objective function values from the `sol` object. However, you can find the values more easily in `fval`.

You can plot the resulting Pareto front in two or three dimensions by calling `paretoplot` on `sol`. For an example, see “Pareto Front for Multiobjective Optimization, Problem-Based” on page 15-5.

Supply Initial Points for Multiobjective Problem

Specifying initial points for multiobjective problems is optional. However, you can sometimes obtain better solutions by doing so. For an example showing the benefit, see “Pareto Front for Multiobjective Optimization, Problem-Based” on page 15-5.

To specify initial points, create an `OptimizationValues` object using the `optimvalues` function. For examples, see the `optimvalues` reference page.

Hybrid Function

To obtain more accurate solutions, the `gamultiobj` solver can optionally call `fgoalattain`. For an example, see “Design Optimization of a Welded Beam” on page 14-62. To use this hybrid function in the problem-based workflow, set the `HybridFcn` option to “`fgoalattain`”:

```
options = optimoptions('gamultiobj',HybridFcn="fgoalattain");
```

Include the solver and options arguments in the `solve` call:

```
[sol,fval,exitflag,output] = solve(prob,...
    Solver="gamultiobj",...
    Options=options);
```

View Pareto Set

To view the Pareto set in two or three dimensions while the solver proceeds, set a plot option.

- For the `gamultiobj` function, set the `PlotFcn` option to ‘`gaplotpareto`’.

```
options = optimoptions("gamultiobj",PlotFcn="gaplotpareto");
sol = solve(prob,Options=options)
```

- For the `paretosearch` function, set the `PlotFcn` option to ‘`psplotparetof`’.

To view the Pareto set after the solver finishes, call `paretoplot` on the solution.

```
sol = solve(prob);
paretoplot(sol)
```

For an example, see “Pareto Front for Multiobjective Optimization, Problem-Based” on page 15-5.

If you have more than three objectives, `paretplot` allows you to choose which objectives to plot. See the `paretplot` reference page for details.

See Also

`gamultiobj` | `paretosearch` | `solve` | `optimvalues` | `paretplot`

Related Examples

- “Problem-Based Global Optimization Setup”
- “Pareto Front for Multiobjective Optimization, Problem-Based” on page 15-5
- “Multiobjective Optimization”

Pareto Front for Multiobjective Optimization, Problem-Based

This example shows how to solve a multiobjective optimization problem using optimization variables, and how to plot the solution.

Problem Formulation

The problem has a two-dimensional optimization variable and two objective functions. Create the optimization variable `x` as a row vector, the orientation expected by multiobjective solvers. Set bounds specifying that the components of `x` range from -50 through 50.

```
x = optimvar("x",1,2,LowerBound=-50,UpperBound=50);
```

Create the two-component objective function. Include the objective function in an optimization problem.

```
fun(1) = x(1)^4 + x(2)^4 + x(1)*x(2) - x(1)^2*x(2)^2 - 9*x(1)^2;
fun(2) = x(1)^4 + x(2)^4 + x(1)*x(2) - x(1)^2*x(2)^2 + 3*x(2)^3;
prob = optimproblem("Objective",fun);
```

Review the problem.

```
show(prob)
```

```
OptimizationProblem :

Solve for:
    x

minimize :
    (((x(1).^4 + x(2).^4) + (x(1) .* x(2)))) - (x(1).^2 .* x(2).^2) - (9 .* x(1).^2)
    (((x(1).^4 + x(2).^4) + (x(1) .* x(2)))) - (x(1).^2 .* x(2).^2) + (3 .* x(2).^3)

variable bounds:
    -50 <= x(1) <= 50
    -50 <= x(2) <= 50
```

Solve and Plot Solution

Call `solve` to solve the problem.

```
rng default % For reproducibility
sol = solve(prob)
```

Solving problem using gamultiobj.

Optimization terminated: average change in the spread of Pareto solutions less than options.Function

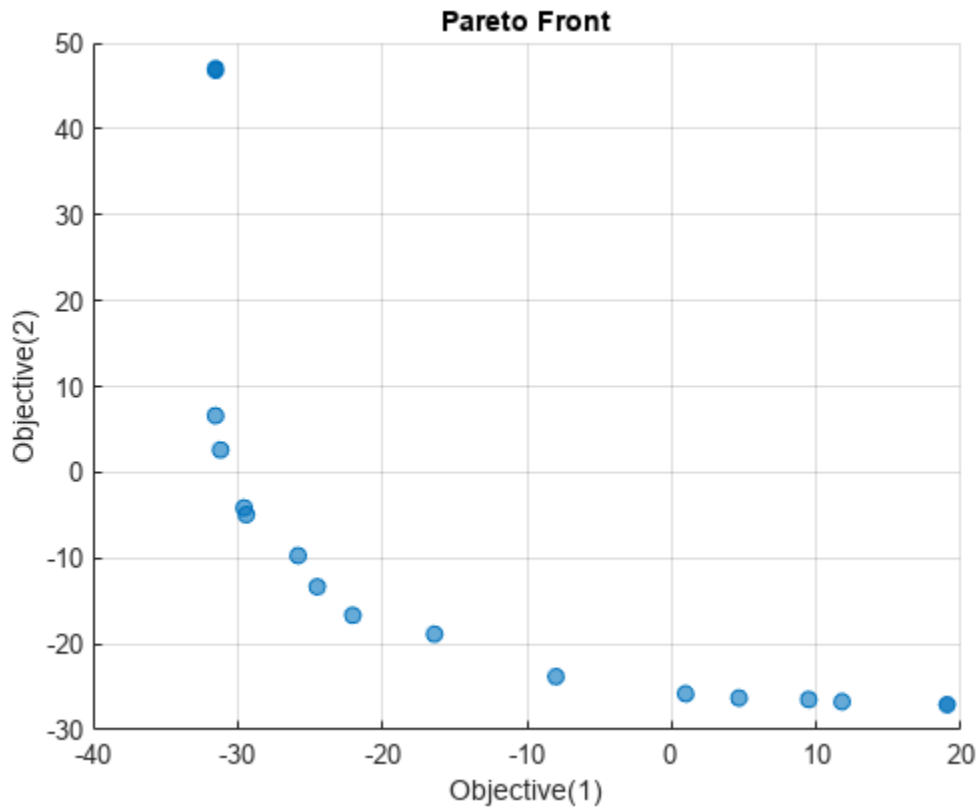
```
sol =
    1x18 OptimizationValues vector with properties:
```

```
Variables properties:
    x: [2x18 double]
```

```
Objective properties:
    Objective: [2x18 double]
```

Plot the resulting Pareto front.

```
paretplot(sol)
```



Solve the problem again using the `paretosearch` solver.

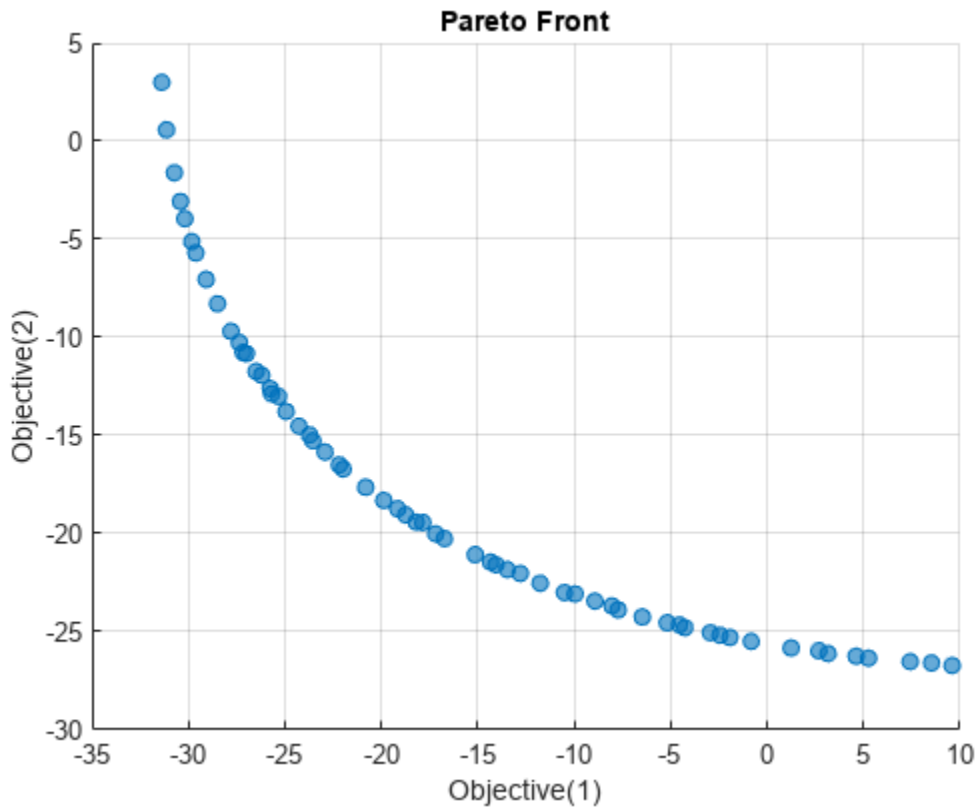
```
sol2 = solve(prob,Solver="paretosearch");
```

```
Solving problem using paretosearch.
```

```
Pareto set found that satisfies the constraints.
```

```
Optimization completed because the relative change in the volume of the Pareto set  
is less than 'options.ParetoSetChangeTolerance' and constraints are satisfied to within  
'options.ConstraintTolerance'.
```

```
paretplot(sol2)
```

Using default options, the `paretosearch` solver obtains a denser set of solution points than `gamultiobj`. However, `gamultiobj` obtains a wider range of values.

Start from Single-Objective Solutions

To attempt to achieve a better spread of solutions, find the single-objective solutions starting from $x = [1 \ 1]$.

```
x0.x = [1 1];
prob1 = optimproblem("Objective",fun(1));
solp1 = solve(prob1,x0);
```

Solving problem using `fmincon`.

Local minimum possible. Constraints satisfied.

`fmincon` stopped because the size of the current step is less than the value of the step size tolerance and constraints are satisfied to within the value of the constraint tolerance.

```
prob2 = optimproblem("Objective",fun(2));
solp2 = solve(prob2,x0);
```

Solving problem using `fmincon`.

Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in

feasible directions, to within the value of the optimality tolerance, and constraints are satisfied to within the value of the constraint tolerance.

Prepare the single-objective solutions as an initial point for `solve`. Each point must be passed as a column vector to the `optimvalues` function.

```
start = optimvalues(prob,"x",[solp1.x' solp2.x']);
```

Solve the multiobjective problem with `paretosearch` starting from the `start` points.

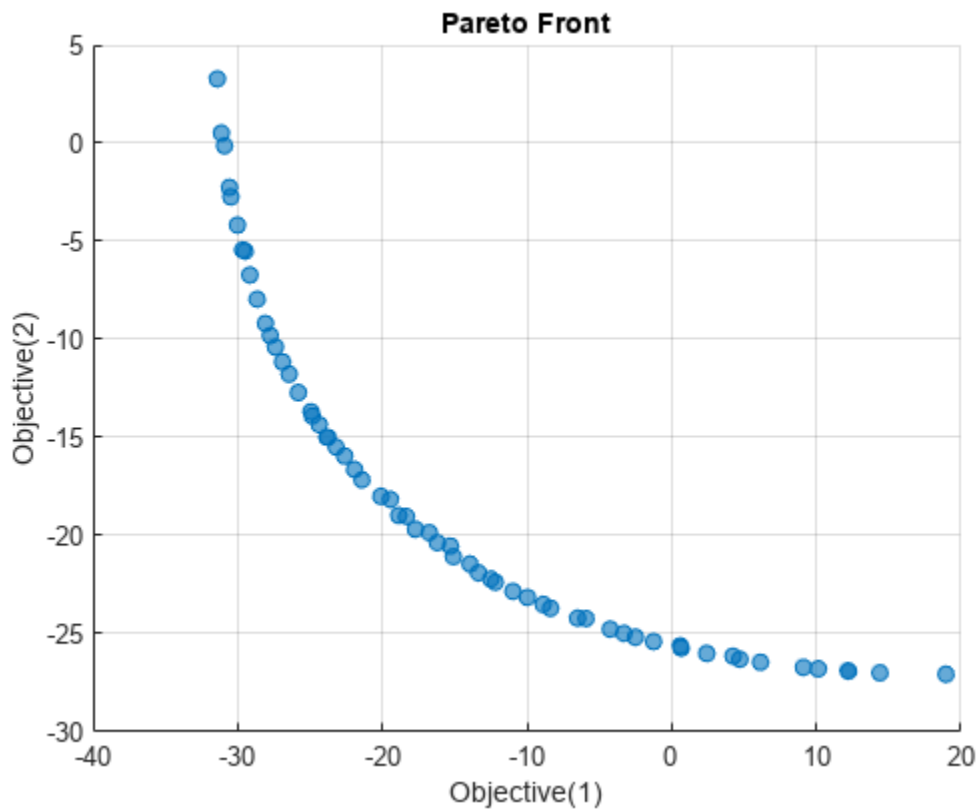
```
sol3 = solve(prob,start,Solver="paretosearch");
```

Solving problem using `paretosearch`.

Pareto set found that satisfies the constraints.

Optimization completed because the relative change in the volume of the Pareto set is less than `'options.ParetoSetChangeTolerance'` and constraints are satisfied to within `'options.ConstraintTolerance'`.

```
paretoplot(sol3)
```



This time, `paretosearch` finds a larger range of the objective functions, going almost to 10 in Objective 2 and almost to 20 in Objective 1. This range is similar to the `gamultiobj` range, except for the anomalous solution point near Objective 1 = -31, Objective 2 = 48.

See Also

`gamultiobj` | `paretosearch` | `solve` | `paretplot`

Related Examples

- “Multiobjective Optimization”

Plan Nuclear Fuel Disposal Using Multiobjective Optimization

This example shows how to formulate and solve a large nonlinear multiobjective problem that has some integer constraints. The problem is adapted from Montonen, Ranta, and Mäkelä [1] on page 15-23. The goal is to dispose of spent nuclear fuel, with objectives of minimizing cost, minimizing the amount of time between removal of a spent nuclear fuel assembly from a reactor until it is buried, and minimizing the number of spent fuel assemblies in storage at any one time. The problem is a multiperiod planning problem, and each period is five years long.

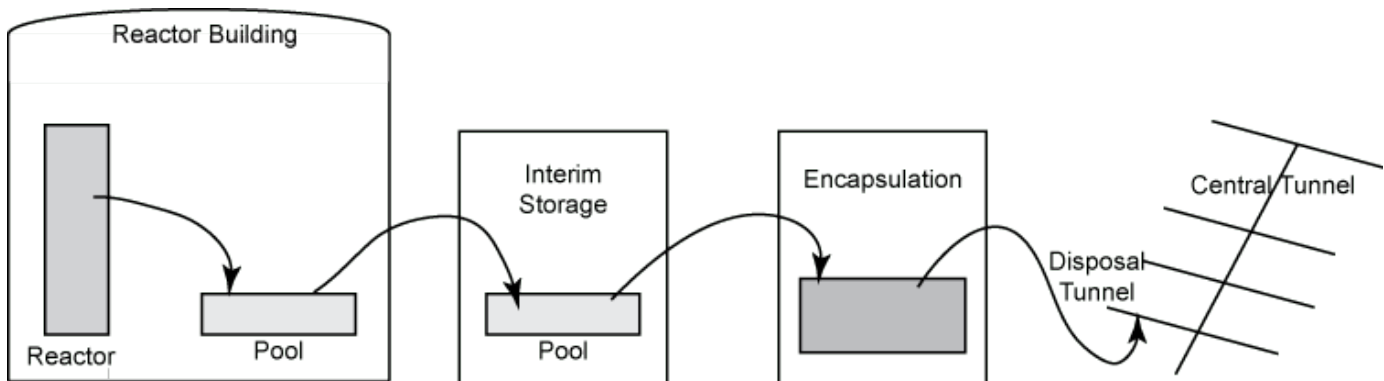
Model Overview

A nuclear reactor creates waste products that must be buried for long-term disposal. These waste products are fuel rod assemblies containing spent nuclear fuel. The assemblies are hot and radioactive when taken from the reactor, and gradually become less so. At period t , the radiation level is given by a double-exponential decay function with parameter vector u :

$$\text{radiation} = u_1 \exp(-u_2 t) + u_3 \exp(-u_4 t).$$

Each assembly remains in a pool of water in the reactor building until it is cool enough to be transferred to an interim storage facility, where it is again placed in a pool of water. When the assembly is cooled further, it can be encapsulated with other assemblies in a copper-iron canister and then buried in a disposal tunnel. All disposal tunnels connect to a central tunnel.

This figure illustrates the stages of the nuclear fuel assemblies from reactor to final disposal.



The problem variables relate to a schedule where each unit of time represents 5 years. Time periods begin at 1.

Constants Associated with Model

Z is the last time period in which fuel assemblies are removed from the reactor. Removal periods are $1:Z$. In [1], $Z = 11$. Each period is 5 years, so the last period in which fuel assemblies are removed is at time 55.

N is the last time period in which canisters are buried. Burial periods are $1:N$. In [1], $N = 19$. Each period is 5 years, so the last period in which canisters can be buried is at time 95.

$$\begin{aligned} Z &= 11; \\ N &= 19; \end{aligned}$$

a is the period of the last removal before the first disposal.

b is the disposal period in which the last removal occurs.

$a = 5;$

$b = 6;$

R is the minimum number of periods to store an assembly.

$R = 4;$

K is the maximum number of assemblies to fit into one canister.

$K = 4;$

T is the minimum number of canisters disposed in one period.

$T = 50;$

U is the maximum number of canisters disposed in one period.

$U = 500;$

Q is the length of a disposal tunnel in meters.

$Q = 350;$

$M(i)$ is the number of assemblies removed at time i .

$M = 300 - 60*(-1).^(1:Z);$ % 360 for odd indices, 240 for even

$A(i, j)$ is the storage time of an assembly from removal i in period j , where $i \leq Z$ and $j \leq N$.

$A = \text{zeros}(Z,N);$

for $i = 1:Z$

 for $j = i:N$

$A(i, j) = j - i;$

 end

end

$p(i, j)$ is the decay heat power of an assembly (in watts) from removal i in period j , where $i \leq Z$ and $j \leq N$.

% The following parameters fit $P_{i,j}$ of Table A2 from [1] to within 1 in each
% entry (fractional error $\leq 1/250$).

$u = [503 \ 0.1346 \ 260 \ 0.0231];$

$\text{myfun} = @(d)\text{round}(u(1)*\text{exp}(-u(2)*d) + u(3)*\text{exp}(-u(4)*d));$

$PP = \text{myfun}(1:N);$

$pij = \text{zeros}(Z,N);$

for $i = 2:Z$

 for $j = 1:i$

$pij(i,j) = 1e3;$ % Dummy values because $j \geq i$ does not occur.

 end

end

for $i=1:Z$

$pij(i,i:end) = PP(1:(N-i+1));$ % Same decay profile for all removal times

end

$p_{\max up}$ is the upper bound on the average power of a canister, and $p_{\max low}$ is the lower bound.

```
pmaxup = 1830;  
pmaxlow = 1300;
```

dDT_{up} is the upper bound on the distance between disposal tunnels, and dDT_{low} is the lower bound.

```
dDTup = 50;  
dDTlow = 25;
```

dCA_{up} is the upper bound on the distance between canisters in a disposal tunnel, and dCA_{low} is the lower bound.

```
dCAup = 15;  
dCALow = 6;
```

The costs associated with the operations are not given in [1]. This example assumes the following values:

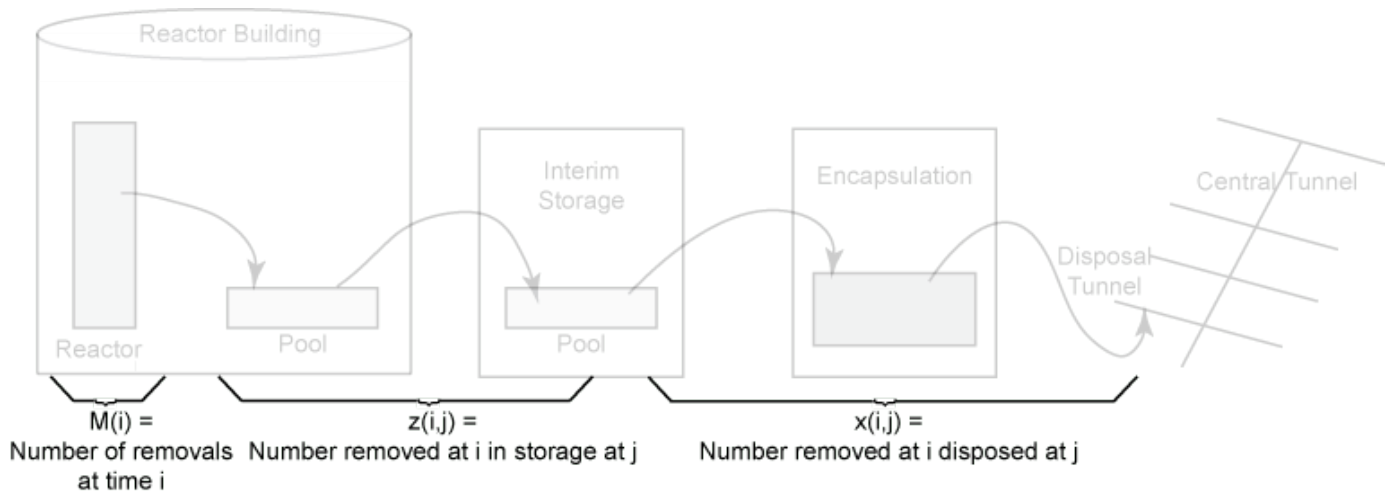
- C_{as} is the storage cost for one assembly per period.
- C_{is} is the storage cost for one assembly per period in interim storage.
- C_{sp} is the cost of a storage place per assembly.
- C_{ca} is the cost of a canister.
- C_{ef} is the cost of operating the encapsulation facility per period.
- C_{dt} is the cost of a disposal tunnel per meter.
- C_{ct} is the cost of the central tunnel per meter.

```
Cas = 50;  
Cis = 60;  
Csp = 10;  
Cca = 1200;  
Cef = 300;  
Cdt = 3000;  
Cct = 5000;
```

Optimization Variables for Problem

To create the problem for MATLAB®, use the problem-based approach. Define continuous variables for most quantities.

This figure illustrates the variables associated with the movement of the assemblies.



$x(i, j)$ is the number of assemblies removed at time i and disposed at time j , $i \leq Z$ and $j \leq N$.

$x = \text{optimvar}("x", Z, N, \text{LowerBound}=0, \text{UpperBound}=U*K);$

$z(i, j)$ is the number of assemblies removed at time i and in storage at time j , $i \leq Z$ and $j \leq N$.

$z = \text{optimvar}("z", Z, N, \text{LowerBound}=0, \text{UpperBound}=U*K*N/2);$

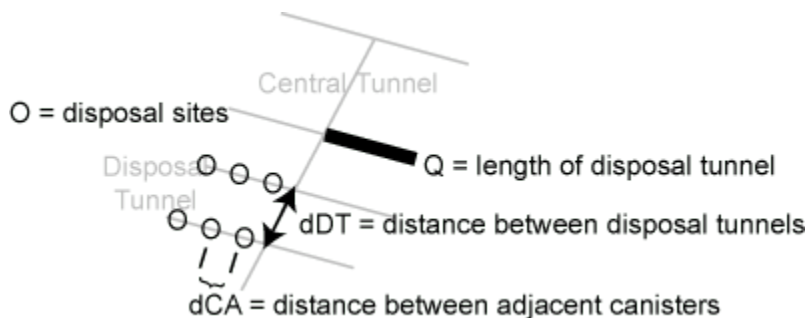
$y(j)$ is the number of canisters disposed at time $j \leq N$.

$y = \text{optimvar}("y", N, \text{LowerBound}=0, \text{UpperBound}=U);$

p_{\max} is the maximum average power of a canister.

$p_{\max} = \text{optimvar}("p_{\max}", \text{LowerBound}=p_{\max\text{low}}, \text{UpperBound}=p_{\max\text{up}});$

This figure illustrates quantities associated with the disposal tunnels.

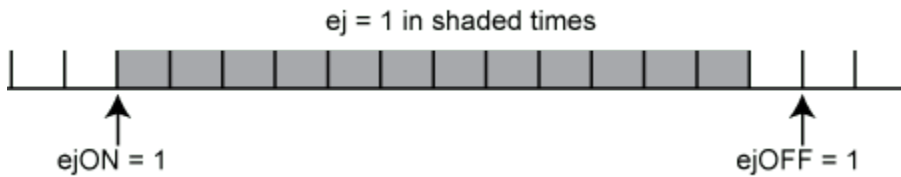


dDT is the distance between adjacent disposal tunnels.

$dDT = \text{optimvar}("dDT", \text{LowerBound}=dDT_{\text{low}}, \text{UpperBound}=dDT_{\text{up}});$

dCA is the distance between adjacent canisters in a disposal tunnel. You compute this distance later on, in the Problem Constraints section of this example, using the function g .

This figure relates to the encapsulation times.



Specify the following variables as integer type binary variables, which have lower bounds of 0 and upper bounds of 1.

$e_j(j)$ indicates that the encapsulation facility is in operation during the period $j \leq N$.

```
ej = optimvar("ej",N,Type="integer",LowerBound=0,UpperBound=1);
```

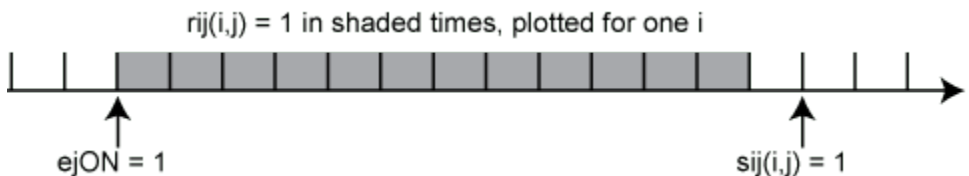
$e_{jON}(j)$ indicates that encapsulation starts at the beginning of period $j \leq N$.

```
ejON = optimvar("ejON",N,Type="integer",LowerBound=0,UpperBound=1);
```

$e_{jOFF}(j)$ indicates that encapsulation ends at the beginning of period $j \leq N$.

```
ejOFF = optimvar("ejOFF",N,Type="integer",LowerBound=0,UpperBound=1);
```

This figure relates to the times when assemblies can be disposed, $r_{ij} = 1$. These times start when $e_{jON} = 1$ and end when $s_{ij} = 1$.



$s_{ij}(i,j)$ indicates that assemblies removed at time i can no longer be disposed starting at the beginning of time j , $i \leq Z$ and $j \leq N$.

```
sij = optimvar("sij",Z,N,Type="integer",LowerBound=0,UpperBound=1);
```

$r_{ij}(i,j)$ indicates that assemblies removed at time i can be disposed at time j , $i \leq Z$ and $j \leq N$.

```
rij = optimvar("rij",Z,N,Type="integer",LowerBound=0,UpperBound=1);
```

All optimization variables and problem parameters are now defined.

Problem Constraints

Create an optimization problem to hold the objective and constraints.

```
prob = optimproblem;
```

The constraint numbers match the equations in [1]. The first three constraints relate to the number of assemblies in interim storage.

```
jnot1 = 2:N;
prob.Constraints.cons10 = z(:,1) - M(:) + x(:,1) == 0;
prob.Constraints.cons11 = z(:,jnot1) - z(:,(jnot1 - 1)) + x(:,jnot1) == 0;
prob.Constraints.cons12 = z(:,N) == 0;
```


Set the constraint that all assemblies are disposed once.

```
prob.Constraints.cons13 = sum(sij,2) == 1;
```

Define the variable rij by setting the following constraints.

```
cons15 = optimconstr(Z,N);
cons15(:,1) = rij(:,1) == -sij(:,1) + ejON(1); % equation 14
cons15(:,jnot1) = rij(:,jnot1) == ...
    rij(:,jnot1-1) - sij(:,jnot1) + repmat(ejON(jnot1)',Z,1); % equation 15
prob.Constraints.cons15 = cons15;
```

Set the constraint that disposal occurs only during times when the encapsulation facility is in operation.

```
cons16 = rij <= repmat(ej', Z, 1);
prob.Constraints.cons16 = cons16;
```

Specify the constraint that production capacity is not exceeded.

```
prob.Constraints.cons17 = x <= U*K*rij;
```

The next constraint enforces that assemblies are cool enough before disposal.

```
prob.Constraints.cons18 = x.*(A - R) >= 0;
```

The following constraints relate to the encapsulation facility. These constraints enforce that the facility is turned on and off only once, which means that all canisters are encapsulated in one run,

```
prob.Constraints.cons19 = sum(ejON) == 1;
prob.Constraints.cons20 = sum(ejOFF) == 1;
```

Define variable ej by setting the following constraints.

```
cons21 = optimconstr(N);
cons21(1) = ej(1) == ejON(1) - ejOFF(1); % equation 21
cons21(jnot1) = ej(jnot1) == ...
    ej(jnot1 - 1) + ejON(jnot1) - ejOFF(jnot1); % equation 22
prob.Constraints.cons21 = cons21;
```

Set constraints that the number of canisters is sufficient for disposal, does not exceed production capacity, and obeys the minimum production constraint.

```
prob.Constraints.cons23 = y' >= (1/K)*sum(x,1);
prob.Constraints.cons24 = y <= U*ej;
jnotN = 1:(N-1);
prob.Constraints.cons25 = y(jnotN) >= T*(ej(jnotN) - ejOFF(jnotN + 1));
```

Regarding the disposal facility, set the constraint that the heat power of canisters is allowable.

```
prob.Constraints.cons26 = sum(pij.*x,1) <= pmax*y';
```

Specify a nonlinear constraint on the distance between buried canisters. The function is piecewise linear, and is defined using the `max` function, which is not a supported operation for optimization expressions. Therefore, use `fcn2optimexpr` to place the constraint into `prob`.

```
g = @(pmax,dDT)max([-2.26911*dDT + 0.00675*pmax + 54.5288,...
    -0.05833*dDT + 0.00596*pmax - 0.727083,...
    -0.14*dDT + 0.17701*pmax - 350.651]);
```

```
dCA = fcn2optimexpr(@(pmax,dDT)g(pmax,dDT),pmax,dDT);
prob.Constraints.cons29a = dCA >= dCAlow;
prob.Constraints.cons29b = dCA <= dCAup;
```

Cost Objective

The first objective for this multiobjective problem is the cost, which has seven components.

```
cost = optimexpr(7,1);
```

1. Storage cost of assemblies. This cost is the sum of the cost per unit time multiplied by the length of time each assembly is stored.

```
cost(1) = Cas*sum(A.*x,"all");
```

2. Cost of interim storage. This cost is $j \cdot e_{jOFF}(j)$ for the one component of e_{jOFF} that is 1.

```
cost(2) = Cis*max(ejOFF(1)-1,2*ejOFF(2)-1,3*ejOFF(3)-1,...,N*ejOFF(N)-1).
```

To represent this expression briefly, represent $cost(2) = Cis \cdot u$ for a new optimization variable u , along with the constraint

```
ucons = u >= ((1:N)'.*ejOFF) - 1.
```

```
u = optimvar("u",LowerBound=0);
cost(2) = Cis*u;
ucons = u >= ((1:N)'.*ejOFF) - 1;
prob.Constraints.ucons = ucons;
```

3. Cost of positions for assembly storage. $cost(3)$ can be represented by $Csp \cdot v1$, where $v1$ is a new optimization variable, along with the constraints

```
v1 >= sum(M)
```

```
v1 >= sum_{i=1}^j z(i,j) for each 1 <= j <= N
```

```
v1 >= sum_{i=1}^Z z(i,j) for each b <= j <= N.
```

Create these costs and associated constraints.

```
v1 = optimvar("v1",LowerBound=0);
cost(3) = Csp*v1; % Include the three v1 constraints given below.
vlconsa = v1 >= sum(M);
bmin1 = 1:(b-1);
vlconsb = optimconstr(b-1);
for j=bmin1
    vlconsb(j) = sum(z(1:a+j,j)) <= v1;
end
ell = b:N;
vlconsc = sum(z(:,ell),1) <= v1;
prob.Constraints.vlconsa = vlconsa;
prob.Constraints.vlconsb = vlconsb;
prob.Constraints.vlconsc = vlconsc;
```

4. Cost of canisters. This cost is the cost per canister times the total number of canisters buried.

```
cost(4) = Cca*sum(y);
```

5. Cost of running the encapsulation facility. This cost is the cost per unit time multiplied by the length of time the facility operates.

```
cost(5) = Cef*sum(ej);
```

6. Cost of disposal tunnels. This is the cost per unit length times the length between canisters times the total number of canisters buried.

```
cost(6) = Cdt*dCA*sum(y);
```

7. Cost of central tunnel. This cost is the cost per unit length times the required length of the central tunnel. The number of canisters that can be buried in one disposal tunnel is its length Q divided by the distance between canisters dCA . The length of the central tunnel is proportional to the number of buried canisters $\text{sum}(y)$ and inversely proportional to Q/dCA , and has cost proportional to Cct .

```
cost(7) = Cct/Q*dDT*dCA*sum(y);
```

The total cost is the sum of the seven cost components. To change the scale of the total cost to match that of the other objectives, take the logarithm of the sum.

```
prob.Objective.cost = log(sum(cost));
```

Safety Objectives

The problem has two objectives related to safety. Objective 2, named `safety1`, tries to minimize the maximum storage time over all removals. Objective 3, named `safety2`, tries to stop the disposal as early as possible. Define these two objectives using the helper functions `max1` and `max2`, which appear at the end of this script on page 15-23.

```
prob.Objective.safety1 = fcn2optimexpr(@max1,A,sij);
% Minimize maximum storage time, objective (2) in [1]
```

```
prob.Objective.safety2 = fcn2optimexpr(@max2,ejOFF);
% Stop disposal as early as possible, objective (5) in [1]
```

Set Options

Set the options for a Pareto plot as the solver proceeds. Because the problem has over 900 variables, set options to use a population size of 500, which is larger than the default. Also, because the problem contains binary variables, use the `mutationgaussian` mutation function. This mutation function works better than the default `mutationpower` for binary variables.

```
opts = optimoptions("gamultiobj",PlotFcn="gaplotpareto",PopulationSize=5e2,...
    MutationFcn=@mutationgaussian);
```

Run Problem

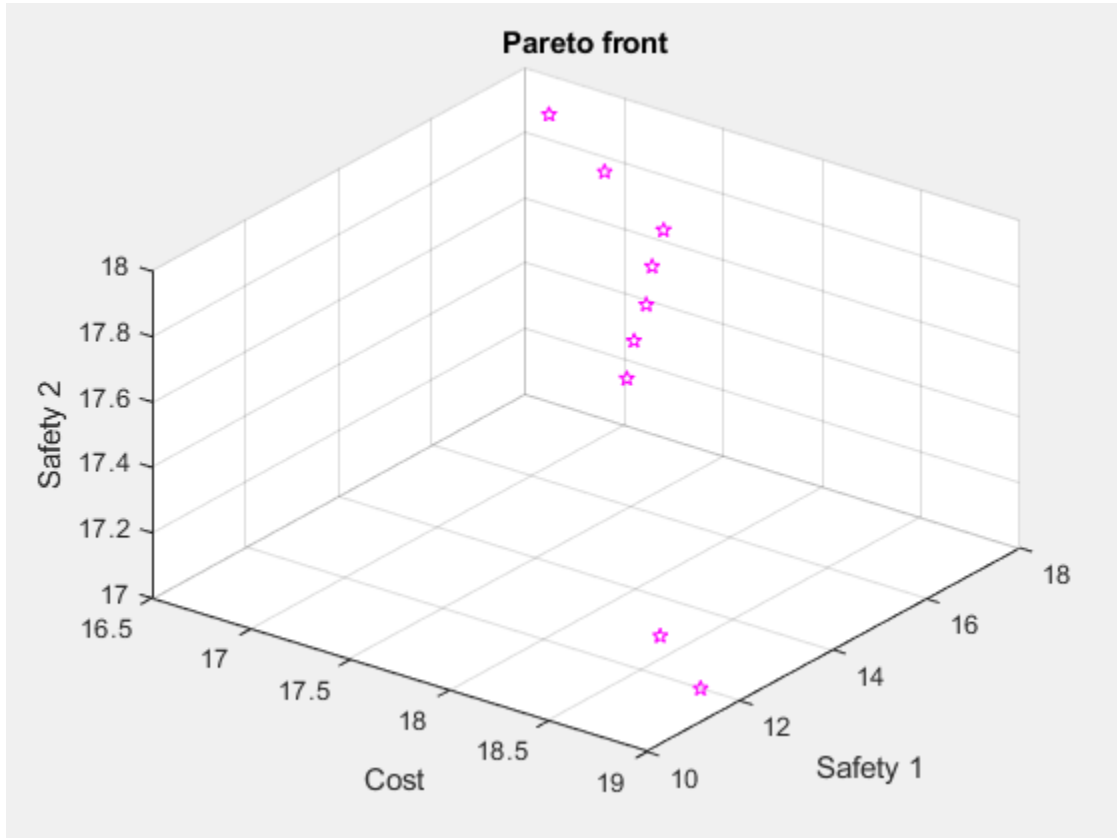
The problem formulation is complete, and the options are set for this multiobjective problem. Run the problem.

```
rng default % For reproducibility
[sol,fval,exitflag,output] = solve(prob,Options=opts);
```

```
Solving problem using gamultiobj.
```

```
Optimization terminated: average change in the spread of Pareto solutions less than options.Func
```

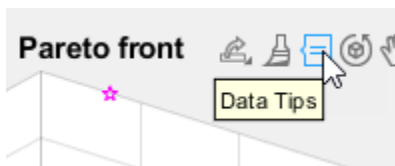
```
xlabel("Cost")
ylabel("Safety 1")
zlabel("Safety 2")
```



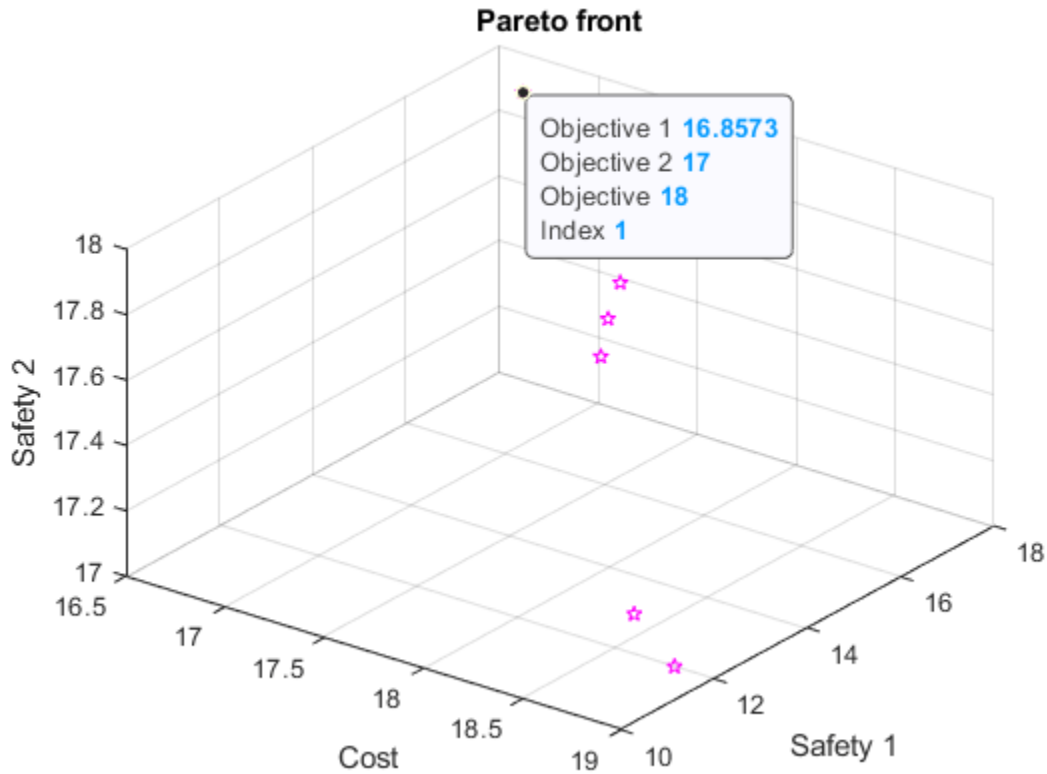
The Pareto plot shows a clear tradeoff between Cost and Safety 1. The true cost is the exponential of the amount shown, so the tradeoff is more severe than illustrated.

Examine Solution

`gamultiobj` finds several feasible solutions with varying fitness function values. To find the control variables associated with the solutions, use the data tips.



After activating Data Tips, click the upper-left solution.



The index of the selected point is 1. The variables associated with this point are in `sol(1)`.

Examine the `x` variables associated with this solution. Recall that $x(i, j)$ are the removals at time i that are disposed at time j .

```
disp(sol(1).x)
```

```
Columns 1 through 12
```

```

0      0      0      0      -0.0000      0      0      0      0
0      0      0      0      0      0.0000      0      0      0
0      0      0      0      0      0      0      0      0
0      0      0      0      0      0      0      -0.0000      0
0      0      0      0      0      0      0      0      -0.0000
0      0      0      0      0      0      0      0      0
0      0      0      0      0      0      0      0      0
0      0      0      0      0      0      0      0      0
0      0      0      0      0      0      0      0      0
0      0      0      0      0      0      0      0      0
0      0      0      0      0      0      0      0      0

```

```
Columns 13 through 19
```

```

0      0      0      0      228.4792      131.5208      0
0      0      0      0      0      240.0000      0
0      0      0      0      0      360.0000      0
0      0      0      0      0      240.0000      0
0      0      0      0      0      360.0000      0

```

```

      0      0      0      0      0  240.0000      0
      0      0      0      0  354.0977   5.9023      0
      0      0      0      0  228.2471  11.7529      0
-0.0000      0      0      0      0  360.0000      0
      0  0.0000      0      0  240.0000      0      0
      0      0      0      0  360.0000      0      0
    
```

Clearly, the x schedule is not restricted to integer values. View the sums of the x schedule over disposals compared to the removal quantities $M(:)$.

```
disp([sum(sol(1).x,2),M(:)])
```

```

360.0000  360.0000
240.0000  240.0000
360.0000  360.0000
240.0000  240.0000
360.0000  360.0000
240.0000  240.0000
360.0000  360.0000
240.0000  240.0000
360.0000  360.0000
240.0000  240.0000
360.0000  360.0000
    
```

The x schedule accounts for all removals.

What are the times when the encapsulation facility is in operation?

```
disp(sol(1).ej')
```

```

Columns 1 through 12
      0      0      0      0      0      0      0      0      0
Columns 13 through 19
      0      0      0      0  1.0000  1.0000      0
    
```

The encapsulation runs for times 17 and 18.

What is the distance between disposal tunnels?

```
disp(sol(1).dDT)
```

```
33.1986
```

The distance is about halfway between its lower bound of 25 and its upper bound of 50.

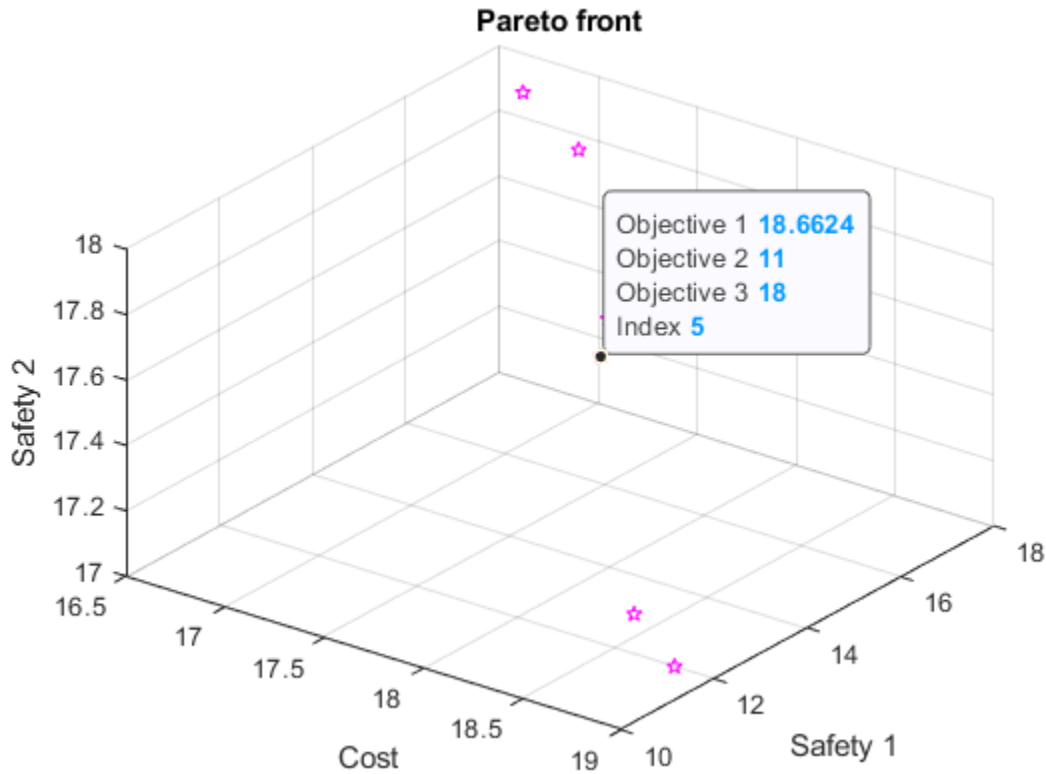
What is the dollar cost of the schedule? To find out, calculate $\exp(\text{sol}(1).\text{cost})$, because $\text{sol}(1).\text{cost}$ is the logarithm of the total disposal cost.

```
disp(exp(sol(1).cost))
```

```
2.0943e+07
```

The cost is about \$21 million.

Examine the point in the Pareto set with the lowest value of Objective 2.



The monetary cost of this operating point is much higher.

```
disp(exp(sol(5).cost))
```

```
1.2735e+08
```

The monetary cost is about \$127 million, which is over five times the previous value. But the gain is that Objective 2 is 11 instead of 16, which corresponds to waste burial that is 25 years earlier. Earlier burial can be considered safer.

View the schedule of x for this solution.

```
disp(sol(5).x)
```

```
Columns 1 through 12
```

0	0	0	0	360.0000	0	0	0	0	0	0	0
0	0	0	0	0	0.0000	240.0000	0	0	0	0	0
0	0	0	0	0	0	0.0000	-0.0000	0	0	0	0
0	0	0	0	0	0	0	-0.0000	0	0	0	0
0	0	0	0	0	0	0	0	0	227.2852	0	0
0	0	0	0	0	0	0	0	0	0	0	240.0000
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

Columns 13 through 19

```

0      0      0      0      0      0      0
0      0      0      0      0      0      0
0      0      0      0      0      0      0
67.2852  0 172.7148  0      0      0      0
132.7148  0      0      0      0      0      0
0      0      0      0      0      0      0
0      0      0 360.0000  0      0      0
0 210.2641  0 29.7359  0      0      0
0      0      0      0      0 360.0000  0
0      0      0 240.0000  0      0      0
0      0 360.0000  0      0      0      0

```

View the sums of the x schedule over disposals compared to the removal quantities $M(:)$.

```
disp([sum(sol(5).x,2),M(:)])
```

```

360.0000 360.0000
240.0000 240.0000
360.0000 360.0000
240.0000 240.0000
360.0000 360.0000
240.0000 240.0000
360.0000 360.0000
240.0000 240.0000
360.0000 360.0000
240.0000 240.0000
360.0000 360.0000

```

Again, the schedule accounts for all removals.

What are the times when the encapsulation facility is in operation?

```
disp(sol(5).ej')
```

Columns 1 through 12

```

0      0  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000

```

Columns 13 through 19

```

1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  0

```

The encapsulation runs for times 3 through 18.

What is the distance between disposal tunnels for this solution?

```
disp(sol(5).dDT)
```

```
50
```

This time, the distance between disposal tunnels is as great as possible, 50 meters.

Conclusion

This example shows the formulation of a nonlinear, multiobjective, mixed-integer optimization problem using the problem-based approach. The Data Tips in the Pareto plot enable you to analyze

the solution. Instead of specifying the `gaplotpareto` plot function, you can use the `paretoplot` function to obtain a similar plot from the solution.

References

[1] Montonen, Outi, Timo Ranta, and Marko M. Mäkelä. *Planning the Schedule for the Disposal of the Spent Nuclear Fuel with Interactive Multiobjective Optimization*. Algorithms Vol. 12, Issue 12, 2019. Available at <https://www.mdpi.com/1999-4893/12/12/252>.

Helper Functions

This code creates the `max1` helper function.

```
function v = max1(A,sij)
v = round(max(max(A.*sij - 1))); % "round" ensures integer values
end
```

This code creates the `max2` helper function.

```
function v = max2(ej0FF)
v = round(max((ej0FF').*(1:length(ej0FF)) - 1));
end
```

See Also

`gamultiobj` | `solve` | `paretoplot` | `paretosearch`

Related Examples

- “Multiobjective Optimization”

Parallel Processing

- “How Solvers Compute in Parallel” on page 16-2
- “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11
- “Minimizing an Expensive Optimization Problem Using Parallel Computing Toolbox” on page 16-16

How Solvers Compute in Parallel

In this section...

“Parallel Processing Types in Global Optimization Toolbox” on page 16-2

“How Toolbox Functions Distribute Processes” on page 16-3

Parallel Processing Types in Global Optimization Toolbox

Parallel processing is an attractive way to speed optimization algorithms. To use parallel processing, you must have a Parallel Computing Toolbox license, and have a parallel worker pool (`parpool`). For more information, see “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11.

Global Optimization Toolbox solvers use parallel computing in various ways.

Solver	Parallel?	Parallel Characteristics
<code>GlobalSearch</code>	×	No parallel functionality. However, <code>fmincon</code> can use parallel gradient estimation when run in <code>GlobalSearch</code> . See “Using Parallel Computing in Optimization Toolbox”.
<code>MultiStart</code>	✓	Start points distributed to multiple processors. From these points, local solvers run to completion. For more details, see “MultiStart” on page 16-4 and “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11. For <code>fmincon</code> , no parallel gradient estimation with parallel <code>MultiStart</code> .
<code>ga</code> , <code>gamultiobj</code>	✓	Population evaluated in parallel, which occurs once per iteration. For more details, see “Genetic Algorithm” on page 16-7 and “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11. No vectorization of fitness or constraint functions.
<code>particleswarm</code>	✓	Population evaluated in parallel, which occurs once per iteration. For more details, see “Particle Swarm” on page 16-8 and “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11. No vectorization of objective or constraint functions.
<code>patternsearch</code> , <code>paretosearch</code>	✓	Poll points evaluated in parallel, which occurs once per iteration. For more details, see “Pattern Search” on page 16-6 and “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11. No vectorization of objective or constraint functions.
<code>simulannealbnd</code>	×	No parallel functionality. However, <code>simulannealbnd</code> can use a hybrid function that runs in parallel. See “Simulated Annealing” on page 16-9.
<code>surrogateopt</code>	✓	Search points evaluated in parallel. No vectorization of objective or constraint functions.

In addition, several solvers have hybrid functions that run after they finish. Some hybrid functions can run in parallel. Also, most `patternsearch` search methods can run in parallel. For more information, see “Parallel Search Functions or Hybrid Functions” on page 16-14.

How Toolbox Functions Distribute Processes

- “parfor Characteristics and Caveats” on page 16-3
- “MultiStart” on page 16-4
- “GlobalSearch” on page 16-5
- “Pattern Search” on page 16-6
- “Genetic Algorithm” on page 16-7
- “Parallel Computing with gamultiobj” on page 16-8
- “Particle Swarm” on page 16-8
- “Simulated Annealing” on page 16-9
- “Pareto Search” on page 16-9
- “Surrogate Optimization” on page 16-9

parfor Characteristics and Caveats

No Nested parfor Loops

Most solvers employ the Parallel Computing Toolbox `parfor` function to perform parallel computations. Two solvers, `surrogateopt` and `paretosearch`, use `parfeval` instead.

Note `parfor` does not work in parallel when called from within another `parfor` loop.

Note The documentation recommends not to use `parfor` or `parfeval` when calling Simulink®; see “Using sim Function Within parfor” (Simulink). Therefore, you might encounter issues when optimizing a Simulink simulation in parallel using a solver's built-in parallel functionality.

Suppose, for example, your objective function `userfcn` calls `parfor`, and you want to call `fmincon` using `MultiStart` and parallel processing. Suppose also that the conditions for parallel gradient evaluation of `fmincon` are satisfied, as given in “Parallel Optimization Functionality”. The figure “When parfor Runs In Parallel” on page 16-4 shows three cases:

- 1 The outermost loop is parallel `MultiStart`. Only that loop runs in parallel.
- 2 The outermost `parfor` loop is in `fmincon`. Only `fmincon` runs in parallel.
- 3 The outermost `parfor` loop is in `userfcn`. In this case, `userfcn` can use `parfor` in parallel.

Bold indicates the function that runs in parallel

```

...
① problem = createOptimProblem(fmincon,'objective',@userfcn,...)
   ms = MultiStart('UseParallel',true);
   x = run(ms,problem,10)
...
      Only the outermost parfor loop
      runs in parallel

      If fmincon UseParallel = true
      fmincon estimates gradients in parallel

...
② x(i) = fmincon(@userfcn,...)
...

      If fmincon UseParallel = false
      userfcn can use parfor in parallel

...
③ x = fmincon(@userfcn,...)
...

```

When parfor Runs In Parallel

Parallel Random Numbers Are Not Reproducible

Random number sequences in MATLAB are pseudorandom, determined from a *seed*, or an initial setting. Parallel computations use seeds that are not necessarily controllable or reproducible. For example, each instance of MATLAB has a default global setting that determines the current seed for random sequences.

For `patternsearch`, if you select MADS as a poll or search method, parallel pattern search does not have reproducible runs. If you select the genetic algorithm or Latin hypercube as search methods, parallel pattern search does not have reproducible runs.

For `ga` and `gamultiobj`, parallel population generation gives nonreproducible results.

`MultiStart` is different. You *can* have reproducible runs from parallel `MultiStart`. Runs are reproducible because `MultiStart` generates pseudorandom start points locally, and then distributes the start points to parallel processors. Therefore, the parallel processors do not use random numbers. For more details, see “Parallel Processing and Random Number Streams” on page 4-55.

Limitations and Performance Considerations

More caveats related to `parfor` appear in “Parallel for-Loops (`parfor`)” (Parallel Computing Toolbox).

For information on factors that affect the speed of parallel computations, and factors that affect the results of parallel computations, see “Improving Performance with Parallel Computing”. The same considerations apply to parallel computing with Global Optimization Toolbox functions.

MultiStart

`MultiStart` can automatically distribute a problem and start points to multiple processes or processors. The problems run independently, and `MultiStart` combines the distinct local minima into a vector of `GlobalOptimSolution` objects. `MultiStart` uses parallel computing when you:

- Have a license for Parallel Computing Toolbox software.

- Enable parallel computing with `parpool`, a Parallel Computing Toolbox function.
- Set the `UseParallel` property to `true` in the `MultiStart` object:

```
ms = MultiStart('UseParallel',true);
```

When these conditions hold, `MultiStart` distributes a problem and start points to processes or processors one at a time. The algorithm halts when it reaches a stopping condition or runs out of start points to distribute. If the `MultiStart Display` property is `'iter'`, then `MultiStart` displays:

```
Running the local solvers in parallel.
```

For an example of parallel `MultiStart`, see “Parallel `MultiStart`” on page 4-83.

Implementation Issues in Parallel `MultiStart`

`fmincon` cannot estimate gradients in parallel when used with parallel `MultiStart`. This lack of parallel gradient estimation is due to the limitation of `parfor` described in “No Nested `parfor` Loops” on page 16-3.

`fmincon` can take longer to estimate gradients in parallel rather than in serial. In this case, using `MultiStart` with parallel gradient estimation in `fmincon` amplifies the slowdown. For example, suppose the `ms` `MultiStart` object has `UseParallel` set to `false`. Suppose `fmincon` takes 1 s longer to solve problem with `problem.options.UseParallel` set to `true`. Then `run(ms,problem,200)` takes 200 s longer than the same run with `problem.options.UseParallel` set to `false`.

Note When executing serially, `parfor` loops run slower than `for` loops. Therefore, for best performance, set your local solver `UseParallel` option to `false` when the `MultiStart UseParallel` property is `true`.

Note Even when running in parallel, a solver occasionally calls the objective and nonlinear constraint functions serially on the host machine. Therefore, ensure that your functions have no assumptions about whether they are evaluated in serial and parallel.

GlobalSearch

`GlobalSearch` does not distribute a problem and start points to multiple processes or processors. However, when `GlobalSearch` runs the `fmincon` local solver, `fmincon` can estimate gradients by parallel finite differences. `fmincon` uses parallel computing when you:

- Have a license for Parallel Computing Toolbox software.
- Enable parallel computing with `parpool`, a Parallel Computing Toolbox function.
- Set the `UseParallel` option to `true` with `optimoptions`. Set this option in the problem structure:

```
opts = optimoptions(@fmincon,'UseParallel',true,'Algorithm','sqp');
problem = createOptimProblem('fmincon','objective',@myobj,...
    'x0',startpt,'options',opts);
```

For more details, see “Using Parallel Computing in Optimization Toolbox”.

Pattern Search

`patternsearch` can automatically distribute the evaluation of objective and constraint functions associated with the points in a pattern to multiple processes or processors. `patternsearch` uses parallel computing when you:

- Have a license for Parallel Computing Toolbox software.
- Enable parallel computing with `parpool`, a Parallel Computing Toolbox function.
- Set the following options using `optimoptions`:
 - `UseCompletePoll` is `true`.
 - `UseVectorized` is `false` (default).
 - `UseParallel` is `true`.

When these conditions hold, the solver computes the objective function and constraint values of the pattern search in parallel during a poll. Furthermore, `patternsearch` overrides the setting of the `Cache` option, and uses the default 'off' setting.

Beginning in R2019a, when you set the `UseParallel` option to `true`, `patternsearch` internally overrides the `UseCompletePoll` setting to `true` so that the function polls in parallel.

Note Even when running in parallel, `patternsearch` occasionally calls the objective and nonlinear constraint functions serially on the host machine. Therefore, ensure that your functions have no assumptions about whether they are evaluated in serial or parallel.

Parallel Search Function

`patternsearch` can optionally call a search function at each iteration. The search is parallel when you:

- Set `UseCompleteSearch` to `true`.
- Do not set the search method to `@searchneldermead` or `custom`.
- Set the search method to a `patternsearch` poll method or Latin hypercube search, and set `UseParallel` to `true`.
- Or, if you set the search method to `ga`, create a search method option with `UseParallel` set to `true`.

Implementation Issues in Parallel Pattern Search

The limitations on `patternsearch` options, listed in “Pattern Search” on page 16-6, arise partly from the limitations of `parfor`, and partly from the nature of parallel processing:

- `Cache` is overridden to be 'off' — `patternsearch` implements `Cache` as a persistent variable. `parfor` does not handle persistent variables, because the variable could have different settings at different processors.
- `UseCompletePoll` is `true` — `UseCompletePoll` determines whether a poll stops as soon as `patternsearch` finds a better point. When searching in parallel, `parfor` schedules all evaluations simultaneously, and `patternsearch` continues after all evaluations complete. `patternsearch` cannot halt evaluations after they start.

Beginning in R2019a, when you set the `UseParallel` option to `true`, `patternsearch` internally overrides the `UseCompletePoll` setting to `true` so that the function polls in parallel.

- `UseVectorized` is `false` — `UseVectorized` determines whether `patternsearch` evaluates all points in a pattern with one function call in a vectorized fashion. If `UseVectorized` is `true`, `patternsearch` does not distribute the evaluation of the function, so does not use `parfor`.

Genetic Algorithm

`ga` and `gamultiobj` can automatically distribute the evaluation of objective and nonlinear constraint functions associated with a population to multiple processors. `ga` uses parallel computing when you:

- Have a license for Parallel Computing Toolbox software.
- Enable parallel computing with `parpool`, a Parallel Computing Toolbox function.
- Set the following options using `optimoptions`:
 - `UseVectorized` is `false` (default).
 - `UseParallel` is `true`.

When these conditions hold, `ga` computes the objective function and nonlinear constraint values of the individuals in a population in parallel.

Note Even when running in parallel, `ga` occasionally calls the fitness and nonlinear constraint functions serially on the host machine. Therefore, ensure that your functions have no assumptions about whether they are evaluated in serial or parallel.

Implementation Issues in Parallel Genetic Algorithm

The limitations on options, listed in “Genetic Algorithm” on page 16-7, arise partly from limitations of `parfor`, and partly from the nature of parallel processing:

- `UseVectorized` is `false` — `UseVectorized` determines whether `ga` evaluates an entire population with one function call in a vectorized fashion. If `UseVectorized` is `true`, `ga` does not distribute the evaluation of the function, so does not use `parfor`.

`ga` can have a hybrid function that runs after it finishes; see “Hybrid Scheme in the Genetic Algorithm” on page 8-95. If you want the hybrid function to take advantage of parallel computation, set its options separately so that `UseParallel` is `true`. If the hybrid function is `patternsearch`, set `UseCompletePoll` to `true` so that `patternsearch` runs in parallel.

If the hybrid function is `fmincon`, set the following options with `optimoptions` to have parallel gradient estimation:

- `GradObj` must not be `'on'` — it can be `'off'` or `[]`.
- Or, if there is a nonlinear constraint function, `GradConstr` must not be `'on'` — it can be `'off'` or `[]`.

To find out how to write options for the hybrid function, see “Parallel Hybrid Functions” on page 16-14.

Parallel Computing with `gamultiobj`

Parallel computing with `gamultiobj` works almost the same as with `ga`. For detailed information, see “Genetic Algorithm” on page 16-7.

The difference between parallel computing with `gamultiobj` and `ga` has to do with the hybrid function. `gamultiobj` allows only one hybrid function, `fgoalattain`. This function optionally runs after `gamultiobj` finishes its run. Each individual in the calculated Pareto frontier, that is, the final population found by `gamultiobj`, becomes the starting point for an optimization using `fgoalattain`. These optimizations run in parallel. The number of processors performing these optimizations is the smaller of the number of individuals and the size of your `parpool`.

For `fgoalattain` to run in parallel, set its options correctly:

```
fgoalopts = optimoptions(@fgoalattain,'UseParallel',true)
gaoptions = optimoptions('ga','HybridFcn',{@fgoalattain,fgoalopts});
```

Run `gamultiobj` with `gaoptions`, and `fgoalattain` runs in parallel. For more information about setting the hybrid function, see “Hybrid Function Options” on page 17-39.

`gamultiobj` calls `fgoalattain` using a `parfor` loop, so `fgoalattain` does not estimate gradients in parallel when used as a hybrid function with `gamultiobj`. For more information, see “No Nested `parfor` Loops” on page 16-3.

Particle Swarm

`particleswarm` can automatically distribute the evaluation of the objective function associated with a population to multiple processors. `particleswarm` uses parallel computing when you:

- Have a license for Parallel Computing Toolbox software.
- Enable parallel computing with `parpool`, a Parallel Computing Toolbox function.
- Set the following options using `optimoptions`:
 - `UseVectorized` is `false` (default).
 - `UseParallel` is `true`.

When these conditions hold, `particleswarm` computes the objective function of the particles in a population in parallel.

Note Even when running in parallel, `particleswarm` occasionally calls the objective function serially on the host machine. Therefore, ensure that your objective function has no assumptions about whether it is evaluated in serial or parallel.

Implementation Issues in Parallel Particle Swarm Optimization

The limitations on options, listed in “Particle Swarm” on page 16-8, arise partly from limitations of `parfor`, and partly from the nature of parallel processing:

- `UseVectorized` is `false` — `UseVectorized` determines whether `particleswarm` evaluates an entire population with one function call in a vectorized fashion. If `UseVectorized` is `true`, `particleswarm` does not distribute the evaluation of the function, so does not use `parfor`.

`particleswarm` can have a hybrid function that runs after it finishes; see “Hybrid Scheme in the Genetic Algorithm” on page 8-95. If you want the hybrid function to take advantage of parallel

computation, set its options separately so that `UseParallel` is `true`. If the hybrid function is `patternsearch`, set `UseCompletePoll` to `true` so that `patternsearch` runs in parallel.

If the hybrid function is `fmincon`, set the `GradObj` option to `'off'` or `[]` with `optimoptions` to have parallel gradient estimation.

To find out how to write options for the hybrid function, see “Parallel Hybrid Functions” on page 16-14.

Simulated Annealing

`simulannealbnd` does not run in parallel automatically. However, it can call hybrid functions that take advantage of parallel computing. To find out how to write options for the hybrid function, see “Parallel Hybrid Functions” on page 16-14.

Pareto Search

`paretosearch` can automatically distribute the evaluation of the objective function associated with a population to multiple processors. `paretosearch` uses parallel computing when you:

- Have a license for Parallel Computing Toolbox software.
- Enable parallel computing with `parpool`, a Parallel Computing Toolbox function.
- Set the following option using `optimoptions`:
 - `UseParallel` is `true`.

When these conditions hold, `paretosearch` computes the objective function of the particles in a population in parallel.

Note Even when running in parallel, `paretosearch` occasionally calls the objective function serially on the host machine. Therefore, ensure that your objective function has no assumptions about whether it is evaluated in serial or parallel.

For algorithmic details, see “Modifications for Parallel Computation and Vectorized Function Evaluation” on page 14-16.

Surrogate Optimization

`surrogateopt` can automatically distribute the evaluation of the objective function associated with a population to multiple processors. `surrogateopt` uses parallel computing when you:

- Have a license for Parallel Computing Toolbox software.
- Enable parallel computing with `parpool`, a Parallel Computing Toolbox function.
- Set the following option using `optimoptions`:
 - `UseParallel` is `true`.

When these conditions hold, `surrogateopt` computes the objective function of the particles in a population in parallel.

Note Even when running in parallel, `surrogateopt` occasionally calls the objective function serially on the host machine. Therefore, ensure that your objective function has no assumptions about whether it is evaluated in serial or parallel.

For algorithmic details, see “Parallel surrogateopt Algorithm” on page 11-10.

See Also

More About

- “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11
- “Minimizing an Expensive Optimization Problem Using Parallel Computing Toolbox” on page 16-16

How to Use Parallel Processing in Global Optimization Toolbox

In this section...

“Multicore Processors” on page 16-11

“Processor Network” on page 16-12

“Parallel Search Functions or Hybrid Functions” on page 16-14

“Testing Parallel Optimization” on page 16-15

Multicore Processors

If you have a multicore processor, you can increase processing speed by using parallel processing. You can establish a parallel pool of several workers with a Parallel Computing Toolbox license. For a description of Parallel Computing Toolbox software, see “Get Started with Parallel Computing Toolbox” (Parallel Computing Toolbox).

Suppose you have a dual-core processor, and want to use parallel computing. Enter this code at the command line.

```
parpool
```

MATLAB starts a pool of workers using the multicore processor. If you previously set a nondefault cluster profile, you can enforce multicore (local) computing by entering this code.

```
parpool('local')
```

Note Depending on your preferences, MATLAB can start a parallel pool automatically. To enable this feature, select **Parallel > Parallel Preferences** in the **Environment** group on the **Home** tab, and then select **Automatically create a parallel pool**.

Set your solver to use parallel processing.

Solver	Command-Line Settings
ga	<code>options = optimoptions('ga','UseParallel', true, 'UseVectorized', false);</code>
gamultiobj	<code>options = optimoptions('gamultiobj','UseParallel', true, 'UseVectorized', false);</code>
MultiStart	<code>ms = MultiStart('UseParallel', true);</code> or <code>ms.UseParallel = true</code>
paretosearch	<code>options = optimoptions('paretosearch','UseParallel', true);</code>

Solver	Command-Line Settings
particleswarm	<code>options = optimoptions('particleswarm', 'UseParallel', true, 'UseVectorized', false);</code>
patternsearch	<code>options = optimoptions('patternsearch', 'UseParallel', true, 'UseCompletePoll', true, 'UseVectorized', false);</code>
surrogateopt	<code>options = optimoptions('surrogateopt', 'UseParallel', true);</code>

Beginning in R2019a, when you set the `UseParallel` option to `true`, `patternsearch` internally overrides the `UseCompletePoll` setting to `true` so that the function polls in parallel.

When you run an applicable solver with `options`, applicable solvers automatically use parallel computing.

To stop computing optimizations in parallel, set `UseParallel` to `false`. To halt all parallel computation, enter this code.

```
delete(gcf)
```

Note The documentation recommends not to use `parfor` or `parfeval` when calling Simulink; see “Using sim Function Within `parfor`” (Simulink). Therefore, you might encounter issues when optimizing a Simulink simulation in parallel using a solver's built-in parallel functionality.

Processor Network

If you have multiple processors on a network, use Parallel Computing Toolbox functions and MATLAB Parallel Server™ software to establish parallel computation.

Make sure your system is configured properly for parallel computing. Check with your systems administrator, or refer to the Parallel Computing Toolbox documentation.

- 1 Perform a basic check by entering this code, where `prof` is your cluster profile.

```
parpool(prof)
```

- 2 Workers must be able to access your objective function file and, if applicable, your nonlinear constraint function file. Complete one of these steps to ensure access:

- Distribute the files to the workers using the `parpool AttachedFiles` argument. In this example, `objfun.m` is your objective function file, and `constrfun.m` is your nonlinear constraint function file.

```
parpool('AttachedFiles', {'objfun.m', 'constrfun.m'});
```

Workers access their own copies of the files.

- Give a network file path to your objective or constraint function files.

```
pctRunOnAll('addpath network_file_path')
```

Workers access the function files over the network.

- 3 Check whether a file is on the path of every worker.

```
pctRunOnAll('which filename')
```

If any worker does not have a path to the file, it reports

```
filename not found.
```

Set your solver to use parallel processing.

Solver	Command-Line Settings
ga	<code>options = optimoptions('ga','UseParallel', true, 'UseVectorized', false);</code>
gamultiobj	<code>options = optimoptions('gamultiobj','UseParallel', true, 'UseVectorized', false);</code>
MultiStart	<code>ms = MultiStart('UseParallel', true);</code> or <code>ms.UseParallel = true</code>
paretosearch	<code>options = optimoptions('paretosearch','UseParallel', true);</code>
particleswarm	<code>options = optimoptions('particleswarm', 'UseParallel', true, 'UseVectorized', false);</code>
patternsearch	<code>options = optimoptions('patternsearch','UseParallel', true, 'UseCompletePoll', true, 'UseVectorized', false);</code>
surrogateopt	<code>options = optimoptions('surrogateopt','UseParallel', true);</code>

Beginning in R2019a, when you set the `UseParallel` option to `true`, `patternsearch` internally overrides the `UseCompletePoll` setting to `true` so that the function polls in parallel.

After you establish your parallel computing environment, applicable solvers automatically use parallel computing whenever you call them with `options`.

To stop computing optimizations in parallel, set `UseParallel` to `false`. To halt all parallel computation, enter this code.

```
delete(gcp)
```

Note The documentation recommends not to use `parfor` or `parfeval` when calling Simulink; see “Using sim Function Within `parfor`” (Simulink). Therefore, you might encounter issues when optimizing a Simulink simulation in parallel using a solver's built-in parallel functionality.

Parallel Search Functions or Hybrid Functions

To have a `patternsearch` search function run in parallel, or a hybrid function for `ga` or `simulannealbnd` run in parallel, do the following.

- 1 Set up parallel processing as described in “Multicore Processors” on page 16-11 or “Processor Network” on page 16-12.
- 2 Ensure that your search function or hybrid function has the conditions outlined in these sections:
 - “`patternsearch` Search Function” on page 16-14
 - “Parallel Hybrid Functions” on page 16-14

`patternsearch` Search Function

`patternsearch` uses a parallel search function under the following conditions:

- `UseCompleteSearch` is true.
- The search method is not `@searchneldermead` or `custom`.
- If the search method is a `patternsearch` poll method or Latin hypercube search, `UseParallel` is true. Set at the command line with `optimoptions`:

```
options = optimoptions('patternsearch','UseParallel',true,...
    'UseCompleteSearch',true,'SearchFcn',@GPSPositiveBasis2N);
```

- If the search method is `ga`, the search method option has `UseParallel` set to true. Set at the command line with `optimoptions`:

```
iterlim = 1; % iteration limit, specifies # ga runs
gaopt = optimoptions('ga','UseParallel',true);
options = optimoptions('patternsearch','SearchFcn',...
    {@searchga,iterlim,gaopt});
```

For more information about search options, see “Search Options” on page 17-12. For an example, see “Search and Poll” on page 6-42.

Parallel Hybrid Functions

`ga`, `particleswarm`, and `simulannealbnd` can have other solvers run after or interspersed with their iterations. These other solvers are called hybrid functions. For information on using a hybrid function with `gamultiobj`, see “Parallel Computing with `gamultiobj`” on page 16-8. Both `patternsearch` and `fmincon` can be hybrid functions. You can set options so that `patternsearch` runs in parallel, or `fmincon` estimates gradients in parallel.

Set the options for the hybrid function as described in “Hybrid Function Options” on page 17-39 for `ga`, “Hybrid Function” on page 17-47 for `particleswarm`, or “Hybrid Function Options” on page 17-61 for `simulannealbnd`. To summarize:

- If your hybrid function is `patternsearch`
 - 1 Create `patternsearch` options:


```
hybridopts = optimoptions('patternsearch','UseParallel',true,...
    'UseCompletePoll',true);
```
 - 2 Set the `ga` or `simulannealbnd` options to use `patternsearch` as a hybrid function:


```

options = optimoptions('ga','UseParallel',true); % for ga
options = optimoptions('ga',options,...
    'HybridFcn',{@patternsearch,hybridopts});
% or, for simulannealbnd:
options = optimoptions(@simulannealbnd,'HybridFcn',{@patternsearch,hybridopts});

```

For more information on parallel `patternsearch`, see “Pattern Search” on page 16-6.

- If your hybrid function is `fmincon`:

- 1 Create `fmincon` options:

```

hybridopts = optimoptions(@fmincon,'UseParallel',true,...
    'Algorithm','interior-point');
% You can use any Algorithm except trust-region-reflective

```

- 2 Set the `ga` or `simulannealbnd` options to use `fmincon` as a hybrid function:

```

options = optimoptions('ga','UseParallel',true);
options = optimoptions('ga',options,'HybridFcn',{@fmincon,hybridopts});
% or, for simulannealbnd:
options = optimoptions(@simulannealbnd,'HybridFcn',{@fmincon,hybridopts});

```

For more information on parallel `fmincon`, see “Parallel Computing”.

Testing Parallel Optimization

Follow these steps to test whether your problem runs correctly in parallel.

- 1 Try your problem without parallel computation to ensure that it runs serially. Make sure this test is successful (gives correct results) before going to the next test.
- 2 Set `UseParallel` to `true`, and ensure that no parallel pool exists by entering `delete(gcp)`. To make sure that MATLAB does not create a parallel pool, select **Parallel > Parallel Preferences** in the **Environment** group on the **Home** tab, and then clear **Automatically create a parallel pool**. Your problem runs `parfor` serially, with loop iterations in reverse order from a `for` loop. Make sure this test is successful (gives correct results) before going to the next test.
- 3 Set `UseParallel` to `true`, and create a parallel pool using `parpool`. Unless you have a multicore processor or a network set up, this test does not increase processing speed. This testing is simply to verify the correctness of the computations.

Remember to call your solver using an `options` argument to test or use parallel functionality.

See Also

More About

- “How Solvers Compute in Parallel” on page 16-2
- “Minimizing an Expensive Optimization Problem Using Parallel Computing Toolbox” on page 16-16

Minimizing an Expensive Optimization Problem Using Parallel Computing Toolbox

This example shows how to speed up the minimization of an expensive optimization problem using functions in Optimization Toolbox™ and Global Optimization Toolbox. In the first part of the example we solve the optimization problem by evaluating functions in a serial fashion, and in the second part of the example we solve the same problem using the parallel for loop (`parfor`) feature by evaluating functions in parallel. We compare the time taken by the optimization function in both cases.

Expensive Optimization Problem

For the purpose of this example, we solve a problem in four variables, where the objective and constraint functions are made artificially expensive by pausing.

```
function f = expensive_objfun(x)
%EXPENSIVE_OBJFUN An expensive objective function used in optimparfor example.

% Copyright 2007-2013 The MathWorks, Inc.

% Simulate an expensive function by pausing
pause(0.1)
% Evaluate objective function
f = exp(x(1)) * (4*x(3)^2 + 2*x(4)^2 + 4*x(1)*x(2) + 2*x(2) + 1);

function [c,ceq] = expensive_confun(x)
%EXPENSIVE_CONFUN An expensive constraint function used in optimparfor example.

% Copyright 2007-2013 The MathWorks, Inc.

% Simulate an expensive function by pausing
pause(0.1);
% Evaluate constraints
c = [1.5 + x(1)*x(2)*x(3) - x(1) - x(2) - x(4);
     -x(1)*x(2) + x(4) - 10];
% No nonlinear equality constraints:
ceq = [];
```

Minimizing Using `fmincon`

We are interested in measuring the time taken by `fmincon` in serial so that we can compare it to the parallel time.

```
startPoint = [-1 1 1 -1];
options = optimoptions('fmincon','Display','iter','Algorithm','interior-point');
startTime = tic;
xsol = fmincon(@expensive_objfun,startPoint,[],[],[],[],[],[],[],@expensive_confun,options);
time_fmincon_sequential = toc(startTime);
fprintf('Serial FMINCON optimization takes %g seconds.\n',time_fmincon_sequential);
```

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of step
------	---------	------	-------------	------------------------	--------------

0	5	1.839397e+00	1.500e+00	3.211e+00	
1	11	-9.760099e-01	3.708e+00	7.902e-01	2.362e+00
2	16	-1.480976e+00	0.000e+00	8.344e-01	1.069e+00
3	21	-2.601599e+00	0.000e+00	8.390e-01	1.218e+00
4	29	-2.823630e+00	0.000e+00	2.598e+00	1.118e+00
5	34	-3.905338e+00	0.000e+00	1.210e+00	7.302e-01
6	39	-6.212992e+00	3.934e-01	7.372e-01	2.405e+00
7	44	-5.948761e+00	0.000e+00	1.784e+00	1.905e+00
8	49	-6.940062e+00	1.233e-02	7.668e-01	7.553e-01
9	54	-6.973887e+00	0.000e+00	2.549e-01	3.920e-01
10	59	-7.142993e+00	0.000e+00	1.903e-01	4.735e-01
11	64	-7.155325e+00	0.000e+00	1.365e-01	2.626e-01
12	69	-7.179122e+00	0.000e+00	6.336e-02	9.115e-02
13	74	-7.180116e+00	0.000e+00	1.069e-03	4.670e-02
14	79	-7.180409e+00	0.000e+00	7.797e-04	2.815e-03
15	84	-7.180410e+00	0.000e+00	6.368e-06	3.120e-04

Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in feasible directions, to within the value of the optimality tolerance, and constraints are satisfied to within the value of the constraint tolerance.

Serial FMINCON optimization takes 18.2876 seconds.

Minimizing Using Genetic Algorithm

Since `ga` usually takes many more function evaluations than `fmincon`, we remove the expensive constraint from this problem and perform unconstrained optimization instead. Pass empty matrices `[]` for constraints. In addition, we limit the maximum number of generations to 15 for `ga` so that `ga` can terminate in a reasonable amount of time. We are interested in measuring the time taken by `ga` so that we can compare it to the parallel `ga` evaluation. Note that running `ga` requires Global Optimization Toolbox.

```
rng default % for reproducibility
try
    gaAvailable = false;
    nvar = 4;
    gaoptions = optimoptions('ga','MaxGenerations',15,'Display','iter');
    startTime = tic;
    gasol = ga(@expensive_objfun,nvar,[],[],[],[],[],[],[],[],gaoptions);
    time_ga_sequential = toc(startTime);
    fprintf('Serial GA optimization takes %g seconds.\n',time_ga_sequential);
    gaAvailable = true;
catch ME
    warning(message('optim:optimdemos:optimparfor:gaNotFound'));
end
```

Single objective optimization:
4 Variable(s)

```
Options:
CreationFcn: @gacreationuniform
CrossoverFcn: @crossoverscattered
SelectionFcn: @selectionstochunif
MutationFcn: @mutationgaussian
```

Generation	Func-count	Best f(x)	Mean f(x)	Stall Generations
1	100	-5.546e+05	1.483e+15	0
2	147	-5.581e+17	-1.116e+16	0
3	194	-7.556e+17	6.679e+22	0
4	241	-7.556e+17	-7.195e+16	1
5	288	-9.381e+27	-1.876e+26	0
6	335	-9.673e+27	-7.497e+26	0
7	382	-4.511e+36	-9.403e+34	0
8	429	-5.111e+36	-3.011e+35	0
9	476	-7.671e+36	9.346e+37	0
10	523	-1.52e+43	-3.113e+41	0
11	570	-2.273e+45	-4.67e+43	0
12	617	-2.589e+47	-6.281e+45	0
13	664	-2.589e+47	-1.015e+46	1
14	711	-8.149e+47	-5.855e+46	0
15	758	-9.503e+47	-1.29e+47	0

Optimization terminated: maximum number of generations exceeded.
Serial GA optimization takes 81.5878 seconds.

Setting Parallel Computing Toolbox

The finite differencing used by the functions in Optimization Toolbox to approximate derivatives is done in parallel using the `parfor` feature if Parallel Computing Toolbox™ is available and there is a parallel pool of workers. Similarly, `ga`, `gamultiobj`, and `patternsearch` solvers in Global Optimization Toolbox evaluate functions in parallel. To use the `parfor` feature, we use the `parpool` function to set up the parallel environment. The computer on which this example is published has four cores, so `parpool` starts four MATLAB® workers. If there is already a parallel pool when you run this example, we use that pool; see the documentation for `parpool` for more information.

```
if max(size(gcf)) == 0 % parallel pool needed
    parpool % create the parallel pool
end
```

Minimizing Using Parallel `fmincon`

To minimize our expensive optimization problem using the parallel `fmincon` function, we need to explicitly indicate that our objective and constraint functions can be evaluated in parallel and that we want `fmincon` to use its parallel functionality wherever possible. Currently, finite differencing can be done in parallel. We are interested in measuring the time taken by `fmincon` so that we can compare it to the serial `fmincon` run.

```
options = optimoptions(options, 'UseParallel', true);
startTime = tic;
xsol = fmincon(@expensive_objfun, startPoint, [], [], [], [], [], [], @expensive_confun, options);
time_fmincon_parallel = toc(startTime);
fprintf('Parallel FMINCON optimization takes %g seconds.\n', time_fmincon_parallel);
```

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of step
0	5	1.839397e+00	1.500e+00	3.211e+00	
1	11	-9.760099e-01	3.708e+00	7.902e-01	2.362e+00
2	16	-1.480976e+00	0.000e+00	8.344e-01	1.069e+00
3	21	-2.601599e+00	0.000e+00	8.390e-01	1.218e+00
4	29	-2.823630e+00	0.000e+00	2.598e+00	1.118e+00
5	34	-3.905338e+00	0.000e+00	1.210e+00	7.302e-01
6	39	-6.212992e+00	3.934e-01	7.372e-01	2.405e+00

7	44	-5.948761e+00	0.000e+00	1.784e+00	1.905e+00
8	49	-6.940062e+00	1.233e-02	7.668e-01	7.553e-01
9	54	-6.973887e+00	0.000e+00	2.549e-01	3.920e-01
10	59	-7.142993e+00	0.000e+00	1.903e-01	4.735e-01
11	64	-7.155325e+00	0.000e+00	1.365e-01	2.626e-01
12	69	-7.179122e+00	0.000e+00	6.336e-02	9.115e-02
13	74	-7.180116e+00	0.000e+00	1.069e-03	4.670e-02
14	79	-7.180409e+00	0.000e+00	7.797e-04	2.815e-03
15	84	-7.180410e+00	0.000e+00	6.368e-06	3.120e-04

Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in feasible directions, to within the value of the optimality tolerance, and constraints are satisfied to within the value of the constraint tolerance.

Parallel FMINCON optimization takes 8.79291 seconds.

Minimizing Using Parallel Genetic Algorithm

To minimize our expensive optimization problem using the `ga` function, we need to explicitly indicate that our objective function can be evaluated in parallel and that we want `ga` to use its parallel functionality wherever possible. To use the parallel `ga` we also require that the 'Vectorized' option be set to the default (i.e., 'off'). We are again interested in measuring the time taken by `ga` so that we can compare it to the serial `ga` run. Though this run may be different from the serial one because `ga` uses a random number generator, the number of expensive function evaluations is the same in both runs. Note that running `ga` requires Global Optimization Toolbox.

```
rng default % to get the same evaluations as the previous run
if gaAvailable
    gaoptions = optimoptions(gaoptions,'UseParallel',true);
    startTime = tic;
    gasol = ga(@expensive_objfun,nvar,[],[],[],[],[],[],[],gaoptions);
    time_ga_parallel = toc(startTime);
    fprintf('Parallel GA optimization takes %g seconds.\n',time_ga_parallel);
end
```

Single objective optimization:
4 Variable(s)

Options:
CreationFcn: @gacreationuniform
CrossoverFcn: @crossoverscattered
SelectionFcn: @selectionstochunif
MutationFcn: @mutationgaussian

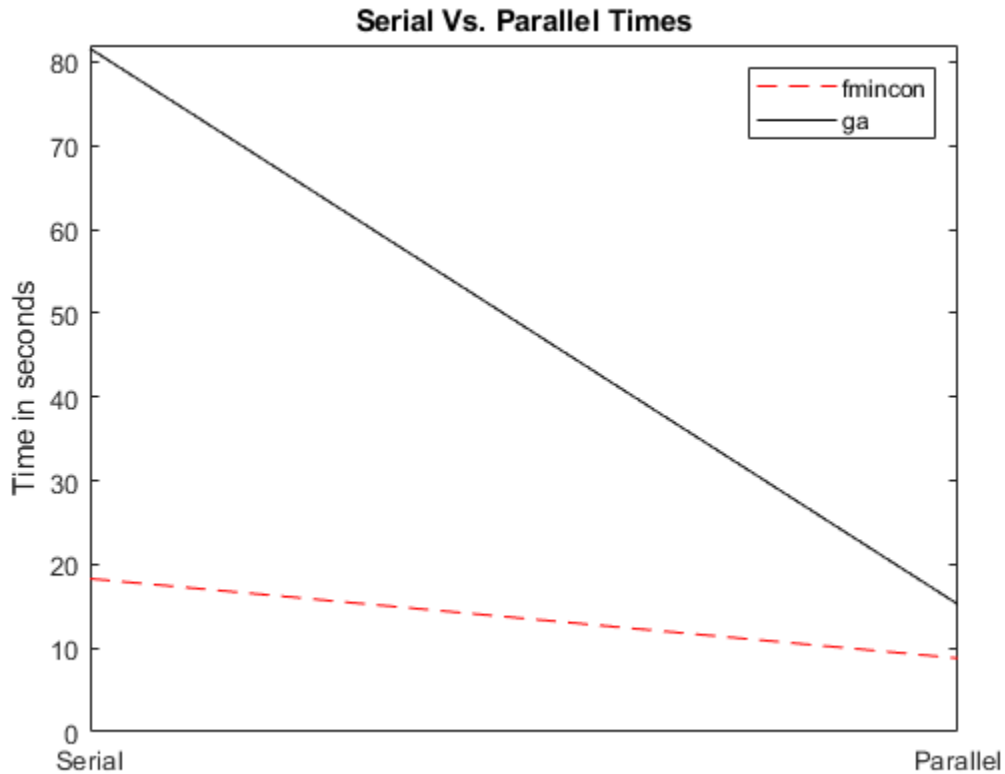
Generation	Func-count	Best f(x)	Mean f(x)	Stall Generations
1	100	-5.546e+05	1.483e+15	0
2	147	-5.581e+17	-1.116e+16	0
3	194	-7.556e+17	6.679e+22	0
4	241	-7.556e+17	-7.195e+16	1
5	288	-9.381e+27	-1.876e+26	0
6	335	-9.673e+27	-7.497e+26	0
7	382	-4.511e+36	-9.403e+34	0
8	429	-5.111e+36	-3.011e+35	0
9	476	-7.671e+36	9.346e+37	0

10	523	-1.52e+43	-3.113e+41	0
11	570	-2.273e+45	-4.67e+43	0
12	617	-2.589e+47	-6.281e+45	0
13	664	-2.589e+47	-1.015e+46	1
14	711	-8.149e+47	-5.855e+46	0
15	758	-9.503e+47	-1.29e+47	0

Optimization terminated: maximum number of generations exceeded.
Parallel GA optimization takes 15.2253 seconds.

Compare Serial and Parallel Time

```
X = [time_fmincon_sequential time_fmincon_parallel];
Y = [time_ga_sequential time_ga_parallel];
t = [0 1];
plot(t,X,'r--',t,Y,'k-')
ylabel('Time in seconds')
legend('fmincon','ga')
ax = gca;
ax.XTick = [0 1];
ax.XTickLabel = {'Serial' 'Parallel'};
axis([0 1 0 ceil(max([X Y]))])
title('Serial Vs. Parallel Times')
```



Utilizing parallel function evaluation via `parfor` improved the efficiency of both `fmincon` and `ga`. The improvement is typically better for expensive objective and constraint functions.

See Also

More About

- “How Solvers Compute in Parallel” on page 16-2
- “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11

Options Reference

- “GlobalSearch and MultiStart Properties (Options)” on page 17-2
- “Pattern Search Options” on page 17-7
- “Genetic Algorithm Options” on page 17-23
- “Particle Swarm Options” on page 17-45
- “Surrogate Optimization Options” on page 17-51
- “Simulated Annealing Options” on page 17-58
- “Options Changes in R2016a” on page 17-65

GlobalSearch and MultiStart Properties (Options)

In this section...

“How to Set Properties” on page 17-2

“Properties of Both Objects” on page 17-2

“GlobalSearch Properties” on page 17-5

“MultiStart Properties” on page 17-6

How to Set Properties

To create a `GlobalSearch` or `MultiStart` object with nondefault properties, use name-value pairs. For example, to create a `GlobalSearch` object that has iterative display and runs only from feasible points with respect to bounds and inequalities, enter

```
gs = GlobalSearch("Display","iter", ...
    "StartPointsToRun","bounds-ineqs");
```

To set a property of an existing `GlobalSearch` or `MultiStart` object, use dot notation. For example, if `ms` is a `MultiStart` object, and you want to set the `Display` property to `"iter"`, enter

```
ms.Display = "iter";
```

To set multiple properties of an existing object simultaneously, use the constructor (`GlobalSearch` or `MultiStart`) with name-value arguments. For example, to set the `Display` property to `"iter"` and the `MaxTime` property to `100`, enter

```
ms = MultiStart(ms,Display="iter",MaxTime=100);
```

For more information on setting properties, see “Changing Global Options” on page 4-52.

Properties of Both Objects

You can create a `MultiStart` object from a `GlobalSearch` object and vice-versa.

The syntax for creating a new object from an existing object is:

```
ms = MultiStart(gs);
or
gs = GlobalSearch(ms);
```

The new object contains the properties that apply of the old object. This section describes those shared properties:

- “Display” on page 17-3
- “FunctionTolerance” on page 17-3
- “MaxTime” on page 17-3
- “OutputFcn” on page 17-3
- “PlotFcn” on page 17-4
- “StartPointsToRun” on page 17-4

- "XTolerance" on page 17-5

Display

Values for the `Display` property are:

- "final" (default) — Summary results to command line after last solver run.
- "off" — No output to command line.
- "iter" — Summary results to command line after each local solver run.

FunctionTolerance

The `FunctionTolerance` property describes how close two objective function values must be for solvers to consider them identical for creating the vector of local solutions. Set `FunctionTolerance` to 0 to obtain the results of every local solver run. Set `FunctionTolerance` to a larger value to have fewer results.

Solvers consider two solutions identical if they are within `XTolerance` distance of each other and have objective function values within `FunctionTolerance` of each other. If both conditions are not met, solvers report the solutions as distinct. The tolerances are relative, not absolute. For details, see "When `fmincon` Runs" on page 4-37 for `GlobalSearch`, and "Create `GlobalOptimSolution` Object" on page 4-39 for `MultiStart`.

MaxTime

The `MaxTime` property describes a tolerance on the number of seconds since the solver began its run. Solvers halt when they see `MaxTime` seconds have passed since the beginning of the run. Time means *wall clock* as opposed to processor cycles. The default is `Inf`.

OutputFcn

The `OutputFcn` property directs the global solver to run one or more output functions after each local solver run completes. The output functions also run when the global solver starts and ends. Include a handle to an output function written in the appropriate syntax, or include a cell array of such handles. The default is an empty entry (`[]`).

The syntax of an output function is:

```
stop = outFcn(optimValues,state)
```

- `stop` is a Boolean. When `true`, the algorithm stops. When `false`, the algorithm continues.

Note A local solver can have an output function. The global solver does not necessarily stop when a local solver output function causes a local solver run to stop. If you want the global solver to stop in this case, have the global solver output function stop when `optimValues.localsolution.exitflag=-1`.

- `optimValues` is a structure, described in "optimValues Structure" on page 17-4.
- `state` is the current state of the global algorithm:
 - "init" — The global solver has not called the local solver. The fields in the `optimValues` structure are empty, except for `localrunindex`, which is 0, and `funccount`, which contains the number of objective and constraint function evaluations.

- "iter" — The global solver calls output functions after each local solver run.
- "done" — The global solver finished calling local solvers. The fields in `optimValues` generally have the same values as the ones from the final output function call with `state="iter"`. However, the value of `optimValues.funccount` for `GlobalSearch` can be larger than the value in the last function call with "iter", because the `GlobalSearch` algorithm might have performed some function evaluations that were not part of a local solver. For more information, see "GlobalSearch Algorithm" on page 4-35.

For an example using an output function, see "GlobalSearch Output Function" on page 4-27.

Note Output and plot functions do not run when `MultiStart` has the `UseParallel` option set to `true` and there is an open `parpool`.

optimValues Structure

The `optimValues` structure contains the following fields:

- `bestx` — The current best point
- `bestfval` — Objective function value at `bestx`
- `funccount` — Total number of function evaluations
- `localrunindex` — Index of the local solver run
- `localsolution` — A structure containing part of the output of the local solver call: `X`, `Fval` and `Exitflag`

PlotFcn

The `PlotFcn` property directs the global solver to run one or more plot functions after each local solver run completes. Include a handle to a plot function written in the appropriate syntax, or include a cell array of such handles. The default is an empty entry (`[]`).

The syntax of a plot function is the same as that of an output function. For details, see "OutputFcn" on page 17-3.

There are two predefined plot functions for the global solvers:

- `@gsplotbestf` plots the best objective function value.
- `@gsplotfunccount` plots the number of function evaluations.

For an example using a plot function, see "MultiStart Plot Function" on page 4-30.

If you specify more than one plot function, all plots appear as subplots in the same window. Right-click any subplot to obtain a larger version in a separate figure window.

Note Output and plot functions do not run when `MultiStart` has the `UseParallel` option set to `true` and there is an open `parpool`.

StartPointsToRun

The `StartPointsToRun` property directs the solver to exclude certain start points from being run:

- `all` — Accept all start points.
- `bounds` — Reject start points that do not satisfy bounds.
- `bounds-ineqs` — Reject start points that do not satisfy bounds or inequality constraints.

XTolerance

The `XTolerance` property describes how close two points must be for solvers to consider them identical for creating the vector of local solutions. Set `XTolerance` to 0 to obtain the results of every local solver run. Set `XTolerance` to a larger value to have fewer results. Solvers compute the distance between a pair of points with `norm`, the Euclidean distance.

Solvers consider two solutions identical if they are within `XTolerance` distance of each other and have objective function values within `FunctionTolerance` of each other. If both conditions are not met, solvers report the solutions as distinct. The tolerances are relative, not absolute. For details, see “When `fmincon` Runs” on page 4-37 for `GlobalSearch`, and “Create `GlobalOptimSolution` Object” on page 4-39 for `MultiStart`.

GlobalSearch Properties

- “`NumTrialPoints`” on page 17-5
- “`NumStageOnePoints`” on page 17-5
- “`MaxWaitCycle`” on page 17-5
- “`BasinRadiusFactor`” on page 17-6
- “`DistanceThresholdFactor`” on page 17-6
- “`PenaltyThresholdFactor`” on page 17-6

NumTrialPoints

Number of potential start points to examine in addition to `x0` from the `problem` structure. `GlobalSearch` runs only those potential start points that pass several tests. For more information, see “`GlobalSearch` Algorithm” on page 4-35.

Default: 1000

NumStageOnePoints

Number of start points in Stage 1. For details, see “Obtain Stage 1 Start Point, Run” on page 4-36.

Default: 200

MaxWaitCycle

A positive integer tolerance appearing in several points in the algorithm.

- If the observed penalty function of `MaxWaitCycle` consecutive trial points is at least the penalty threshold, then raise the penalty threshold (see “`PenaltyThresholdFactor`” on page 17-6).
- If `MaxWaitCycle` consecutive trial points are in a basin, then update that basin's radius (see “`BasinRadiusFactor`” on page 17-6).

Default: 20

BasinRadiusFactor

A basin radius decreases after `MaxWaitCycle` consecutive start points are within the basin. The basin radius decreases by a factor of $1 - \text{BasinRadiusFactor}$.

Default: 0.2

DistanceThresholdFactor

A multiplier for determining whether a trial point is in an existing basin of attraction. For details, see “Examine Stage 2 Trial Point to See if `fmincon` Runs” on page 4-36. Default: 0.75

PenaltyThresholdFactor

Determines increase in penalty threshold. For details, see `React to Large Counter Values` on page 4-38.

Default: 0.2

MultiStart Properties**UseParallel**

The `UseParallel` property determines whether the solver distributes start points to multiple processors:

- `false` (default) — Do not run in parallel.
- `true` — Run in parallel.

For the solver to run in parallel you must set up a parallel environment with `parpool`. For details, see “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11.

Pattern Search Options

In this section...

"Options for Pattern Search" on page 17-7
 "Algorithm Options" on page 17-7
 "Plot Options" on page 17-8
 "Poll Options" on page 17-10
 "Multiobjective Options" on page 17-11
 "Search Options" on page 17-12
 "Mesh Options" on page 17-15
 "Constraint Parameters" on page 17-16
 "Cache Options" on page 17-16
 "Stopping Criteria" on page 17-17
 "Output Function Options" on page 17-17
 "Display to Command Window Options" on page 17-19
 "Vectorized and Parallel Options" on page 17-19
 "Options Table for Pattern Search Algorithms" on page 17-21

Options for Pattern Search

Set options for `patternsearch` by using `optimoptions`.

```
options = optimoptions("patternsearch",...
    Option1=value1,Option2=value2);
```

- Some options are listed in *italics*. These options do not appear in the listing that `optimoptions` returns. To see why `optimoptions` hides these option values, see "Options that `optimoptions` Hides" on page 17-65.
- Ensure that you pass options to the solver. Otherwise, `patternsearch` uses the default option values.

```
[x,fval] = patternsearch(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
```

Algorithm Options

Algorithm specifies the algorithm used by `patternsearch` to solve a problem.

- "classic" — Use the original algorithm as described in "How Pattern Search Polling Works" on page 6-27.
- "nups" — Use the Nonuniform Pattern Search algorithm as described in "Nonuniform Pattern Search (NUPS) Algorithm" on page 6-35.
- "nups-gps" — Use the Nonuniform Pattern Search algorithm restricted to the GPS (generalized pattern search) polling algorithm (no OrthoMADS (orthogonal mesh adaptive direct search) polling).
- "nups-mads" — Use the Nonuniform Pattern Search algorithm restricted to the OrthoMADS polling algorithm (no GPS polling).

For examples of algorithm effects, see “Explore patternsearch Algorithms” on page 6-103 and “Explore patternsearch Algorithms in Optimize Live Editor Task” on page 6-107.

Plot Options

`PlotFcn` specifies the plot function or functions called at each iteration by `patternsearch` or `paretosearch`. Set the `PlotFcn` option to be a built-in plot function name or a handle to the plot function. You can stop the algorithm at any time by clicking the **Stop** button on the plot window. For example, to display the best function value, set `options` as follows:

```
options = optimoptions("patternsearch",PlotFcn="psplotbestf");
```

To display multiple plots, use a cell array of built-in plot function names or a cell array of function handles:

```
options = optimoptions("patternsearch",...
    PlotFcn={@plotfun1, @plotfun2, ...});
```

where `@plotfun1`, `@plotfun2`, and so on are function handles to the plot functions. If you specify more than one plot function, all plots appear as subplots in the same window. Right-click any subplot to obtain a larger version in a separate figure window.

Available plot functions for `patternsearch` or for `paretosearch` with a single objective function:

- "psplotbestf" — Plot the best objective function value.
- "psplotfuncount" — Plot the number of function evaluations.
- "psplotmeshsize" — Plot the mesh size.
- "psplotbestx" — Plot the current best point.
- "psplotmaxconstr" — Plot the maximum nonlinear constraint violation.
- You can also create and use your own plot function. “Structure of the Plot Functions” on page 17-8 describes the structure of a custom plot function. Pass any custom function as a function handle.

For `paretosearch` with multiple objective functions, you can select a custom function that you pass as a function handle, or any of the following functions.

- "psplotfuncount" — Plot the number of function evaluations.
- "psplotmaxconstr" — Plot the maximum nonlinear constraint violation.
- "psplotdistance" — Plot the distance metric. See “paretosearch Algorithm” on page 14-10.
- "psplotparetof" — Plot the objective function values. Applies to three or fewer objectives.
- "psplotparetox" — Plot the current points in parameter space. Applies to three or fewer dimensions.
- "psplotspread" — Plot the spread metric. See “paretosearch Algorithm” on page 14-10.
- "psplotvolume" — Plot the volume metric. See “paretosearch Algorithm” on page 14-10.

For `patternsearch`, the `PlotInterval` option specifies the number of iterations between consecutive calls to the plot function.

Structure of the Plot Functions

The first line of a plot function has the form

```
function stop = plotfun(optimvalues, flag)
```

The input arguments to the function are

- `optimvalues` — Structure containing information about the current state of the solver. The structure contains the following fields for `patternsearch`:
 - `x` — Current point
 - `iteration` — Iteration number
 - `fval` — Objective function value
 - `meshsize` — Current mesh size
 - `funccount` — Number of function evaluations
 - `method` — Method used in last iteration
 - `TolFun` — Tolerance on function value in last iteration
 - `TolX` — Tolerance on `x` value in last iteration
 - `nonlineq` — Nonlinear inequality constraints, displayed only when a nonlinear constraint function is specified
 - `nonlineq` — Nonlinear equality constraints, displayed only when a nonlinear constraint function is specified

The structure contains the following fields for `paretosearch`:

- `x` — Current point
- `fval` — Objective function value
- `iteration` — Iteration number
- `funccount` — Number of function evaluations
- `nonlineq` — Nonlinear inequality constraints, displayed only when a nonlinear constraint function is specified
- `nonlineq` — Nonlinear equality constraints, always empty []
- `volume` — Volume measure (see “Definitions for `paretosearch` Algorithm” on page 14-10)
- `averagedistance` — Distance measure (see “Definitions for `paretosearch` Algorithm” on page 14-10)
- `spread` — Spread measure (see “Definitions for `paretosearch` Algorithm” on page 14-10)
- `flag` — Current state in which the plot function is called. The possible values for `flag` are
 - `"init"` — Initialization state
 - `"iter"` — Iteration state
 - `"interrupt"` — Intermediate stage
 - `"done"` — Final state

For details of `flag`, see “Structure of the Output Function” on page 17-18.

“Passing Extra Parameters” explains how to provide additional parameters to the function.

The output argument `stop` provides a way to stop the algorithm at the current iteration. `stop` can have the following values:

- `false` — The algorithm continues to the next iteration.
- `true` — The algorithm terminates at the current iteration.

Poll Options

Poll options control how the pattern search polls the mesh points at each iteration.

`PollMethod` specifies the pattern the algorithm uses to create the mesh. There are two patterns for each of the classes of direct search algorithms: the generalized pattern search (GPS) algorithm, the generating set search (GSS) algorithm, and the mesh adaptive direct search (MADS) algorithm. These patterns are the Positive basis $2N$ and the Positive basis $N+1$:

- The default pattern for `patternsearch`, "`GPSPositiveBasis2N`", consists of the following $2N$ vectors, where N is the number of independent variables for the objective function.
 $[1\ 0\ 0\dots 0]\ [0\ 1\ 0\dots 0]\ \dots\ [0\ 0\ 0\dots 1]\ [-1\ 0\ 0\dots 0]\ [0\ -1\ 0\dots 0]\ [0\ 0\ 0\dots -1]$.

For example, if the optimization problem has three independent variables, the pattern consists of the following six vectors.

$[1\ 0\ 0]\ [0\ 1\ 0]\ [0\ 0\ 1]\ [-1\ 0\ 0]\ [0\ -1\ 0]\ [0\ 0\ -1]$.

- The default pattern for `paretosearch`, "`GPSPositiveBasis2Np2`", is the same as "`GPSPositiveBasis2N`" with two more points: all ones and all minus ones.
 $[1\ 1\ 1\dots 1]\ [-1\ -1\ -1\dots -1]$

For example, if the optimization problem has three independent variables, the pattern consists of the following eight vectors.

$[1\ 0\ 0]\ [0\ 1\ 0]\ [0\ 0\ 1]\ [-1\ 0\ 0]\ [0\ -1\ 0]\ [0\ 0\ -1]\ [1\ 1\ 1]\ [-1\ -1\ -1]$.

- The "`GSSPositiveBasis2N`" pattern is similar to "`GPSPositiveBasis2N`", but adjusts the basis vectors to account for linear constraints. "`GSSPositiveBasis2N`" is more efficient than "`GPSPositiveBasis2N`" when the current point is near a linear constraint boundary. `paretosearch` also has the "`GSSPositiveBasis2Np2`" pattern that is similar to "`GPSPositiveBasis2Np2`".
- The "`MADSPositiveBasis2N`" pattern consists of $2N$ randomly generated vectors, where N is the number of independent variables for the objective function. This is done by randomly generating N vectors which form a linearly independent set, then using this first set and the negative of this set gives $2N$ vectors. As shown above, the "`GPSPositiveBasis2N`" pattern is formed using the positive and negative of the linearly independent identity, however, with the "`MADSPositiveBasis2N`", the pattern is generated using a random permutation of an N -by- N linearly independent lower triangular matrix that is regenerated at each iteration.

Note You cannot use MADS polling when the problem has linear equality constraints.

- The "`GPSPositiveBasisNp1`" pattern consists of the following $N + 1$ vectors.
 $[1\ 0\ 0\dots 0]\ [0\ 1\ 0\dots 0]\ \dots\ [0\ 0\ 0\dots 1]\ [-1\ -1\ -1\dots -1]$.

For example, if the objective function has three independent variables, the pattern consists of the following four vectors.

$[1\ 0\ 0]\ [0\ 1\ 0]\ [0\ 0\ 1]\ [-1\ -1\ -1]$.

- The "`GSSPositiveBasisNp1`" pattern is similar to "`GPSPositiveBasisNp1`", but adjusts the basis vectors to account for linear constraints. "`GSSPositiveBasisNp1`" is more efficient than "`GPSPositiveBasisNp1`" when the current point is near a linear constraint boundary.

- The "MADSPositiveBasisNp1" pattern consists of N randomly generated vectors to form the positive basis, where N is the number of independent variables for the objective function. Then, one more random vector is generated, giving $N+1$ randomly generated vectors. Each iteration generates a new pattern when the "MADSPositiveBasisNp1" is selected.

Note You cannot use MADS polling when the problem has linear equality constraints.

- The "OrthoMADSPositiveBasis2N" pattern is the same as the "GPSPositiveBasis2N" pattern followed by a random rotation in N dimensions.
- The "OrthoMADSPositiveBasisNp1" pattern is the same as the "GPSPositiveBasisNp1" pattern followed by a random rotation in N dimensions.

`UseCompletePoll` specifies whether all the points in the current mesh must be polled at each iteration. `UseCompletePoll` can have the values `true` or `false`. `UseCompletePoll` applies only when `Algorithm` is "classic".

- If you set `UseCompletePoll` to `true`, the algorithm polls all the points in the mesh at each iteration and chooses the point with the smallest objective function value as the current point at the next iteration.
- If you set `UseCompletePoll` to `false`, the default value, the algorithm stops the poll as soon as it finds a point whose objective function value is less than that of the current point. The algorithm then sets that point as the current point at the next iteration.
- For paretosearch only, the `MinPollFraction` option specifies the fraction of poll directions that are investigated during a poll, instead of the binary value of `UseCompletePoll`. To specify a complete poll, set `MinPollFraction` to 1. To specify that the poll stops as soon as it finds a point that improves all objective functions, set `MinPollFraction` to 0.

`PollOrderAlgorithm` specifies the order in which the algorithm searches the points in the current mesh. `PollOrderAlgorithm` applies only when `Algorithm` is "classic". The options are

- "Consecutive" (default) — The algorithm polls the mesh points in *consecutive* order, that is, the order of the pattern vectors as described in "Poll Method" on page 6-32.
- "Random" — The polling order is random.
- "Success" — The first search direction at each iteration is the direction in which the algorithm found the best point at the previous iteration. After the first point, the algorithm polls the mesh points in the same order as "Consecutive".

Multiobjective Options

The paretosearch solver mainly uses `patternsearch` options. Several of the available built-in plot functions differ; see "Plot Options" on page 17-8. The following options apply only to `paretosearch`.

In the table, N represents the number of decision variables.

Multiobjective Pattern Search Options

Option	Definition	Allowed and {Default} Values
ParetoSetSize	Number of points in the Pareto set.	Positive integer {max(60, number of objectives) }
ParetoSetChangeTolerance	Tolerance on the change in volume or spread of solutions. When either of these measures relatively changes by less than ParetoSetChangeTolerance, the iterations end. For details, see "Stopping Conditions" on page 14-15.	Positive scalar {1e-4}
MinPollFraction	Minimum fraction of the pattern to poll.	Scalar from 0 through 1 {0}
InitialPoints	<p>Initial points for paretosearch. Use one of these data types:</p> <ul style="list-style-type: none"> Matrix with nvars columns, where each row represents one initial point. Structure containing the following fields (all fields are optional except X0): <ul style="list-style-type: none"> X0 — Matrix with nvars columns, where each row represents one initial point. Fvals — Matrix with numObjectives columns, where each row represents the objective function values at the corresponding point in X0. Cineq — Matrix with numIneq columns, where each row represents the nonlinear inequality constraint values at the corresponding point in X0. <p>If there are missing entries in the Fvals or Cineq fields, paretosearch computes the missing values.</p>	Matrix with nvars columns structure {[]}

Search Options

The SearchFcn option specifies an optional search that the algorithm can perform at each iteration prior to the polling. If the search returns a point that improves the objective function, the algorithm uses that point at the next iteration and omits the polling. If you select the same SearchFcn and PollMethod, only the Poll method is used, although both are used when the selected options differ.

You can select a poll method as a search method only for the "classic" algorithm.

The values for SearchFcn are listed below.

- [], the default, specifies no search step.
- Any built-in poll algorithm: "GPSPositiveBasis2N", "GPSPositiveBasisNp1", "GSSPositiveBasis2N", "GSSPositiveBasisNp1", "MADSPositiveBasis2N", "MADSPositiveBasisNp1", "OrthoMADSPositiveBasis2N", or "OrthoMADSPositiveBasisNp1".
- "searchga" specifies a search using the genetic algorithm. You can modify the genetic algorithm search using two additional parameters:

```
options = optimoptions("patternsearch",SearchFcn=...
    {@searchga,iterlim,optionsGA})
```

- `iterlim` — Positive integer specifying the number of iterations of the pattern search for which the genetic algorithm search is performed. The default for `iterlim` is 1. The recommendation is not to change this value, because performing these time-consuming searches more than once is not likely to improve results.
- `optionsGA` — Options for the genetic algorithm, which you can set using `optimoptions`. If you do not specify any `searchga` options, then `searchga` uses the same `UseParallel` and `UseVectorized` option settings as `patternsearch`.
- "searchlhs" specifies a Latin hypercube search. `patternsearch` generates each point for the search as follows. For each component, take a random permutation of the vector [1,2,...,k] minus `rand(1,k)`, divided by `k`. (`k` is the number of points.) This yields `k` points, with each component close to evenly spaced. The resulting points are then scaled to fit any bounds. Latin hypercube uses default bounds of -1 and 1.

The way the search is performed depends on the setting for the `UseCompleteSearch` option.

- If you set `UseCompleteSearch` to `true`, the algorithm polls all the points that are randomly generated at each iteration by the Latin hypercube search and chooses the one with the smallest objective function value.
- If you set `UseCompleteSearch` to `false` (the default), the algorithm stops the poll as soon as it finds one of the randomly generated points whose objective function value is less than that of the current point, and chooses that point for the next iteration.

You can modify the Latin hypercube search using two additional parameters:

```
options = optimoptions("patternsearch",SearchFcn=...
    {@searchlhs,iterlim,level})
```

- `iterlim` — Positive integer specifying the number of iterations of the pattern search for which the Latin hypercube search is performed. The default for `iterlim` is 1.
- `level` — The `level` is the number of points `patternsearch` searches, a positive integer. The default for `level` is 15 times the number of dimensions.
- "searchneldermead" specifies a search using `fminsearch`, which uses the Nelder-Mead algorithm. You can modify the Nelder-Mead search using two additional parameters:

```
options = optimoptions("patternsearch",SearchFcn=...
    {@searchneldermead,iterlim,optionsNM})
```

- `iterlim` — Positive integer specifying the number of iterations of the pattern search for which the Nelder-Mead search is performed. The default for `iterlim` is 1.
- `optionsNM` — Options for `fminsearch`, which you can create using the `optimset` function.

- "rbfsurrogate" specifies a search using a radial basis function surrogate, similar to the surrogateopt surrogate (see "Surrogate Optimization Algorithm" on page 11-3). The surrogate is formed from the most recent N+1 or more evaluation points, where N is the number of variables (size of x0). After the algorithm evaluates 10*N points, the surrogate is reset (erased) and the points for a new surrogate come from points after the reset. The radial basis function requires at least N+1 points, so after a reset, the search does not run until the algorithm evaluates at least N+1 additional points. The surrogate requires upper and lower bounds on all variables. If you do not supply a bound, the algorithm constructs one from the recent point list. Therefore, when you do not provide a bound for some variables, the algorithm performs more computations and runs a bit slower. In any case, this search function is relatively time consuming, making it best suited for use with time-consuming objective functions.
- Custom enables you to write your own search function.

```
options = optimoptions("patternsearch",SearchFcn=@myfun);
```

To see a template that you can use to write your own search function, enter

```
edit searchfcn_template
```

The following section describes the structure of the search function.

Structure of the Search Function

Your search function must have the following calling syntax.

```
function [successSearch,xBest,fBest,funcCount] = ...
    searchfcn_template(fun,x,A,b,Aeq,beq,lb,ub, ...
        optimValues,options)
```

The search function has the following input arguments:

- fun — Objective function
- x — Current point
- A,b — Linear inequality constraints
- Aeq,beq — Linear equality constraints
- lb,ub — Lower and upper bound constraints
- optimValues — Structure that enables you to set search options. The structure contains the following fields:
 - x — Current point
 - fval — Objective function value at x
 - iteration — Current iteration number
 - funcCount — Counter for user function evaluation
 - scale — Scale factor used to scale the design points
 - problemtype — Flag passed to the search routines, indicating whether the problem is 'unconstrained', 'boundconstraints', or 'linearconstraints'. This field is a subproblem type for nonlinear constrained problems.
 - meshsize — Current mesh size used in search step

- `method` — Method used in last iteration
- `options` — Pattern search options

The function has the following output arguments:

- `successSearch` — A Boolean identifier indicating whether the search is successful or not
- `xBest`, `fBest` — Best point and best function value found by search method
- `funcCount` — Number of user function evaluation in search method

See “Search and Poll” on page 6-42 for an example.

Complete Search

The `UseCompleteSearch` option applies when you set `SearchFcn` to "GPSPositiveBasis2N", "GPSPositiveBasisNp1", "GSSPositiveBasis2N", "GSSPositiveBasisNp1", "MADSPositiveBasis2N", "MADSPositiveBasisNp1", or "searchlhs". `UseCompleteSearch` can have the values `true` or `false`.

For search functions that are poll algorithms, `UseCompleteSearch` has the same meaning as the poll option `UseCompletePoll`. For the meaning of `UseCompleteSearch` for Latin hypercube search, see the "searchlhs" entry in “Search Options” on page 17-12.

Mesh Options

Mesh options control the mesh that the pattern search uses. The following options are available.

- `InitialMeshSize` specifies the size of the initial mesh, which is the length of the shortest vector from the initial point to a mesh point. `InitialMeshSize` must be a positive scalar. The default is 1.0.
- `MaxMeshSize` specifies a maximum size for the mesh. When the maximum size is reached, the mesh size does not increase after a successful iteration. `MaxMeshSize` must be a positive scalar, and is only used when a GPS or GSS algorithm is selected as the Poll or Search method. The default value is `Inf`. MADS uses a maximum size of 1.
- `AccelerateMesh` specifies whether, when the mesh size is small, the `MeshContractionFactor` is multiplied by 0.5 after each unsuccessful iteration. `AccelerateMesh` can have the values `true` (use accelerator) or `false` (do not use accelerator), the default. `AccelerateMesh` applies only to the GPS and GSS poll algorithms and to the "classic" algorithm..
- `MeshRotate` applies only when the `PollMethod` is "GPSPositiveBasisNp1" or "GSSPositiveBasisNp1". `MeshRotate = "On"` specifies that the mesh vectors are multiplied by -1 when the mesh size is less than 1/100 of the `MeshTolerance` option after an unsuccessful poll. In other words, after the first unsuccessful poll with small mesh size, instead of polling in directions e_i (unit positive directions) and $-\Sigma e_i$, the algorithm polls in directions $-e_i$ and Σe_i . `MeshRotate` can have the values "Off" or "On" (the default).
 - `MeshRotate` is especially useful for discontinuous functions.
 - When the problem has equality constraints, `MeshRotate` is disabled.
- `ScaleMesh` specifies whether the algorithm scales the mesh points by carefully multiplying the pattern vectors by constants proportional to the logarithms of the absolute values of components of the current point (or, for unconstrained problems, of the initial point). `ScaleMesh` can have the values `false` or `true` (the default). When the problem has equality constraints for the "classic" algorithm, `ScaleMesh` is disabled.

- **MeshExpansionFactor** specifies the factor by which the mesh size is increased after a successful poll. The default value is 2.0, which means that the size of the mesh is multiplied by 2.0 after a successful poll. **MeshExpansionFactor** must be a positive scalar and is only used when a GPS or GSS method is selected as the Poll or Search method and the **Algorithm** option is "classic". MADS uses a **MeshExpansionFactor** of 4.0. See "Mesh Expansion and Contraction" on page 6-66 for more information.
- **MeshContractionFactor** specifies the factor by which the mesh size is decreased after an unsuccessful poll. The default value is 0.5, which means that the size of the mesh is multiplied by 0.5 after an unsuccessful poll. **MeshContractionFactor** must be a positive scalar and is only used when a GPS or GSS method is selected as the Poll or Search method and the **Algorithm** option is "classic". MADS uses a **MeshContractionFactor** of 0.25. See "Mesh Expansion and Contraction" on page 6-66 for more information.

Constraint Parameters

For information on the meaning of penalty parameters, see "Nonlinear Constraint Solver Algorithm for Pattern Search" on page 6-46.

- **InitialPenalty** — Specifies an initial value of the penalty parameter that is used by the nonlinear constraint algorithm. **InitialPenalty** must be greater than or equal to 1, and has a default of 10.
- **PenaltyFactor** — Increases the penalty parameter when the problem is not solved to required accuracy and constraints are not satisfied. **PenaltyFactor** must be greater than 1, and has a default of 100.

TolBind specifies the tolerance for the distance from the current point to the boundary of the feasible region with respect to linear constraints. This means **TolBind** specifies when a linear constraint is active. **TolBind** is not a stopping criterion. Active linear constraints change the pattern of points **patternsearch** uses for polling or searching. The default value of **TolBind** is 1e-3.

Cache Options

The pattern search algorithm can keep a record of the points it has already polled, so that it does not have to poll the same point more than once. If the objective function requires a relatively long time to compute, the cache option can speed up the algorithm. The memory allocated for recording the points is called the cache. This option should only be used for deterministic objective functions, and not for stochastic ones.

Cache specifies whether a cache is used. The options are "On" and "Off", the default. When you set **Cache** to "On", the algorithm does not evaluate the objective function at any mesh points that are within **CacheTol** of a point in the cache.

CacheTol specifies how close a mesh point must be to a point in the cache for the algorithm to omit polling it. **CacheTol** must be a positive scalar. The default value is **eps**.

CacheSize specifies the size of the cache. **CacheSize** must be a positive scalar. The default value is 1e4.

Note Cache does not work when you run the solver in parallel.

See "Use Cache" on page 6-80 for more information.

Stopping Criteria

Stopping criteria determine what causes the pattern search algorithm to stop. Pattern search uses the following criteria:

MeshTolerance specifies the minimum tolerance for mesh size. The GPS and GSS algorithms stop if the mesh size becomes smaller than **MeshTolerance**. MADS 2N stops when the mesh size becomes smaller than MeshTolerance^2 . MADS Np1 stops when the mesh size becomes smaller than $(\text{MeshTolerance}/n\text{Var})^2$, where **nVar** is the number of elements of x_0 . The default value of **MeshTolerance** is $1e-6$.

MaxIterations specifies the maximum number of iterations the algorithm performs. The algorithm stops if the number of iterations reaches **MaxIterations**. The default value is 100 times the number of independent variables.

MaxFunctionEvaluations specifies the maximum number of evaluations of the objective function. The algorithm stops if the number of function evaluations reaches **MaxFunctionEvaluations**. The default value is 2000 times the number of independent variables.

MaxTime specifies the maximum time in seconds the pattern search algorithm runs before stopping. This also includes any specified pause time for pattern search algorithms.

StepTolerance specifies the minimum distance between the current points at two consecutive iterations. Does not apply to MADS polling. After an unsuccessful poll, the algorithm stops if the distance between two consecutive points is less than **StepTolerance** and the mesh size is smaller than **StepTolerance**. The default value is $1e-6$.

FunctionTolerance specifies the minimum tolerance for the objective function. Does not apply to MADS polling. After an unsuccessful poll, the algorithm stops if the difference between the function value at the previous best point and function value at the current best point is less than **FunctionTolerance**, and if the mesh size is also smaller than **StepTolerance**. The default value is $1e-6$.

See “Setting Solver Tolerances” on page 6-41 for an example.

ConstraintTolerance is not used as stopping criterion. It is used to determine the feasibility with respect to nonlinear constraints. The default value is $1e-6$.

Output Function Options

OutputFcn specifies functions that the pattern search algorithm calls at each iteration. For an output function file `myfun.m`, set

```
options = optimoptions("patternsearch",OutputFcn=@myfun);
```

For multiple output functions, enter a cell array of function handles:

```
options = optimoptions('patternsearch',...
    OutputFcn={@myfun1,@myfun2,...});
```

To see a template that you can use to write your own output function, enter

```
edit psoutputfcn_template
```

at the MATLAB command prompt.

To pass extra parameters in the output function, use “Anonymous Functions”.

Structure of the Output Function

Your output function must have the following calling syntax:

```
[stop,options,optchanged] = myfun(optimvalues,options,flag)
```

MATLAB passes the `optimvalues`, `options`, and `flag` data to your output function, and the output function returns `stop`, `options`, and `optchanged` data.

The output function has the following input arguments.

- `optimvalues` — Structure containing information about the current state of the solver. The structure contains the following fields:
 - `x` — Current point
 - `iteration` — Iteration number
 - `fval` — Objective function value at `x`
 - `meshsize` — Current mesh size
 - `funccount` — Number of function evaluations
 - `method` — Method used in last iteration, such as 'Update multipliers' or 'Increase penalty' for a nonlinearly constrained problem, or 'Successful Poll', 'Refine Mesh', or 'Successful Search', possibly with a '\Rotate' suffix, for a problem without nonlinear constraints
 - `TolFun` — Absolute value of change in function value in last iteration
 - `TolX` — Norm of change in `x` in last iteration
 - `nonlineq` — Nonlinear inequality constraint function values at `x`, displayed only when a nonlinear constraint function is specified
 - `nonlineq` — Nonlinear equality constraint function values at `x`, displayed only when a nonlinear constraint function is specified
- `options` — Options
- `flag` — Current state in which the output function is called. The possible values for `flag` are
 - 'init' — Initialization state
 - 'iter' — Iteration state
 - 'interrupt' — Iteration of a subproblem of a nonlinearly constrained problem
 - When `flag` is 'interrupt', the values of `optimvalues` fields apply to the subproblem iterations.
 - When `flag` is 'interrupt', `patternsearch` does not accept changes in `options`, and ignores `optchanged`.
 - 'done' — Final state

“Passing Extra Parameters” explains how to provide additional parameters to the output function.

The output function returns the following arguments to `patternsearch`:

- `stop` — Provides a way to stop the algorithm at the current iteration. `stop` can have the following values.
 - `false` — The algorithm continues to the next iteration.
 - `true` — The algorithm terminates at the current iteration.
- `options` — `patternsearch` options.
- `optchanged` — Boolean flag indicating changes to options. To change options for subsequent iterations, set `optchanged` to `true`.

Display to Command Window Options

`Display` specifies how much information is displayed at the command line while the pattern search is running. The available options are

- `"final"` (default) — The reason for stopping is displayed.
- `"off"` or the equivalent `"none"` — No output is displayed.
- `"iter"` — Information is displayed for each iteration.
- `"diagnose"` — Information is displayed for each iteration. In addition, the diagnostic lists some problem information and the options that are changed from the defaults.

Both `"iter"` and `"diagnose"` display the following information:

- `Iter` — Iteration number
- `FunEval` — Cumulative number of function evaluations
- `MeshSize` — Current mesh size
- `FunVal` — Objective function value of the current point
- `Method` — Outcome of the current poll (with no nonlinear constraint function specified). With a nonlinear constraint function, `Method` displays the update method used after a subproblem is solved.
- `Max Constraint` — Maximum nonlinear constraint violation (displayed only when a nonlinear constraint function has been specified)

Vectorized and Parallel Options

You can choose to have your objective and constraint functions evaluated in serial, parallel, or in a vectorized fashion. Set the `UseVectorized` or `UseParallel` options to `true` to use vectorized or parallel computation.

Note To use vectorized or parallel polling for the `"classic"` algorithm, you must set `UseCompletePoll` to `true`. Similarly for the `"classic"` algorithm, set `UseCompleteSearch` to `true` for vectorized or parallel searching.

Beginning in R2019a, when you set the `UseParallel` option to `true`, `patternsearch` internally overrides the `UseCompletePoll` setting to `true` so that the function polls in parallel.

- When `UseVectorized` is `false`, `patternsearch` calls the objective function on one point at a time as it loops through the mesh points. (This assumes `UseParallel` is at its default value of `false`.)

- `UseVectorized` is `true`, `patternsearch` calls the objective function on all the points in the mesh at once, i.e., in a single call to the objective function.

If there are nonlinear constraints, the objective function and the nonlinear constraints all need to be vectorized in order for the algorithm to compute in a vectorized manner.

For details and an example, see “Vectorize the Objective and Constraint Functions” on page 6-83.

- When `UseParallel` is `true`, `patternsearch` calls the objective function in parallel, using the parallel environment you established (see “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11). At the command line, set “`UseParallel`” to `false` to compute serially.

Note You cannot simultaneously use vectorized and parallel computations. If you set `UseParallel` to `true` and `UseVectorized` to `true`, `patternsearch` evaluates your objective and constraint functions in a vectorized manner, not in parallel.

How Objective and Constraint Functions Are Evaluated

Assume <code>UseCompletePoll</code> = <code>true</code>	<code>UseVectorized</code> = <code>false</code>	<code>UseVectorized</code> = <code>true</code>
<code>UseParallel</code> = <code>false</code>	Serial	Vectorized
<code>UseParallel</code> = <code>true</code>	Parallel	Vectorized

Options Table for Pattern Search Algorithms

Option Availability Table for All Algorithms

Option	Description	Algorithm Availability
<i>AccelerateMesh</i>	Accelerate mesh size contraction.	GPS and GSS, "classic" algorithm
<i>Cache</i>	With <i>Cache</i> set to "on", <i>patternsearch</i> keeps a history of the mesh points it polls and does not poll points close to them again at subsequent iterations. Use this option if <i>patternsearch</i> runs slowly because it is taking a long time to compute the objective function. If the objective function is stochastic, it is advised not to use this option. Note <i>Cache</i> does not work when you run the solver in parallel.	All
<i>CacheSize</i>	Size of the cache, in number of points.	All
<i>CacheTol</i>	Positive scalar specifying how close the current mesh point must be to a point in the cache in order for <i>patternsearch</i> to avoid polling it. Available if "Cache" option is set to "on".	All
<i>ConstraintTolerance</i>	Tolerance on nonlinear constraints.	All
<i>Display</i>	Level of display to Command Window.	All
<i>FunctionTolerance</i>	Tolerance on function value.	All
<i>InitialMeshSize</i>	Initial mesh size used in pattern search algorithms.	All
<i>InitialPenalty</i>	Initial value of the penalty parameter.	All
<i>MaxFunctionEvaluations</i>	Maximum number of objective function evaluations.	All
<i>MaxIterations</i>	Maximum number of iterations.	All
<i>MaxMeshSize</i>	Maximum mesh size used in a poll/search step.	GPS and GSS
<i>MaxTime</i>	Total time (in seconds) allowed for optimization. Also includes any specified pause time for pattern search algorithms.	All

Option	Description	Algorithm Availability
MeshContractionFactor	Mesh contraction factor, used when iteration is unsuccessful.	GPS and GSS, "classic" algorithm
MeshExpansionFactor	Mesh expansion factor, expands mesh when iteration is successful.	GPS and GSS, "classic" algorithm
<i>MeshRotate</i>	Rotate the pattern before declaring a point to be optimum.	GPS Np1 and GSS Np1
MeshTolerance	Tolerance on mesh size.	All
OutputFcn	User-specified function that a pattern search calls at each iteration.	All
<i>PenaltyFactor</i>	Penalty update parameter.	All
PlotFcn	Specifies function to plot at run time.	All
<i>PlotInterval</i>	Specifies that plot functions will be called at every interval.	All
PollOrderAlgorithm	Order in which search directions are polled.	GPS and GSS, "classic" algorithm
PollMethod	Polling strategy used in pattern search.	"classic" algorithm
ScaleMesh	Automatic scaling of variables.	All
SearchFcn	Specifies search method used in pattern search.	All
StepTolerance	Tolerance on independent variable.	All
<i>TolBind</i>	Binding tolerance used to determine if linear constraint is active.	All
UseCompletePoll	Complete poll around current iterate. Evaluate all the points in a poll step.	"classic" algorithm
UseCompleteSearch	Complete search around current iterate when the search method is a poll method. Evaluate all the points in a search step.	"classic" algorithm
UseParallel	When true, compute objective functions of a poll or search in parallel. Disable by setting to false.	All
UseVectorized	Specifies whether objective and constraint functions are vectorized.	All

Genetic Algorithm Options

In this section...

“Options for Genetic Algorithm” on page 17-23
 “Plot Options” on page 17-23
 “Population Options” on page 17-26
 “Fitness Scaling Options” on page 17-29
 “Selection Options” on page 17-30
 “Reproduction Options” on page 17-31
 “Mutation Options” on page 17-31
 “Crossover Options” on page 17-34
 “Migration Options” on page 17-37
 “Constraint Parameters” on page 17-37
 “Multiobjective Options” on page 17-38
 “Hybrid Function Options” on page 17-39
 “Stopping Criteria Options” on page 17-40
 “Output Function Options” on page 17-41
 “Display to Command Window Options” on page 17-42
 “Vectorize and Parallel Options (User Function Evaluation)” on page 17-43

Options for Genetic Algorithm

Set options for `ga` by using `optimoptions`.

```
options = optimoptions('ga','Option1','value1','Option2','value2');
```

- Some options are listed in *italics*. These options do not appear in the listing that `optimoptions` returns. To see why `optimoptions` hides these option values, see “Options that `optimoptions` Hides” on page 17-65.
- Ensure that you pass options to the solver. Otherwise, `patternsearch` uses the default option values.

```
[x,fval] = ga(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon, options)
```

Plot Options

`PlotFcn` specifies the plot function or functions called at each iteration by `ga` or `gamultiobj`. Set the `PlotFcn` option to be a built-in plot function name or a handle to the plot function. You can stop the algorithm at any time by clicking the **Stop** button on the plot window. For example, to display the best function value, set `options` as follows:

```
options = optimoptions('ga','PlotFcn','gaplotbestf');
```

To display multiple plots, use a cell array of built-in plot function names or a cell array of function handles:

```
options = optimoptions('patternsearch',...  
    'PlotFcn', {@plotfun1, @plotfun2, ...});
```

where `@plotfun1`, `@plotfun2`, and so on are function handles to the plot functions. If you specify more than one plot function, all plots appear as subplots in the same window. Right-click any subplot to obtain a larger version in a separate figure window.

Available plot functions for `ga` or for `gamultiobj`:

- `'gaplotscorediversity'` plots a histogram of the scores at each generation.
- `'gaplotstopping'` plots stopping criteria levels.
- `'gaplotgenealogy'` plots the genealogy of individuals. Lines from one generation to the next are color-coded as follows:
 - Red lines indicate mutation children.
 - Blue lines indicate crossover children.
 - Black lines indicate elite individuals.
- `'gaplotscores'` plots the scores of the individuals at each generation.
- `'gaplotdistance'` plots the average distance between individuals at each generation.
- `'gaplotselection'` plots a histogram of the parents.
- `'gaplotmaxconstr'` plots the maximum nonlinear constraint violation at each generation. For `ga`, available only when the `NonlinearConstraintAlgorithm` option is `'auglag'` (default for non-integer problems). Therefore, not available for integer-constrained problems, as they use the `'penalty'` nonlinear constraint algorithm.
- You can also create and use your own plot function. “Structure of the Plot Functions” on page 17-24 describes the structure of a custom plot function. Pass any custom function as a function handle.

The following plot functions are available for `ga` only:

- `'gaplotbestf'` plots the best score value and mean score versus generation.
- `'gaplotbestindiv'` plots the vector entries of the individual with the best fitness function value in each generation.
- `'gaplotexpectation'` plots the expected number of children versus the raw scores at each generation.
- `'gaplotrange'` plots the minimum, maximum, and mean score values in each generation.

The following plot functions are available for `gamultiobj` only:

- `'gaplotpareto'` plots the Pareto front for the first two or three objective functions.
- `'gaplotparetodistance'` plots a bar chart of the distance of each individual from its neighbors.
- `'gaplotrankhist'` plots a histogram of the ranks of the individuals. Individuals of rank 1 are on the Pareto frontier. Individuals of rank 2 are lower than at least one rank 1 individual, but are not lower than any individuals from other ranks, etc.
- `'gaplotspread'` plots the average spread as a function of iteration number.

Structure of the Plot Functions

The first line of a plot function has this form:

```
function state = plotfun(options,state,flag)
```

The input arguments to the function are

- `options` — Structure containing all the current options settings.
- `state` — Structure containing information about the current generation. “The State Structure” on page 17-25 describes the fields of `state`.
- `flag` — Description of the stage the algorithm is currently in. For details, see “Output Function Options” on page 17-41.

“Passing Extra Parameters” explains how to provide additional parameters to the function.

The output argument `state` is a state structure as well. Pass the input argument, modified if you like; see “Changing the State Structure” on page 17-42. To stop the iterations, set `state.StopFlag` to a nonempty character vector, such as `'y'`.

The State Structure

ga

The state structure for `ga`, which is an input argument to `plot`, `mutation`, and `output` functions, contains the following fields:

- `Generation` — Current generation number.
- `StartTime` — Time when genetic algorithm started, returned by `tic`.
- `StopFlag` — Reason for stopping, a character vector.
- `LastImprovement` — Generation at which the last improvement in fitness value occurred.
- `LastImprovementTime` — Time at which last improvement occurred.
- `Best` — Vector containing the best score in each generation.
- `how` — The `'augLag'` nonlinear constraint algorithm reports one of the following actions: `'Infeasible point'`, `'Update multipliers'`, or `'Increase penalty'`; see “Augmented Lagrangian Genetic Algorithm” on page 8-56.
- `FunEval` — Cumulative number of function evaluations.
- `Expectation` — Expectation for selection of individuals.
- `Selection` — Indices of individuals selected for elite, crossover, and mutation.
- `Population` — Population in the current generation.
- `Score` — Scores of the current population.
- `NonlinIneq` — Nonlinear inequality constraints at current point, present only when a nonlinear constraint function is specified, there are no integer variables, `flag` is not `'interrupt'`, and `NonlinearConstraintAlgorithm` is `'auglag'`.
- `NonlinEq` — Nonlinear equality constraints at current point, present only when a nonlinear constraint function is specified, there are no integer variables, `flag` is not `'interrupt'`, and `NonlinearConstraintAlgorithm` is `'auglag'`.
- `EvalElites` — Logical value indicating whether `ga` evaluates the fitness function of elite individuals. Initially, this value is `true`. In the first generation, if the elite individuals evaluate to their previous values (which indicates that the fitness function is deterministic), then this value becomes `false` by default for subsequent iterations. When `EvalElites` is `false`, `ga` does not

reevaluate the fitness function of elite individuals. You can override this behavior in a custom plot function or custom output function by changing the output state `EvalElites`.

- `HaveDuplicates` — Logical value indicating whether `ga` adds duplicate individuals for the initial population. `ga` uses a small relative tolerance to determine whether an individual is duplicated or unique. If `HaveDuplicates` is `true`, then `ga` locates the unique individuals and evaluates the fitness function only once for each unique individual. `ga` copies the fitness and constraint function values to duplicate individuals. `ga` repeats the test in each generation until all individuals are unique. The test takes order $n*m*\log(m)$ operations, where m is the population size and n is `nvars`. To override this test in a custom plot function or custom output function, set the output state `HaveDuplicates` to `false`.

gamultiobj

The state structure for `gamultiobj`, which is an input argument to `plot`, `mutation`, and `output` functions, contains the following fields:

- `Population` — Population in the current generation
- `Score` — Scores of the current population, a `Population-by-nObjectives` matrix, where `nObjectives` is the number of objectives
- `Generation` — Current generation number
- `StartTime` — Time when genetic algorithm started, returned by `tic`
- `StopFlag` — Reason for stopping, a character vector
- `FunEval` — Cumulative number of function evaluations
- `Selection` — Indices of individuals selected for elite, crossover, and mutation
- `Rank` — Vector of the ranks of members in the population
- `Distance` — Vector of distances of each member of the population to the nearest neighboring member
- `AverageDistance` — Standard deviation (not average) of `Distance`
- `Spread` — Vector where the entries are the spread in each generation
- `mIneq` — Number of nonlinear inequality constraints
- `mEq` — Number of nonlinear equality constraints
- `mAll` — Total number of nonlinear constraints, `mAll = mIneq + mEq`
- `C` — Nonlinear inequality constraints at current point, a `PopulationSize-by-mIneq` matrix
- `Ceq` — Nonlinear equality constraints at current point, a `PopulationSize-by-mEq` matrix
- `isFeas` — Feasibility of population, a logical vector with `PopulationSize` elements
- `maxLinInfeas` — Maximum infeasibility with respect to linear constraints for the population

Population Options

Population options let you specify the parameters of the population that the genetic algorithm uses.

`PopulationType` specifies the type of input to the fitness function. Types and their restrictions are:

- `'doubleVector'` — Use this option if the individuals in the population have type `double`. Also, the recommended data type for mixed integer programming is `'doubleVector'`, using the technique in “Mixed Integer `ga` Optimization” on page 8-40. `'doubleVector'` is the default data type.

- 'bitstring' — You can use this option if the individuals in the population have components that are 0 or 1.

Caution The individuals in a Bit string population are vectors of type double, not strings or characters.

For CreationFcn and MutationFcn, use 'gacreationuniform' and 'mutationuniform' or handles to custom functions. For CrossoverFcn, use 'crossoverscattered', 'crossoversinglepoint', 'crossovertwopoint', or a handle to a custom function.

The 'bitstring' data type can be awkward to use. ga ignores all constraints, including bounds, linear constraints, and nonlinear constraints. You cannot use a HybridFcn. To use binary variables most easily in ga, see “Mixed Integer ga Optimization” on page 8-40.

- 'custom' — Indicates a custom population type. In this case, you must also use a custom CrossoverFcn and MutationFcn. You must provide either a custom creation function or an InitialPopulationMatrix. You cannot use a HybridFcn, and ga ignores all constraints, including bounds, linear constraints, and nonlinear constraints.

PopulationSize specifies how many individuals there are in each generation. With a large population size, the genetic algorithm searches the solution space more thoroughly, thereby reducing the chance that the algorithm returns a local minimum that is not a global minimum. However, a large population size also causes the algorithm to run more slowly. The default is '50 when numberOfVariables <= 5, else 200'.

If you set PopulationSize to a vector, the genetic algorithm creates multiple subpopulations, the number of which is the length of the vector. The size of each subpopulation is the corresponding entry of the vector. Note that this option is not useful. See “Migration Options” on page 17-37.

CreationFcn specifies the function that creates the initial population for ga. Choose from:

- [] uses the default creation function for your problem type.
- 'gacreationuniform' creates a random initial population with a uniform distribution. This is the default when there are no linear constraints, or when there are integer constraints. The uniform distribution is in the initial population range (InitialPopulationRange). The default values for InitialPopulationRange are [-10;10] for every component, or [-9999;10001] when there are integer constraints. These bounds are shifted and scaled to match any existing bounds lb and ub.

Caution Do not use 'gacreationuniform' when you have linear constraints. Otherwise, your population might not satisfy the linear constraints.

- 'gacreationlinearfeasible' is the default when there are linear constraints and no integer constraints. This choice creates a random initial population that satisfies all bounds and linear constraints. If there are linear constraints, 'gacreationlinearfeasible' creates many individuals on the boundaries of the constraint region, and creates a well-dispersed population. 'gacreationlinearfeasible' ignores InitialPopulationRange. 'gacreationlinearfeasible' calls linprog to create a feasible population with respect to bounds and linear constraints.

For an example showing its behavior, see “Custom Plot Function and Linear Constraints in ga” on page 8-75.

- 'gacreationnonlinearfeasible' is the default creation function for the 'penalty' nonlinear constraint algorithm. For details, see “Constraint Parameters” on page 17-37.
- 'gacreationuniformint' is the default creation function for `ga` when the problem has integer constraints. This function applies an artificial bound to unbounded components, generates individuals uniformly at random within the bounds, and then enforces integer constraints.

Note When your problem has integer constraints, `ga` and `gamultiobj` enforce that integer constraints, bounds, and all linear constraints are feasible at each iteration. For nondefault mutation, crossover, creation, and selection functions, `ga` and `gamultiobj` apply extra feasibility routines after the functions operate.

- 'gacreationsobol' is the default creation function for `gamultiobj` when the problem has integer constraints. The creation function uses a quasirandom Sobol sequence to generate a well-dispersed initial population. The population is feasible with respect to bounds, linear constraints, and integer constraints.
- A function handle lets you write your own creation function, which must generate data of the type that you specify in `PopulationType`. For example,

```
options = optimoptions('ga','CreationFcn',@myfun);
```

Your creation function must have the following calling syntax.

```
function Population = myfun(GenomeLength, FitnessFcn, options)
```

The input arguments to the function are:

- `GenomeLength` — Number of independent variables for the fitness function
- `FitnessFcn` — Fitness function
- `options` — Options

The function returns `Population`, the initial population for the genetic algorithm.

“Passing Extra Parameters” explains how to provide additional parameters to the function.

Caution When you have bounds or linear constraints, ensure that your creation function creates individuals that satisfy these constraints. Otherwise, your population might not satisfy the constraints.

`InitialPopulationMatrix` specifies an initial population for the genetic algorithm. The default value is `[]`, in which case `ga` uses the default `CreationFcn` to create an initial population. If you enter a nonempty array in the `InitialPopulationMatrix`, the array must have no more than `PopulationSize` rows, and exactly `nvars` columns, where `nvars` is the number of variables, the second input to `ga` or `gamultiobj`. If you have a partial initial population, meaning fewer than `PopulationSize` rows, then the genetic algorithm calls `CreationFcn` to generate the remaining individuals.

`InitialScoreMatrix` specifies initial scores for the initial population. The initial scores can also be partial. If your problem has nonlinear constraints then the algorithm does not use `InitialScoreMatrix`.

`InitialPopulationRange` specifies the range of the vectors in the initial population that is generated by the `gacreationuniform` creation function. You can set `InitialPopulationRange`

to be a matrix with two rows and `nvars` columns, each column of which has the form `[lb;ub]`, where `lb` is the lower bound and `ub` is the upper bound for the entries in that coordinate. If you specify `InitialPopulationRange` to be a 2-by-1 vector, each entry is expanded to a constant row of length `nvars`. If you do not specify an `InitialPopulationRange`, the default is `[-10;10]` (`[-1e4+1;1e4+1]` for integer-constrained problems), modified to match any existing bounds. `'gacreationlinearfeasible'` ignores `InitialPopulationRange`. See “Set Initial Range” on page 8-72 for an example.

Fitness Scaling Options

Fitness scaling converts the raw fitness scores that are returned by the fitness function to values in a range that is suitable for the selection function.

`FitnessScalingFcn` specifies the function that performs the scaling. The options are

- `'fitscalingrank'` — The default fitness scaling function, `'fitscalingrank'`, scales the raw scores based on the rank of each individual instead of its score. The rank of an individual is its position in the sorted scores. An individual with rank r has scaled score proportional to $1/\sqrt{r}$. So the scaled score of the most fit individual is proportional to 1, the scaled score of the next most fit is proportional to $1/\sqrt{2}$, and so on. Rank fitness scaling removes the effect of the spread of the raw scores. The square root makes poorly ranked individuals more nearly equal in score, compared to rank scoring. For more information, see “Fitness Scaling” on page 8-80.
- `'fitscalingprop'` — Proportional scaling makes the scaled value of an individual proportional to its raw fitness score.
- `'fitscalingtop'` — Top scaling scales the top individuals equally. You can modify the top scaling using an additional parameter:

```
options = optimoptions('ga',...
    'FitnessScalingFcn',{@fitscalingtop,quantity})
```

`quantity` specifies the number of individuals that are assigned positive scaled values. `quantity` can be an integer from 1 through the population size or a fraction from 0 through 1 specifying a fraction of the population size. The default value is 0.4. Each of the individuals that produce offspring is assigned an equal scaled value, while the rest are assigned the value 0. The scaled values have the form `[0 1/n 1/n 0 0 1/n 0 0 1/n ...]`.

- `'fitscalingshiftlinear'` — Shift linear scaling scales the raw scores so that the expectation of the fittest individual is equal to a constant called `rate` multiplied by the average score. You can modify the `rate` parameter:

```
options = optimoptions('ga','FitnessScalingFcn',...
    {@fitscalingshiftlinear, rate})
```

The default value of `rate` is 2.

- A function handle lets you write your own scaling function.

```
options = optimoptions('ga','FitnessScalingFcn',@myfun);
```

Your scaling function must have the following calling syntax:

```
function expectation = myfun(scores, nParents)
```

The input arguments to the function are:

- `scores` — A vector of scalars, one for each member of the population
- `nParents` — The number of parents needed from this population

The function returns `expectation`, a column vector of scalars of the same length as `scores`, giving the scaled values of each member of the population. The sum of the entries of `expectation` must equal `nParents`.

“Passing Extra Parameters” explains how to provide additional parameters to the function.

See “Fitness Scaling” on page 8-80 for more information.

Selection Options

Selection options specify how the genetic algorithm chooses parents for the next generation.

The `SelectionFcn` option specifies the selection function.

`gamultiobj` uses only the 'selectiontournament' selection function.

For `ga` the options are:

- 'selectionstochunif' — The `ga` default selection function, 'selectionstochunif', lays out a line in which each parent corresponds to a section of the line of length proportional to its scaled value. The algorithm moves along the line in steps of equal size. At each step, the algorithm allocates a parent from the section it lands on. The first step is a uniform random number less than the step size.
- 'selectionremainder' — Remainder selection assigns parents deterministically from the integer part of each individual's scaled value and then uses roulette selection on the remaining fractional part. For example, if the scaled value of an individual is 2.3, that individual is listed twice as a parent because the integer part is 2. After parents have been assigned according to the integer parts of the scaled values, the rest of the parents are chosen stochastically. The probability that a parent is chosen in this step is proportional to the fractional part of its scaled value.
- 'selectionuniform' — Uniform selection chooses parents using the expectations and number of parents. Uniform selection is useful for debugging and testing, but is not a very effective search strategy.
- 'selectionroulette' — Roulette selection chooses parents by simulating a roulette wheel, in which the area of the section of the wheel corresponding to an individual is proportional to the individual's expectation. The algorithm uses a random number to select one of the sections with a probability equal to its area.
- 'selectiontournament' — Tournament selection chooses each parent by choosing `size` players at random and then choosing the best individual out of that set to be a parent. `size` must be at least 2. The default value of `size` is 4. Set `size` to a different value as follows:

```
options = optimoptions('ga','SelectionFcn',...  
                      {@selectiontournament,size})
```

When `NonlinearConstraintAlgorithm` is `Penalty`, `ga` uses 'selectiontournament' with `size` 2.

-
- **Note** When your problem has integer constraints, `ga` and `gamultiobj` enforce that integer constraints, bounds, and all linear constraints are feasible at each iteration. For nondefault mutation, crossover, creation, and selection functions, `ga` and `gamultiobj` apply extra feasibility routines after the functions operate.
-

- A function handle enables you to write your own selection function.

```
options = optimoptions('ga','SelectionFcn',@myfun);
```

Your selection function must have the following calling syntax:

```
function parents = myfun(expectation, nParents, options)
```

`ga` provides the input arguments `expectation`, `nParents`, and `options`. Your function returns the indices of the parents.

The input arguments to the function are:

- `expectation`
 - For `ga`, `expectation` is a column vector of the scaled fitness of each member of the population. The scaling comes from the “Fitness Scaling Options” on page 17-29.

Tip You can ensure that you have a column vector by using `expectation(:,1)`. For example, edit `selectionstochunif` or any of the other built-in selection functions.

- For `gamultiobj`, `expectation` is a matrix whose first column is the negative of the rank of the individuals, and whose second column is the distance measure of the individuals. See “Multiobjective Options” on page 17-38.
- `nParents`— Number of parents to select.
- `options` — Genetic algorithm options.

The function returns `parents`, a row vector of length `nParents` containing the indices of the parents that you select.

“Passing Extra Parameters” explains how to provide additional parameters to the function.

See “Selection” on page 8-19 for more information.

Reproduction Options

Reproduction options specify how the genetic algorithm creates children for the next generation.

`EliteCount` specifies the number of individuals that are guaranteed to survive to the next generation. Set `EliteCount` to be a positive integer less than or equal to the population size. The default value is `ceil(0.05*PopulationSize)` for continuous problems, and `0.05*(default PopulationSize)` for mixed-integer problems.

`CrossoverFraction` specifies the fraction of the next generation, other than elite children, that are produced by crossover. Set `CrossoverFraction` to be a fraction between 0 and 1. The default value is 0.8.

See “Setting the Crossover Fraction” in “Vary Mutation and Crossover” on page 8-83 for an example.

Mutation Options

Mutation options specify how the genetic algorithm makes small random changes in the individuals in the population to create mutation children. Mutation provides genetic diversity and enables the

genetic algorithm to search a broader space. Specify the mutation function in the `MutationFcn` option.

`MutationFcn` options:

- `'mutationgaussian'` — The default mutation function for `ga` for unconstrained problems, `'mutationgaussian'`, adds a random number taken from a Gaussian distribution with mean 0 to each entry of the parent vector. The standard deviation of this distribution is determined by the parameters `scale` and `shrink`, and by the `InitialPopulationRange` option. Set `scale` and `shrink` as follows:

```
options = optimoptions('ga','MutationFcn', ...
{@mutationgaussian, scale, shrink})
```

- The `scale` parameter determines the standard deviation at the first generation. If you set `InitialPopulationRange` to be a 2-by-1 vector v , the initial standard deviation is the same at all coordinates of the parent vector, and is given by $scale*(v(2) - v(1))$.

If you set `InitialPopulationRange` to be a vector v with two rows and `nvars` columns, the initial standard deviation at coordinate i of the parent vector is given by $scale*(v(i,2) - v(i,1))$.

- The `shrink` parameter controls how the standard deviation shrinks as generations go by. If you set `InitialPopulationRange` to be a 2-by-1 vector, the standard deviation at the k th generation, σ_k , is the same at all coordinates of the parent vector, and is given by the recursive formula

$$\sigma_k = \sigma_{k-1} \left(1 - \text{Shrink} \frac{k}{\text{Generations}} \right).$$

If you set `InitialPopulationRange` to be a vector with two rows and `nvars` columns, the standard deviation at coordinate i of the parent vector at the k th generation, $\sigma_{i,k}$, is given by the recursive formula

$$\sigma_{i,k} = \sigma_{i,k-1} \left(1 - \text{Shrink} \frac{k}{\text{Generations}} \right).$$

If you set `shrink` to 1, the algorithm shrinks the standard deviation in each coordinate linearly until it reaches 0 at the last generation is reached. A negative value of `shrink` causes the standard deviation to grow.

The default value of both `scale` and `shrink` is 1.

Caution Do not use `mutationgaussian` when you have bounds or linear constraints. Otherwise, your population will not necessarily satisfy the constraints. Instead, use `'mutationadaptfeasible'` or a custom mutation function that satisfies linear constraints.

- `'mutationuniform'` — Uniform mutation is a two-step process. First, the algorithm selects a fraction of the vector entries of an individual for mutation, where each entry has a probability rate of being mutated. The default value of `rate` is 0.01. In the second step, the algorithm replaces each selected entry by a random number selected uniformly from the range for that entry.

To change the default value of `rate`,

```
options = optimoptions('ga','MutationFcn', {@mutationuniform, rate})
```

Caution Do not use `mutationuniform` when you have bounds or linear constraints. Otherwise, your population will not necessarily satisfy the constraints. Instead, use `'mutationadaptfeasible'` or a custom mutation function that satisfies linear constraints.

- `'mutationadaptfeasible'`, the default mutation function for `gamultiobj` and for `ga` when there are noninteger constraints, randomly generates directions that are adaptive with respect to the last successful or unsuccessful generation. The mutation chooses a direction and step length that satisfies bounds and linear constraints.
- `'mutationpower'` is the default mutation function for `ga` and `gamultiobj` when the problem has integer constraints. Power mutation mutates a parent, x , via the following. For each component of the parent, the i th component of the child is given by:

$$\begin{aligned} \text{mutationChild}(i) &= x(i) - s(x(i) - \text{lb}(i)) \text{ if } t < r \\ &= x(i) + s(\text{ub}(i) - x(i)) \text{ if } t \geq r. \end{aligned}$$

Here, t is the scaled distance of $x(i)$ from the i th component of the lower bound, $\text{lb}(i)$. s is a random variable drawn from a power distribution and r is a random number drawn from a uniform distribution.

This function can handle $\text{lb}(i) = \text{ub}(i)$. New children are generated with the i th component set to $\text{lb}(i)$, which equals $\text{ub}(i)$. For more information on this crossover function see section 2.1 of the following reference:

Kusum Deep, Krishna Pratap Singh, M. L. Kansal, C. Mohan. *A real coded genetic algorithm for solving integer and mixed integer optimization problems*. Applied Mathematics and Computation, 212 (2009), 505-518.

Note When your problem has integer constraints, `ga` and `gamultiobj` enforce that integer constraints, bounds, and all linear constraints are feasible at each iteration. For nondefault mutation, crossover, creation, and selection functions, `ga` and `gamultiobj` apply extra feasibility routines after the functions operate.

- `'mutationpositivebasis'` — This mutation function is similar to orthogonal MADS steps, modified for linear constraints and bounds.
- A function handle enables you to write your own mutation function.

```
options = optimoptions('ga','MutationFcn',@myfun);
```

Your mutation function must have this calling syntax:

```
function mutationChildren = myfun(parents, options, nvars,
FitnessFcn, state, thisScore, thisPopulation)
```

The arguments to the function are

- `parents` — Row vector of parents chosen by the selection function
- `options` — Options
- `nvars` — Number of variables
- `FitnessFcn` — Fitness function
- `state` — Structure containing information about the current generation. “The State Structure” on page 17-25 describes the fields of `state`.

- `thisScore` — Vector of scores of the current population
- `thisPopulation` — Matrix of individuals in the current population

The function returns `mutationChildren`—the mutated offspring—as a matrix where rows correspond to the children. The number of columns of the matrix is `nvars`.

“Passing Extra Parameters” explains how to provide additional parameters to the function.

Caution When you have bounds or linear constraints, ensure that your mutation function creates individuals that satisfy these constraints. Otherwise, your population will not necessarily satisfy the constraints.

Crossover Options

Crossover options specify how the genetic algorithm combines two individuals, or parents, to form a crossover child for the next generation.

`CrossoverFcn` specifies the function that performs the crossover. You can choose from the following functions:

- `'crossoverscattered'`, the default crossover function for problems without linear constraints, creates a random binary vector and selects the genes where the vector is a 1 from the first parent, and the genes where the vector is a 0 from the second parent, and combines the genes to form the child. For example, if `p1` and `p2` are the parents

```
p1 = [a b c d e f g h]
p2 = [1 2 3 4 5 6 7 8]
```

and the binary vector is `[1 1 0 0 1 0 0 0]`, the function returns the following child:

```
child1 = [a b 3 4 e 6 7 8]
```

Caution When your problem has linear constraints, `'crossoverscattered'` can give a poorly distributed population. In this case, use a different crossover function, such as `'crossoverintermediate'`.

- `'crossoversinglepoint'` chooses a random integer `n` between 1 and `nvars` and then
 - Selects vector entries numbered less than or equal to `n` from the first parent.
 - Selects vector entries numbered greater than `n` from the second parent.
 - Concatenates these entries to form a child vector.

For example, if `p1` and `p2` are the parents

```
p1 = [a b c d e f g h]
p2 = [1 2 3 4 5 6 7 8]
```

and the crossover point is 3, the function returns the following child.

```
child = [a b c 4 5 6 7 8]
```

Caution When your problem has linear constraints, 'crossoversinglepoint' can give a poorly distributed population. In this case, use a different crossover function, such as 'crossoverintermediate'.

- 'crossovertwopoint' selects two random integers m and n between 1 and $nvars$. The function selects
 - Vector entries numbered less than or equal to m from the first parent
 - Vector entries numbered from $m+1$ to n , inclusive, from the second parent
 - Vector entries numbered greater than n from the first parent.

The algorithm then concatenates these genes to form a single gene. For example, if $p1$ and $p2$ are the parents

```
p1 = [a b c d e f g h]
p2 = [1 2 3 4 5 6 7 8]
```

and the crossover points are 3 and 6, the function returns the following child.

```
child = [a b c 4 5 6 g h]
```

Caution When your problem has linear constraints, 'crossovertwopoint' can give a poorly distributed population. In this case, use a different crossover function, such as 'crossoverintermediate'.

- 'crossoverintermediate', the default crossover function when there are linear constraints, creates children by taking a weighted average of the parents. You can specify the weights by a single parameter, `ratio`, which can be a scalar or a row vector of length $nvars$. The default value of `ratio` is a vector of all 1's. Set the `ratio` parameter as follows.

```
options = optimoptions('ga', 'CrossoverFcn', ...
{@crossoverintermediate, ratio});
```

'crossoverintermediate' creates the child from `parent1` and `parent2` using the following formula.

$$\text{child} = \text{parent1} + \text{rand} * \mathbf{Ratio} * (\text{parent2} - \text{parent1})$$

If all the entries of `ratio` lie in the range $[0, 1]$, the children produced are within the hypercube defined by placing the parents at opposite vertices. If `ratio` is not in that range, the children might lie outside the hypercube. If `ratio` is a scalar, then all the children lie on the line between the parents.

- 'crossoverlaplace' is the default crossover function when the problem has integer constraints. The Laplace crossover generates children using either of the following formulae (chosen at random):

$$x\text{OverKid} = p1 + b1 * \text{abs}(p1 - p2)$$

$$x\text{OverKid} = p2 + b1 * \text{abs}(p1 - p2)$$

Here, $p1$, $p2$ are the parents of `xOverKid` and $b1$ is a random number generated from a Laplace distribution. For more information on this crossover function see section 2.1 of the following reference:

Kusum Deep, Krishna Pratap Singsh, M. L. Kansal, C. Mohan. *A real coded genetic algorithm for solving integer and mixed integer optimization problems*. Applied Mathematics and Computation, 212 (2009), 505-518.

- 'crossoverheuristic' returns a child that lies on the line containing the two parents, a small distance away from the parent with the better fitness value in the direction away from the parent with the worse fitness value. You can specify how far the child is from the better parent by the parameter `ratio`. The default value of `ratio` is 1.2. Set the `ratio` parameter as follows.

```
options = optimoptions('ga','CrossoverFcn',...
    {@crossoverheuristic,ratio});
```

If `parent1` and `parent2` are the parents, and `parent1` has the better fitness value, the function returns the child

```
child = parent2 + ratio * (parent1 - parent2);
```

Caution When your problem has linear constraints, 'crossoverheuristic' can give a poorly distributed population. In this case, use a different crossover function, such as 'crossoverintermediate'.

- 'crossoverarithmetic' creates children that are the weighted arithmetic mean of two parents. Children are always feasible with respect to linear constraints and bounds.
- **Note** When your problem has integer constraints, `ga` and `gamultiobj` enforce that integer constraints, bounds, and all linear constraints are feasible at each iteration. For nondefault mutation, crossover, creation, and selection functions, `ga` and `gamultiobj` apply extra feasibility routines after the functions operate.
- A function handle enables you to write your own crossover function.

```
options = optimoptions('ga','CrossoverFcn',@myfun);
```

Your crossover function must have the following calling syntax.

```
xoverKids = myfun(parents, options, nvars, FitnessFcn, ...
    unused,thisPopulation)
```

The arguments to the function are

- `parents` — Row vector of parents chosen by the selection function
- `options` — options
- `nvars` — Number of variables
- `FitnessFcn` — Fitness function
- `unused` — Placeholder not used
- `thisPopulation` — Matrix representing the current population. The number of rows of the matrix is `PopulationSize` and the number of columns is `nvars`.

The function returns `xoverKids`—the crossover offspring—as a matrix where rows correspond to the children. The number of columns of the matrix is `nvars`.

“Passing Extra Parameters” explains how to provide additional parameters to the function.

Caution When you have bounds or linear constraints, ensure that your crossover function creates individuals that satisfy these constraints. Otherwise, your population will not necessarily satisfy the constraints.

Migration Options

Note Subpopulations refer to a form of parallel processing for the genetic algorithm. `ga` currently does not support this form. In subpopulations, each worker hosts a number of individuals. These individuals are a subpopulation. The worker evolves the subpopulation independently of other workers, except when migration causes some individuals to travel between workers.

Because `ga` does not currently support this form of parallel processing, there is no benefit to setting `PopulationSize` to a vector, or to setting the `MigrationDirection`, `MigrationInterval`, or `MigrationFraction` options.

Migration options specify how individuals move between subpopulations. Migration occurs if you set `PopulationSize` to be a vector of length greater than 1. When migration occurs, the best individuals from one subpopulation replace the worst individuals in another subpopulation. Individuals that migrate from one subpopulation to another are copied. They are not removed from the source subpopulation.

You can control how migration occurs by the following three options:

- `MigrationDirection` — Migration can take place in one or both directions.
 - If you set `MigrationDirection` to 'forward', migration takes place toward the last subpopulation. That is, the n th subpopulation migrates into the $(n+1)$ th subpopulation.
 - If you set `MigrationDirection` to 'both', the n th subpopulation migrates into both the $(n-1)$ th and the $(n+1)$ th subpopulation.

Migration wraps at the ends of the subpopulations. That is, the last subpopulation migrates into the first, and the first may migrate into the last.

- `MigrationInterval` — Specifies how many generation pass between migrations. For example, if you set `MigrationInterval` to 20, migration takes place every 20 generations.
- `MigrationFraction` — Specifies how many individuals move between subpopulations. `MigrationFraction` specifies the fraction of the smaller of the two subpopulations that moves. For example, if individuals migrate from a subpopulation of 50 individuals into a subpopulation of 100 individuals and you set `MigrationFraction` to 0.1, the number of individuals that migrate is $0.1 * 50 = 5$.

Constraint Parameters

Constraint parameters refer to the nonlinear constraint solver. For details on the algorithm, see “Nonlinear Constraint Solver Algorithms for Genetic Algorithm” on page 8-56.

Choose between the nonlinear constraint algorithms by setting the `NonlinearConstraintAlgorithm` option to 'auglag' (Augmented Lagrangian) or 'penalty' (Penalty algorithm).

- “Augmented Lagrangian Genetic Algorithm” on page 17-38

- “Penalty Algorithm” on page 17-38

Augmented Lagrangian Genetic Algorithm

- *InitialPenalty* — Specifies an initial value of the penalty parameter that is used by the nonlinear constraint algorithm. *InitialPenalty* must be greater than or equal to 1, and has a default of 10.
- *PenaltyFactor* — Increases the penalty parameter when the problem is not solved to required accuracy and constraints are not satisfied. *PenaltyFactor* must be greater than 1, and has a default of 100.

Penalty Algorithm

The penalty algorithm uses the 'gacreationnonlinearfeasible' creation function by default. This creation function uses `fmincon` to find feasible individuals.

'gacreationnonlinearfeasible' starts `fmincon` from a variety of initial points within the bounds from the `InitialPopulationRange` option. Optionally, 'gacreationnonlinearfeasible' can run `fmincon` in parallel on the initial points.

You can specify tuning parameters for 'gacreationnonlinearfeasible' using the following name-value pairs.

Name	Value
<code>SolverOpts</code>	<code>fmincon</code> options, created using <code>optimoptions</code> or <code>optimset</code> .
<code>UseParallel</code>	When <code>true</code> , run <code>fmincon</code> in parallel on initial points; default is <code>false</code> .
<code>NumStartPts</code>	Number of start points, a positive integer up to <code>sum(PopulationSize)</code> in value.

Include the name-value pairs in a cell array along with `@gacreationnonlinearfeasible`.

```
options = optimoptions('ga','CreationFcn',{@gacreationnonlinearfeasible,...
    'UseParallel',true,'NumStartPts',20});
```

Multiobjective Options

Multiobjective options define parameters characteristic of the `gamultiobj` algorithm. You can specify the following parameters:

- *ParetoFraction* — Sets the fraction of individuals to keep on the first Pareto front while the solver selects individuals from higher fronts. This option is a scalar between 0 and 1.

Note The fraction of individuals on the first Pareto front can exceed `ParetoFraction`. This occurs when there are too few individuals of other ranks in step 6 of “Iterations” on page 14-7.

- *DistanceMeasureFcn* — Defines a handle to the function that computes distance measure of individuals, computed in decision variable space (genotype, also termed design variable space) or in function space (phenotype). For example, the default distance measure function is 'distancecrowding' in function space, which is the same as `{@distancecrowding,'phenotype'}`.

“Distance” measures a crowding of each individual in a population. Choose between the following:

- 'distancecrowding', or the equivalent `{@distancecrowding,'phenotype'}` — Measure the distance in fitness function space.

- `@distancecrowding, 'genotype'` — Measure the distance in decision variable space.
- `@distancefunction` — Write a custom distance function using the following template.

```
function distance = distancefunction(pop,score,options)
% Uncomment one of the following two lines, or use a combination of both
% y = score; % phenotype
% y = pop; % genotype
popSize = size(y,1); % number of individuals
numData = size(y,2); % number of dimensions or fitness functions
distance = zeros(popSize,1); % allocate the output
% Compute distance here
```

`gamultiobj` passes the population in `pop`, the computed scores for the population in `scores`, and the options in `options`. Your distance function returns the distance from each member of the population to a reference, such as the nearest neighbor in some sense. For an example, edit the built-in file `distancecrowding.m`.

Hybrid Function Options

- “ga Hybrid Function” on page 17-39
- “gamultiobj Hybrid Function” on page 17-40

ga Hybrid Function

A hybrid function is another minimization function that runs after the genetic algorithm terminates. You can specify a hybrid function in the `HybridFcn` option. Do not use with integer problems. The choices are

- `[]` — No hybrid function.
- `'fminsearch'` — Uses the MATLAB function `fminsearch` to perform unconstrained minimization.
- `'patternsearch'` — Uses a pattern search to perform constrained or unconstrained minimization.
- `'fminunc'` — Uses the Optimization Toolbox function `fminunc` to perform unconstrained minimization.
- `'fmincon'` — Uses the Optimization Toolbox function `fmincon` to perform constrained minimization.

Note Ensure that your hybrid function accepts your problem constraints. Otherwise, `ga` throws an error.

You can set separate options for the hybrid function. Use `optimset` for `fminsearch`, or `optimoptions` for `fmincon`, `patternsearch`, or `fminunc`. For example:

```
hybridopts = optimoptions('fminunc','Display','iter',...
    'Algorithm','quasi-newton');
```

Include the hybrid options in the Genetic Algorithm options as follows:

```
options = optimoptions('ga',options,'HybridFcn',{fminunc,hybridopts});
```

`hybridopts` must exist before you set `options`.

See “Hybrid Scheme in the Genetic Algorithm” on page 8-95 for an example. See “When to Use a Hybrid Function” on page 8-116.

gamultiobj Hybrid Function

A hybrid function is another minimization function that runs after the multiobjective genetic algorithm terminates. You can specify the hybrid function 'fggoalattain' in the HybridFcn option.

In use as a multiobjective hybrid function, the solver does the following:

- 1 Compute the maximum and minimum of each objective function at the solutions. For objective j at solution k , let

$$F_{\max}(j) = \max_k F_k(j)$$

$$F_{\min}(j) = \min_k F_k(j).$$

- 2 Compute the total weight at each solution k ,

$$w(k) = \sum_j \frac{F_{\max}(j) - F_k(j)}{1 + F_{\max}(j) - F_{\min}(j)}.$$

- 3 Compute the weight for each objective function j at each solution k ,

$$p(j, k) = w(k) \frac{F_{\max}(j) - F_k(j)}{1 + F_{\max}(j) - F_{\min}(j)}.$$

- 4 For each solution k , perform the goal attainment problem with goal vector $F_k(j)$ and weight vector $p(j, k)$.

For more information, see section 9.6 of Deb [3].

Stopping Criteria Options

Stopping criteria determine what causes the algorithm to terminate. You can specify the following options:

- **MaxGenerations** — Specifies the maximum number of iterations for the genetic algorithm to perform. The default is `100*numberOfVariables`.
- **MaxTime** — Specifies the maximum time in seconds the genetic algorithm runs before stopping, as measured by `tic` and `toc`. This limit is enforced after each iteration, so `ga` can exceed the limit when an iteration takes substantial time.
- **FitnessLimit** — The algorithm stops if the best fitness value is less than or equal to the value of **FitnessLimit**. Does not apply to `gamultiobj`.
- **MaxStallGenerations** — The algorithm stops if the average relative change in the best fitness function value over **MaxStallGenerations** is less than or equal to **FunctionTolerance**. (If the **StallTest** option is 'geometricWeighted', then the test is for a *geometric weighted* average relative change.) For a problem with nonlinear constraints, **MaxStallGenerations** applies to the subproblem (see “Nonlinear Constraint Solver Algorithms for Genetic Algorithm” on page 8-56).

For `gamultiobj`, if the geometric average of the relative change in the spread of the Pareto solutions over **MaxStallGenerations** is less than **FunctionTolerance**, and the final spread is smaller than the average spread over the last **MaxStallGenerations**, then the algorithm stops.

The geometric average coefficient is $\frac{1}{2}$. The spread is a measure of the movement of the Pareto front. See “gamultiobj Algorithm” on page 14-5.

- **MaxStallTime** — The algorithm stops if there is no improvement in the best fitness value for an interval of time in seconds specified by **MaxStallTime**, as measured by **tic** and **toc**.
- **FunctionTolerance** — The algorithm stops if the average relative change in the best fitness function value over **MaxStallGenerations** is less than or equal to **FunctionTolerance**. (If the **StallTest** option is 'geometricWeighted', then the test is for a *geometric weighted* average relative change.)

For **gamultiobj**, if the geometric average of the relative change in the spread of the Pareto solutions over **MaxStallGenerations** is less than **FunctionTolerance**, and the final spread is smaller than the average spread over the last **MaxStallGenerations**, then the algorithm stops. The geometric average coefficient is $\frac{1}{2}$. The spread is a measure of the movement of the Pareto front. See “gamultiobj Algorithm” on page 14-5.

- **ConstraintTolerance** — The **ConstraintTolerance** is not used as stopping criterion. It is used to determine the feasibility with respect to nonlinear constraints. Also, $\max(\text{sqrt}(\text{eps}), \text{ConstraintTolerance})$ determines feasibility with respect to linear constraints.

See “Set Maximum Number of Generations and Stall Generations” on page 8-101 for an example.

Output Function Options

Output functions are functions that the genetic algorithm calls at each generation. Unlike other solvers, a **ga** output function can not only read the values of the state of the algorithm, but also modify those values. An output function can also halt the solver according to conditions you set.

```
options = optimoptions('ga', 'OutputFcn', @myfun);
```

For multiple output functions, enter a cell array of function handles:

```
options = optimoptions('ga', 'OutputFcn', {@myfun1, @myfun2, ...});
```

To see a template that you can use to write your own output functions, enter

```
edit gaoutputfcn_template
```

at the MATLAB command line.

For an example, see “Custom Output Function for Genetic Algorithm” on page 8-105.

Structure of the Output Function

Your output function must have the following calling syntax:

```
[state, options, optchanged] = myfun(options, state, flag)
```

MATLAB passes the **options**, **state**, and **flag** data to your output function, and the output function returns **state**, **options**, and **optchanged** data.

Note To stop the iterations, set **state.StopFlag** to a nonempty character vector, such as 'y'.

The output function has the following input arguments:

- `options` — Options
- `state` — Structure containing information about the current generation. “The State Structure” on page 17-25 describes the fields of `state`.
- `flag` — Current status of the algorithm:
 - `'init'` — Initialization state
 - `'iter'` — Iteration state
 - `'interrupt'` — Iteration of a subproblem of a nonlinearly constrained problem for the `'auglag'` nonlinear constraint algorithm. When `flag` is `'interrupt'`:
 - The values of `state` fields apply to the subproblem iterations.
 - `ga` does not accept changes in `options`, and ignores `optchanged`.
 - The `state.NonlinIneq` and `state.NonlinEq` fields are not available.
 - `'done'` — Final state

“Passing Extra Parameters” explains how to provide additional parameters to the function.

The output function returns the following arguments to `ga`:

- `state` — Structure containing information about the current generation. “The State Structure” on page 17-25 describes the fields of `state`. To stop the iterations, set `state.StopFlag` to a nonempty character vector, such as `'y'`.
- `options` — Options as modified by the output function. This argument is optional.
- `optchanged` — Boolean flag indicating changes to `options`. To change `options` for subsequent iterations, set `optchanged` to `true`.

Changing the State Structure

Caution Changing the state structure carelessly can lead to inconsistent or erroneous results. Usually, you can achieve the same or better state modifications by using mutation or crossover functions, instead of changing the state structure in a plot function or output function.

`ga` output functions can change the `state` structure (see “The State Structure” on page 17-25). Be careful when changing values in this structure, as you can pass inconsistent data back to `ga`.

Tip If your output structure changes the `Population` field, then be sure to update the `Score` field, and possibly the `Best`, `NonlinIneq`, or `NonlinEq` fields, so that they contain consistent information.

To update the `Score` field after changing the `Population` field, first calculate the fitness function values of the population, then calculate the fitness scaling for the population. See “Fitness Scaling Options” on page 17-29.

Display to Command Window Options

`'Display'` specifies how much information is displayed at the command line while the genetic algorithm is running. The available options are

- 'final' (default) — The reason for stopping is displayed.
- 'off' or the equivalent 'none' — No output is displayed.
- 'iter' — Information is displayed at each iteration.
- 'diagnose' — Information is displayed at each iteration. In addition, the diagnostic lists some problem information and the options that have been changed from the defaults.

Both 'iter' and 'diagnose' display the following information:

- Generation — Generation number
- f-count — Cumulative number of fitness function evaluations
- Best $f(x)$ — Best fitness function value
- Mean $f(x)$ — Mean fitness function value
- Stall generations — Number of generations since the last improvement of the fitness function

When a nonlinear constraint function has been specified, 'iter' and 'diagnose' do not display the Mean $f(x)$, but additionally display:

- Max Constraint — Maximum nonlinear constraint violation

In addition, 'iter' and 'diagnose' display problem information before the iterative display, such as problem type and which creation, mutation, crossover, and selection functions `ga` or `gamultiobj` is using.

Vectorize and Parallel Options (User Function Evaluation)

You can choose to have your fitness and constraint functions evaluated in serial, parallel, or in a vectorized fashion. Set the 'UseVectorized' and 'UseParallel' options with `optimoptions`.

- When 'UseVectorized' is false (default), `ga` calls the fitness function on one individual at a time as it loops through the population. (This assumes 'UseParallel' is at its default value of false.)
- When 'UseVectorized' is true, `ga` calls the fitness function on the entire population at once, in a single call to the fitness function.

If there are nonlinear constraints, the fitness function and the nonlinear constraints all need to be vectorized in order for the algorithm to compute in a vectorized manner.

See “Vectorize the Fitness Function” on page 8-103 for an example.

- When `UseParallel` is true, `ga` calls the fitness function in parallel, using the parallel environment you established (see “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11). Set `UseParallel` to false (default) to compute serially.

Note You cannot simultaneously use vectorized and parallel computations. If you set 'UseParallel' to true and 'UseVectorized' to true, `ga` evaluates your fitness and constraint functions in a vectorized manner, not in parallel.

How Fitness and Constraint Functions Are Evaluated

	UseVectorized = false	UseVectorized = true
UseParallel = false	Serial	Vectorized
UseParallel = true	Parallel	Vectorized

Particle Swarm Options

In this section...

“Specifying Options for particleswarm” on page 17-45
 “Swarm Creation” on page 17-45
 “Display Settings” on page 17-46
 “Algorithm Settings” on page 17-46
 “Hybrid Function” on page 17-47
 “Output Function and Plot Function” on page 17-48
 “Parallel or Vectorized Function Evaluation” on page 17-49
 “Stopping Criteria” on page 17-50

Specifying Options for particleswarm

Create options using the `optimoptions` function as follows.

```
options = optimoptions('particleswarm',...
    'Param1',value1,'Param2',value2,...);
```

For an example, see “Optimize Using Particle Swarm” on page 10-5.

Each option in this section is listed by its field name in `options`. For example, `Display` refers to the corresponding field of `options`.

Swarm Creation

By default, `particleswarm` calls the `'pswcreationuniform'` swarm creation function. This function works as follows.

- 1 If an `InitialSwarmMatrix` option exists, `'pswcreationuniform'` takes the first `SwarmSize` rows of the `InitialSwarmMatrix` matrix as the swarm. If the number of rows of the `InitialSwarmMatrix` matrix is smaller than `SwarmSize`, then `'pswcreationuniform'` continues to the next step.
- 2 `'pswcreationuniform'` creates enough particles so that there are `SwarmSize` in total. `'pswcreationuniform'` creates particles that are randomly, uniformly distributed. The range for any swarm component is $-\text{InitialSwarmSpan}/2, \text{InitialSwarmSpan}/2$, shifted and scaled if necessary to match any bounds.

After creation, `particleswarm` checks that all particles satisfy any bounds, and truncates components if necessary. If the `Display` option is `'iter'` and a particle needed truncation, then `particleswarm` notifies you.

Custom Creation Function

Set a custom creation function using `optimoptions` to set the `CreationFcn` option to `@customcreation`, where `customcreation` is the name of your creation function file. A custom creation function has this syntax.

```
swarm = customcreation(problem)
```

The creation function should return a matrix of size `SwarmSize-by-nvars`, where each row represents the location of one particle. See `problem` for details of the problem structure. In particular, you can obtain `SwarmSize` from `problem.options.SwarmSize`, and `nvars` from `problem.nvars`.

For an example of a creation function, see the code for `pswcreationuniform`.

```
edit pswcreationuniform
```

Display Settings

The `Display` option specifies how much information is displayed at the command line while the algorithm is running.

- `'off'` or `'none'` — No output is displayed.
- `'iter'` — Information is displayed at each iteration.
- `'final'` (default) — The reason for stopping is displayed.

`iter` displays:

- `Iteration` — Iteration number
- `f-count` — Cumulative number of objective function evaluations
- `Best f(x)` — Best objective function value
- `Mean f(x)` — Mean objective function value over all particles
- `Stall Iterations` — Number of iterations since the last change in `Best f(x)`

The `DisplayInterval` option sets the number of iterations that are performed before the iterative display updates. Give a positive integer.

Algorithm Settings

The details of the `particleswarm` algorithm appear in “Particle Swarm Optimization Algorithm” on page 10-11. This section describes the tuning parameters.

The main step in the particle swarm algorithm is the generation of new velocities for the swarm:

For `u1` and `u2` uniformly (0,1) distributed random vectors of length `nvars`, update the velocity

$$v = W*v + y1*u1.*(p-x) + y2*u2.*(g-x).$$

The variables `W = inertia`, `y1 = SelfAdjustmentWeight`, and `y2 = SocialAdjustmentWeight`.

This update uses a weighted sum of:

- The previous velocity `v`
- `x-p`, the difference between the current position `x` and the best position `p` the particle has seen
- `x-g`, the difference between the current position `x` and the best position `g` in the current neighborhood

Based on this formula, the options have the following effect:

- Larger absolute value of inertia W leads to the new velocity being more in the same line as the old, and with a larger absolute magnitude. A large absolute value of W can destabilize the swarm. The value of W stays within the range of the two-element vector `InertiaRange`.
- Larger values of $y_1 = \text{SelfAdjustmentWeight}$ make the particle head more toward the best place it has visited.
- Larger values of $y_2 = \text{SocialAdjustmentWeight}$ make the particle head more toward the best place in the current neighborhood.

Large values of inertia, `SelfAdjustmentWeight`, or `SocialAdjustmentWeight` can destabilize the swarm.

The `MinNeighborsFraction` option sets both the initial neighborhood size for each particle, and the minimum neighborhood size; see “Particle Swarm Optimization Algorithm” on page 10-11. Setting `MinNeighborsFraction` to 1 has all members of the swarm use the global minimum point as their societal adjustment target.

See “Optimize Using Particle Swarm” on page 10-5 for an example that sets a few of these tuning options.

Hybrid Function

A hybrid function is another minimization function that runs after the particle swarm algorithm terminates. You can specify a hybrid function in the `HybridFcn` option. The choices are

- `[]` — No hybrid function.
- `'fminsearch'` — Use the MATLAB function `fminsearch` to perform unconstrained minimization.
- `'patternsearch'` — Use a pattern search to perform constrained or unconstrained minimization.
- `'fminunc'` — Use the Optimization Toolbox function `fminunc` to perform unconstrained minimization.
- `'fmincon'` — Use the Optimization Toolbox function `fmincon` to perform constrained minimization.

Note Ensure that your hybrid function accepts your problem constraints. Otherwise, `particleswarm` throws an error.

You can set separate options for the hybrid function. Use `optimset` for `fminsearch`, or `optimoptions` for `fmincon`, `patternsearch`, or `fminunc`. For example:

```
hybridopts = optimoptions('fminunc',...
    'Display','iter','Algorithm','quasi-newton');
```

Include the hybrid options in the `particleswarm` options as follows:

```
options = optimoptions(options,'HybridFcn',{@fminunc,hybridopts});
```

`hybridopts` must exist before you set `options`.

For an example that uses a hybrid function, see “Optimize Using Particle Swarm” on page 10-5. See “When to Use a Hybrid Function” on page 8-116.

Output Function and Plot Function

Output functions are functions that `particleswarm` calls at each iteration. Output functions can halt `particleswarm`, or can perform other tasks. To specify an output function,

```
options = optimoptions(@particleswarm, 'OutputFcn', @outfun)
```

where `outfun` is a function with syntax specified in “Structure of the Output Function or Plot Function” on page 17-48. If you have several output functions, pass them as a cell array of function handles:

```
options = optimoptions(@particleswarm, ...
    'OutputFcn', {@outfun1, @outfun2, @outfun3})
```

Similarly, plot functions are functions that `particleswarm` calls at each iteration. The difference between an output function and a plot function is that a plot function has built-in plotting enhancements, such as buttons that appear on the plot window to pause or stop `particleswarm`. The lone built-in plot function `'pswplotbestf'` plots the best objective function value against iterations. To specify it,

```
options = optimoptions(@particleswarm, 'PlotFcn', 'pswplotbestf')
```

To create a custom plot function, write a function with syntax specified in “Structure of the Output Function or Plot Function” on page 17-48. To specify a custom plot function, use a function handle. If you have several plot functions, pass them as a cell array of function handles:

```
options = optimoptions(@particleswarm, ...
    'PlotFcn', {@plotfun1, @plotfun2, @plotfun3})
```

For an example of a custom output function, see “Particle Swarm Output Function” on page 10-8.

Structure of the Output Function or Plot Function

An output function has the following calling syntax:

```
stop = myfun(optimValues, state)
```

If your function sets `stop` to `true`, iterations end. Set `stop` to `false` to have `particleswarm` continue to calculate.

The function has the following input arguments:

- `optimValues` — Structure containing information about the swarm in the current iteration. Details are in “optimValues Structure” on page 17-49.
- `state` — String giving the state of the current iteration.
 - `'init'` — The solver has not begun to iterate. Your output function or plot function can use this state to open files, or set up data structures or plots for subsequent iterations.
 - `'iter'` — The solver is proceeding with its iterations. Typically, this is where your output function or plot function performs its work.
 - `'done'` — The solver reached a stopping criterion. Your output function or plot function can use this state to clean up, such as closing any files it opened.

“Passing Extra Parameters” explains how to provide additional parameters to output functions or plot functions.

optimValues Structure

particleswarm passes the `optimValues` structure to your output functions or plot functions. The `optimValues` structure has the following fields.

Field	Contents
<code>funccount</code>	Total number of objective function evaluations.
<code>bestx</code>	Best solution point found, corresponding to the best objective function value <code>bestfval</code> .
<code>bestfval</code>	Best (lowest) objective function value found.
<code>iteration</code>	Iteration number.
<code>meanfval</code>	Mean objective function among all particles at the current iteration.
<code>stalliterations</code>	Number of iterations since the last change in <code>bestfval</code> .
<code>swarm</code>	Matrix containing the particle positions. Each row contains the position of one particle, and the number of rows is equal to the swarm size.
<code>swarmfvals</code>	Vector containing the objective function values of particles in the swarm. For particle i , <code>swarmfvals(i) = fun(swarm(i,:))</code> , where <code>fun</code> is the objective function.

Parallel or Vectorized Function Evaluation

For increased speed, you can set your options so that `particleswarm` evaluates the objective function for the swarm in parallel or in a vectorized fashion. You can use only one of these options. If you set `UseParallel` to `true` and `UseVectorized` to `true`, then the computations are done in a vectorized fashion, and not in parallel.

- “Parallel particleswarm” on page 17-49
- “Vectorized particleswarm” on page 17-49

Parallel particleswarm

If you have a Parallel Computing Toolbox license, you can distribute the evaluation of the objective functions to the swarm among your processors or cores. Set the `UseParallel` option to `true`.

Parallel computation is likely to be faster than serial when your objective function is computationally expensive, or when you have many particles and processors. Otherwise, communication overhead can cause parallel computation to be slower than serial computation.

For details, see “Parallel Computing”.

Vectorized particleswarm

If your objective function can evaluate all the particles at once, you can usually save time by setting the `UseVectorized` option to `true`. Your objective function should accept an M-by-N matrix, where each row represents one particle, and return an M-by-1 vector of objective function values. This option works the same way as the `patternsearch` and `ga` `UseVectorized` options. For `patternsearch` details, see “Vectorize the Objective and Constraint Functions” on page 6-83.

Stopping Criteria

particleswarm stops iterating when any of the following occur.

Stopping Option	Stopping Test	Exit Flag
MaxStallIterations and FunctionTolerance	Relative change in the best objective function value g over the last MaxStallIterations iterations is less than FunctionTolerance.	1
MaxIterations	Number of iterations reaches MaxIterations.	0
OutputFcn or PlotFcn	OutputFcn or PlotFcn can halt the iterations.	-1
ObjectiveLimit	Best objective function value g of a feasible point is less than ObjectiveLimit.	-3
MaxStallTime	Best objective function value g did not change in the last MaxStallTime seconds.	-4
MaxTime	Function run time exceeds MaxTime seconds.	-5

Also, if you set the FunValCheck option to 'on', and the swarm has particles with NaN, Inf, or complex objective function values, particleswarm stops and issues an error.

Surrogate Optimization Options

Algorithm Control

To control the surrogate optimization algorithm, use the following options.

- **ConstraintTolerance** — The constraint tolerance is not used as a stopping criterion. It is used to determine feasibility with respect to nonlinear constraints. The tolerance is satisfied when $\max(\text{fun}(x).\text{Ineq}) \leq \text{ConstraintTolerance}$, and otherwise is violated. The default value is $1e-3$.
- **InitialPoints** — Specify initial points in one of two ways.
 - **Matrix** — Each row of the matrix represents an initial point. The length of each row is the same as the number of elements in the bounds `lb` or `ub`. The number of rows is arbitrary. `surrogateopt` uses all the rows to construct the initial surrogate. If there are fewer than `MinSurrogatePoints` rows, then `surrogateopt` generates the remaining initial points. `surrogateopt` evaluates the objective function at each initial point.
 - **Structure** — The structure contains the field `X` and, optionally, the fields `Fval` and `Ineq`. The `X` field contains a matrix where each row represents an initial point. The `Fval` field contains a vector representing the objective function values at each point in `X`. Passing `Fval` saves time for the solver, because otherwise the solver evaluates the objective function value at each initial point. The `Ineq` field contains a matrix containing nonlinear inequality constraint values. Each row of `Ineq` represents one initial point, and each column represents a nonlinear constraint function value at that point. Passing `Ineq` saves time for the solver, because otherwise the solver evaluates the constraint function values at each initial point.
- **MinSurrogatePoints** — Number of initial points used for constructing the surrogate. Larger values lead to a more accurate finished surrogate, but take more time to finish the surrogate. `surrogateopt` creates this number of random points after each switch to the random generation phase. See “Surrogate Optimization Algorithm” on page 11-3.

When `BatchUpdateInterval` > 1, the minimum number of random sample points used to create a surrogate is the larger of `MinSurrogatePoints` and `BatchUpdateInterval`.

- **MinSampleDistance** — This option controls two aspects of the algorithm.
 - During the phase to estimate the minimum value of the surrogate, the algorithm generates random points at which to evaluate the surrogate. If any of these points are closer than `MinSampleDistance` to any previous point whose objective function value was evaluated, then `surrogateopt` discards the newly generated points and does not evaluate them.
 - If `surrogateopt` discards all of the random points, then it does not try to minimize the surrogate and, instead, switches to the random generation phase. If the `surrogateoptplot` plot function is running, then it marks this switch with a blue vertical line.
- **BatchUpdateInterval** — This option controls three aspects of the algorithm:
 - Number of function evaluations before the surrogate is updated.
 - Number of points to pass in a vectorized evaluation. When `UseVectorized` is `true`, `surrogateopt` passes a matrix of size `BatchUpdateInterval`-by-`nvar`, where `nvar` is the number of problem variables. Each row of the matrix represents one evaluation point. For the final iteration (the one that causes `MaxFunctionEvaluations` function evaluations), if `MaxFunctionEvaluations` is not an integer multiple of `BatchUpdateInterval`, `surrogateopt` passes a matrix with fewer than `BatchUpdateInterval` rows.

- When `BatchUpdateInterval > 1`, the minimum number of random sample points used to create a surrogate is the larger of `MinSurrogatePoints` and `BatchUpdateInterval`.

Output functions and plot functions are updated after each batch is evaluated completely.

For details, see “Surrogate Optimization Algorithm” on page 11-3.

Stopping Criteria

Generally, the algorithm stops only when it reaches a limit that you set in the solver options. Additionally, a plot function or output function can halt the solver.

Stopping Option	Stopping Test	Exit Flag
<code>MaxFunctionEvaluations</code>	The solver stops after it completes <code>MaxFunctionEvaluations</code> function evaluations. When computing in parallel, the solver stops all workers after a worker returns with the final function evaluation, leaving some computations incomplete and unused.	0
<code>MaxTime</code>	The solver stops after it reaches <code>MaxTime</code> seconds from the start of the optimization, as measured by <code>tic / toc</code> . The solver does not interrupt a function evaluation in progress, so the actual compute time can exceed <code>MaxTime</code> .	0
<code>ObjectiveLimit</code>	The solver stops if an objective function value of a feasible point is less than <code>ObjectiveLimit</code> .	1
<code>OutputFcn</code> or <code>PlotFcn</code>	An <code>OutputFcn</code> or <code>PlotFcn</code> can halt the iterations.	-1
Bounds <code>lb</code> and <code>ub</code>	If an entry in <code>lb</code> exceeds the corresponding entry in <code>ub</code> , the solver stops because the bounds are inconsistent.	-2

Command-Line Display

Set the `Display` option to control what surrogateopt returns to the command line.

- `'final'` — Return only the exit message. This is the default behavior.
- `'iter'` — Return iterative display.
- `'off'` or the equivalent `'none'` — No command-line display.

With an iterative display, the solver returns the following information in table format.

- `F-count` — Number of function evaluations
- `Time(s)` — Time in seconds since the solver started
- `Best Fval` — Lowest objective function value obtained

- **Current Fval** — Latest objective function value
- **Trial Type** — Algorithm giving the evaluated point, either **random** or **adaptive**. For details, see “Surrogate Optimization Algorithm” on page 11-3.

When the objective function returns a nonlinear constraint, the iterative display of **Best Fval** and **Current Fval** changes. Instead, the titles are **Best** and **Current**, and each displays two columns, (**Fval**, **Infeas**).

- When a point is feasible, **surrogateopt** displays the function value, and shows - as the infeasibility.
- When a point is infeasible, **surrogateopt** displays the maximum infeasibility among all nonlinear constraint functions (a positive number), and shows - as the function value.
- Once **surrogateopt** finds a feasible point, subsequent entries in the **Best** column show only the smallest function value found, and show - as the best infeasibility.

With iterative display, the solver also returns problem information before the table:

- Number of variables
- Type of objective function (scalar or none)
- Number of inequalities

Output Function

An output function can halt the solver or perform a computation at each iteration. To include an output function, set the **OutputFcn** option to **@myoutputfcn**, where **myoutputfcn** is a function with the syntax described in the next paragraph. This syntax is the same as for Optimization Toolbox output functions, but with different meanings of the **x** and **optimValues** arguments. For information about those output functions, see “Output Function and Plot Function Syntax”. For an example of an output function, see “Optimal Component Choice Using **surrogateopt**” on page 11-67.

The syntax of an output function is:

```
stop = outfun(x,optimValues,state)
```

surrogateopt passes the values of **x**, **optimValues**, and **state** to the output function (**outfun**, in this case) at each iteration. The output function returns **stop**, a Boolean value (**true** or **false**) indicating whether to stop **surrogateopt**.

- **x** — The input argument **x** is the best point found so far, meaning the point with the lowest objective function value.
- **optimValues** — This input argument is a structure containing the following fields. For more information about these fields, see “Surrogate Optimization Algorithm” on page 11-3.

optimValues Structure

Field Name	Contents
constrviolation	Maximum constraint violation of best point, <code>max(optimValues.ineq)</code>
currentConstrviolation	Maximum constraint violation of current point, <code>max(optimValues.currentIneq)</code>
currentFlag	How the current point was created. <ul style="list-style-type: none"> 'initial' — Initial point passed in <code>options.InitialPoints</code> 'random' — Random sample within the bounds 'adaptive' — Result of the solver trying to minimize the surrogate
currentFval	Objective function value at the current point
currentIneq	Constraint violation vector of current point, <code>fun(currentX).Ineq</code>
currentX	Current point
elapsedtime	Time in seconds since the solver started
flag	How the best point was created <ul style="list-style-type: none"> 'initial' — Initial point passed in <code>options.InitialPoints</code> 'random' — Random sample within the bounds 'adaptive' — Result of the solver trying to minimize the surrogate
funccount	Total number of objective function evaluations
fval	Lowest objective function value encountered
incumbentConstrviolation	Maximum constraint violation of current point, <code>max(optimValues.incumbentIneq)</code>
incumbentIneq	Constraint violation vector of incumbent point, <code>fun(incumbentX).Ineq</code>
incumbentFlag	How the incumbent point was created <ul style="list-style-type: none"> 'initial' — Initial point passed in <code>options.InitialPoints</code> 'random' — Random sample within the bounds 'adaptive' — Result of the solver trying to minimize the surrogate
incumbentFval	Objective function value at the incumbent point
incumbentX	Incumbent point, meaning the best point found since the last phase shift to random sampling
ineq	Constraint violation vector of best point, <code>fun(x).Ineq</code>

Field Name	Contents
iteration	Number of iterations completed. Equal to <code>funccount</code> , except during the 'iter' state, when iteration is equal to <code>funccount - 1</code> . This field allows <code>surrogateopt</code> to use the same plot functions as some other solvers
surrogateReset	Boolean value indicating that the current iteration resets the model and switches to random sampling
surrogateResetCount	Total number of times that <code>surrogateReset</code> is true

- `state` — This input argument is the state of the algorithm, specified as one of these values.
 - 'init' — The algorithm is in the initial state before the first iteration. When the algorithm is in this state, you can set up plot axes or other data structures or open files.

Note When `state` is 'init', the input arguments `x` and `optimValues.fval` are empty (`[]`) because `surrogateopt` is designed for time-consuming objective functions, and so does not evaluate the objective function before calling the initialization step.

- 'iter' — The algorithm just evaluated the objective function. You perform most calculations and view most displays when the algorithm is in this state.
- 'done' — The algorithm performed its final objective function evaluation. When the algorithm is in this state, you can close files, finish plots, or prepare in other ways for `surrogateopt` to stop.

Plot Function

A plot function displays information at each iteration. You can pause or halt the solver by clicking buttons on the plot. To include a plot function, set the `PlotFcn` option to a function name or function handle or cell array of function names or handles to plot functions. The four built-in plot functions are:

- 'optimplotfvalconstr' (default) — Plot the best feasible objective function value found as a line plot. If there is no objective function, plot the maximum nonlinear constraint violation as a line plot.
 - The plot shows infeasible points as red and feasible points as blue.
 - If there is no objective function, the plot title shows the number of feasible solutions.
- 'optimplotfval' — Shows the best function value. If you do not choose a plot function, `surrogateopt` uses `@optimplotfval`.
- 'optimplotx' — Shows the best point found as a bar plot.
- 'surrogateoptplot' — Shows the current objective function value, best function value, and information about the algorithm phase. See “Interpret `surrogateoptplot`” on page 11-25.

You can write a custom plot function using the syntax of an “Output Function” on page 17-53. For an example, examine the code for `surrogateoptplot` by entering `type surrogateoptplot` at the MATLAB command line.

Parallel Computing

When you set the `UseParallel` option to `true`, `surrogateopt` computes in parallel. Computing in parallel requires a Parallel Computing Toolbox license. For details, see “Surrogate Optimization Algorithm” on page 11-3.

You cannot specify both `UseParallel = true` and `UseVectorized = true`. If you set both to `true`, the solver ignores `UseVectorized` and attempts to compute in parallel using a parallel pool, if possible.

Vectorized Computing

When you set the `UseVectorized` option to `true`, `surrogateopt` passes a matrix to the objective function. Each row of the matrix represents one point to evaluate. The matrix has `options.BatchUpdateInterval` rows; however, the matrix can have fewer rows during the final iteration. Use this option for custom parallel computing, as shown in “Vectorized Surrogate Optimization for Custom Parallel Simulation” on page 11-92.

You cannot specify both `UseParallel = true` and `UseVectorized = true`. If you set both to `true`, the solver ignores `UseVectorized` and attempts to compute in parallel using a parallel pool, if possible.

Checkpoint File

When you set the name of a checkpoint file using the `CheckpointFile` option, `surrogateopt` writes data to the file after each iteration, which enables the function to resume the optimization from the current state. When restarting, `surrogateopt` does not evaluate the objective function value at previously evaluated points.

A checkpoint file can be a file path such as `"C:\Documents\MATLAB\check1.mat"` or a file name such as `'checkpoint1June2019.mat'`. If you specify a file name without a path, `surrogateopt` saves the checkpoint file in the current folder.

You can change only the following options when resuming the optimization:

- `BatchUpdateInterval`
- `CheckpointFile`
- `Display`
- `MaxFunctionEvaluations`
- `MaxTime`
- `MinSurrogatePoints`
- `ObjectiveLimit`
- `OutputFcn`
- `PlotFcn`
- `UseParallel`
- `UseVectorized`

To resume the optimization from a checkpoint file, call `surrogateopt` with the file name as the first argument.


```
[x,fval,exitflag,output] = surrogateopt('check1.mat')
```

To resume the optimization using new options, include the new options as the second argument.

```
opts = optimoptions(options,'MaxFunctionEvaluations',500);  
[x,fval,exitflag,output] = surrogateopt('check1.mat',opts)
```

During the restart, `surrogateopt` runs any output functions and plot functions, based on the original function evaluations. So, for example, you can create a different plot based on an optimization that already ran. See “Work with Checkpoint Files” on page 11-56.

Note `surrogateopt` does not save all details of the state in the checkpoint file. Therefore, subsequent iterations can differ from the iterations that the solver takes without stopping at the checkpointed state.

Note Checkpointing takes time. This overhead is especially noticeable for functions that otherwise take little time to evaluate.

Warning Do not resume `surrogateopt` from a checkpoint file created with a different MATLAB version. `surrogateopt` can throw an error or give inconsistent results.

See Also

`surrogateopt`

More About

- “Surrogate Optimization”
- “Surrogate Optimization Algorithm” on page 11-3

Simulated Annealing Options

In this section...

“Set Simulated Annealing Options at the Command Line” on page 17-58
 “Plot Options” on page 17-58
 “Temperature Options” on page 17-59
 “Algorithm Settings” on page 17-60
 “Hybrid Function Options” on page 17-61
 “Stopping Criteria Options” on page 17-62
 “Output Function Options” on page 17-62
 “Display Options” on page 17-64

Set Simulated Annealing Options at the Command Line

Specify options by creating an options object using the `optimoptions` function as follows:

```
options = optimoptions(@simulannealbnd,...
    'Param1',value1,'Param2',value2, ...);
```

Each option in this section is listed by its field name in `options`. For example, `InitialTemperature` refers to the corresponding field of `options`.

Plot Options

Plot options enable you to plot data from the simulated annealing solver while it is running.

`PlotInterval` specifies the number of iterations between consecutive calls to the plot function.

To display a plot when calling `simulannealbnd` from the command line, set the `PlotFcn` field of `options` to be a built-in plot function name or handle to the plot function. You can specify any of the following plots:

- `'saplotbestf'` plots the best objective function value.
- `'saplotbestx'` plots the current best point.
- `'saplotf'` plots the current function value.
- `'saplotx'` plots the current point.
- `'saplotstopping'` plots stopping criteria levels.
- `'saplottemperature'` plots the temperature at each iteration.
- `@myfun` plots a custom plot function, where `myfun` is the name of your function. See “Structure of the Plot Functions” on page 17-8 for a description of the syntax.

For example, to display the best objective plot, set `options` as follows

```
options = optimoptions(@simulannealbnd,'PlotFcn','saplotbestf');
```

To display multiple plots, use the cell array syntax

```
options = optimoptions(@simulannealbnd,...
    'PlotFcn',{@plotfun1,@plotfun2, ...});
```

where `@plotfun1`, `@plotfun2`, and so on are function handles to the plot functions.

If you specify more than one plot function, all plots appear as subplots in the same window. Right-click any subplot to obtain a larger version in a separate figure window.

Structure of the Plot Functions

The first line of a plot function has the form

```
function stop = plotfun(options,optimvalues,flag)
```

The input arguments to the function are

- `options` — Options created using `optimoptions`.
- `optimvalues` — Structure containing information about the current state of the solver. The structure contains the following fields:
 - `x` — Current point
 - `fval` — Objective function value at `x`
 - `bestx` — Best point found so far
 - `bestfval` — Objective function value at best point
 - `temperature` — Current temperature
 - `iteration` — Current iteration
 - `funccount` — Number of function evaluations
 - `t0` — Start time for algorithm
 - `k` — Annealing parameter
- `flag` — Current state in which the plot function is called. The possible values for `flag` are
 - `'init'` — Initialization state
 - `'iter'` — Iteration state
 - `'done'` — Final state

The output argument `stop` provides a way to stop the algorithm at the current iteration. `stop` can have the following values:

- `false` — The algorithm continues to the next iteration.
- `true` — The algorithm terminates at the current iteration.

Temperature Options

Temperature options specify how the temperature will be lowered at each iteration over the course of the algorithm.

- `InitialTemperature` — Initial temperature at the start of the algorithm. The default is `100`. The initial temperature can be a vector with the same length as `x`, the vector of unknowns. `simulannealbnd` expands a scalar initial temperature into a vector.
- `TemperatureFcn` — Function used to update the temperature schedule. Let `k` denote the annealing parameter. (The annealing parameter is the same as the iteration number until reannealing.) The options are:

- 'temperatureexp' — The temperature is equal to $\text{InitialTemperature} * 0.95^k$. This is the default.
- 'temperaturefast' — The temperature is equal to $\text{InitialTemperature} / k$.
- 'temperatureboltz' — The temperature is equal to $\text{InitialTemperature} / \ln(k)$.
- @myfun — Uses a custom function, myfun, to update temperature. The syntax is:

```
temperature = myfun(optimValues,options)
```

where `optimValues` is a structure described in “Structure of the Plot Functions” on page 17-59. `options` is either created with `optimoptions`, or consists of default options, if you did not create any options. Both the annealing parameter `optimValues.k` and the temperature `optimValues.temperature` are vectors with length equal to the number of elements of the current point `x`. For example, the function `temperaturefast` is:

```
temperature = options.InitialTemperature./optimValues.k;
```

Algorithm Settings

Algorithm settings define algorithmic specific parameters used in generating new points at each iteration.

Parameters that can be specified for `simulannealbnd` are:

- `DataType` — Type of data to use in the objective function. Choices:
 - 'double' (default) — A vector of type `double`.
 - 'custom' — Any other data type. You must provide a 'custom' annealing function. You cannot use a hybrid function.
- `AnnealingFcn` — Function used to generate new points for the next iteration. The choices are:
 - 'annealingfast' — The step has length `temperature`, with direction uniformly at random. This is the default.
 - 'annealingboltz' — The step has length square root of `temperature`, with direction uniformly at random.
 - @myfun — Uses a custom annealing algorithm, myfun. The syntax is:

```
newx = myfun(optimValues,problem)
```

where `optimValues` is a structure described in “Structure of the Output Function” on page 17-63, and `problem` is a structure containing the following information:

- `objective`: function handle to the objective function
- `x0`: the start point
- `nvar`: number of decision variables
- `lb`: lower bound on decision variables
- `ub`: upper bound on decision variables

For example, the current position is `optimValues.x`, and the current objective function value is `problem.objective(optimValues.x)`.

You can write a custom objective function by modifying the `saannealingfcn_template.m` file. To keep all iterates within bounds, have your custom annealing function call `saonorbounds` as the final command.

- `ReannealInterval` — Number of points accepted before reannealing. The default value is 100.
- `AcceptanceFcn` — Function used to determine whether a new point is accepted or not. The choices are:
 - `'acceptancesa'` — Simulated annealing acceptance function, the default. If the new objective function value is less than the old, the new point is always accepted. Otherwise, the new point is accepted at random with a probability depending on the difference in objective function values and on the current temperature. The acceptance probability is

$$\frac{1}{1 + \exp\left(\frac{\Delta}{\max(T)}\right)},$$

where Δ = new objective - old objective, and T is the current temperature. Since both Δ and T are positive, the probability of acceptance is between 0 and 1/2. Smaller temperature leads to smaller acceptance probability. Also, larger Δ leads to smaller acceptance probability.

- `@myfun` — A custom acceptance function, `myfun`. The syntax is:

```
acceptpoint = myfun(optimValues,newx,newfval);
```

where `optimValues` is a structure described in “Structure of the Output Function” on page 17-63, `newx` is the point being evaluated for acceptance, and `newfval` is the objective function at `newx`. `acceptpoint` is a Boolean, with value `true` to accept `newx`, and `false` to reject `newx`.

Hybrid Function Options

A hybrid function is another minimization function that runs during or at the end of iterations of the solver. `HybridInterval` specifies the interval (if not `never` or `end`) at which the hybrid function is called. You can specify a hybrid function using the `HybridFcn` option. The choices are:

- `[]` — No hybrid function.
- `'fminsearch'` — Uses the MATLAB function `fminsearch` to perform unconstrained minimization.
- `'patternsearch'` — Uses `patternsearch` to perform constrained or unconstrained minimization.
- `'fminunc'` — Uses the Optimization Toolbox function `fminunc` to perform unconstrained minimization.
- `'fmincon'` — Uses the Optimization Toolbox function `fmincon` to perform constrained minimization.

Note Ensure that your hybrid function accepts your problem constraints. Otherwise, `simulannealbnd` throws an error.

You can set separate options for the hybrid function. Use `optimset` for `fminsearch`, or `optimoptions` for `fmincon`, `patternsearch`, or `fminunc`. For example:

```
hybridopts = optimoptions('fminunc',...  
    'Display','iter','Algorithm','quasi-newton');
```

Include the hybrid options in the `simulannealbnd` options as follows:

```
options = optimoptions(@simulannealbnd,options,...  
    'HybridFcn',{@fminunc,hybridopts});
```

`hybridopts` must exist before you set `options`.

See “Hybrid Scheme in the Genetic Algorithm” on page 8-95 for an example. See “When to Use a Hybrid Function” on page 8-116.

Stopping Criteria Options

Stopping criteria determine what causes the algorithm to terminate. You can specify the following options:

- `FunctionTolerance` — The algorithm runs until the average change in value of the objective function in `StallIterLim` iterations is less than `FunctionTolerance`. The default value is `1e-6`.
- `MaxIterations` — The algorithm stops if the number of iterations exceeds this maximum number of iterations. You can specify the maximum number of iterations as a positive integer or `Inf`. `Inf` is the default.
- `MaxFunctionEvaluations` specifies the maximum number of evaluations of the objective function. The algorithm stops if the number of function evaluations exceeds the maximum number of function evaluations. The allowed maximum is `3000*numberofvariables`.
- `MaxTime` specifies the maximum time in seconds the algorithm runs before stopping.
- `ObjectiveLimit` — The algorithm stops if the best objective function value of a feasible point is less than `ObjectiveLimit`.

Output Function Options

Output functions are functions that the algorithm calls at each iteration. The default value is to have no output function, `[]`. You must first create an output function using the syntax described in “Structure of the Output Function” on page 17-63.

Using the Optimization app:

- Specify **Output function** as `@myfun`, where `myfun` is the name of your function.
- To pass extra parameters in the output function, use “Anonymous Functions”.
- For multiple output functions, enter a cell array of output function handles: `{@myfun1,@myfun2,...}`.

At the command line:

- `options = optimoptions(@simulannealbnd,'OutputFcn',@myfun);`
- For multiple output functions, enter a cell array of function handles:

```
options = optimoptions(@simulannealbnd,...  
    'OutputFcn',{@myfun1,@myfun2,...});
```

To see a template that you can use to write your own output functions, enter

```
edit saoutputfcntemplate
```

at the MATLAB command line.

Structure of the Output Function

The output function has the following calling syntax.

```
[stop,options,optchanged] = myfun(options,optimvalues,flag)
```

The function has the following input arguments:

- `options` — Options created using `optimoptions`.
- `optimvalues` — Structure containing information about the current state of the solver. The structure contains the following fields:
 - `x` — Current point
 - `fval` — Objective function value at `x`
 - `bestx` — Best point found so far
 - `bestfval` — Objective function value at best point
 - `temperature` — Current temperature, a vector the same length as `x`
 - `iteration` — Current iteration
 - `funccount` — Number of function evaluations
 - `t0` — Start time for algorithm
 - `k` — Annealing parameter, a vector the same length as `x`
- `flag` — Current state in which the output function is called. The possible values for `flag` are
 - `'init'` — Initialization state
 - `'iter'` — Iteration state
 - `'done'` — Final state

“Passing Extra Parameters” explains how to provide additional parameters to the output function.

The output function returns the following arguments:

- `stop` — Provides a way to stop the algorithm at the current iteration. `stop` can have the following values:
 - `false` — The algorithm continues to the next iteration.
 - `true` — The algorithm terminates at the current iteration.
- `options` — Options as modified by the output function.
- `optchanged` — A Boolean flag indicating changes were made to `options`. This must be set to `true` if options are changed.

Display Options

Use the `Display` option to specify how much information is displayed at the command line while the algorithm is running. The available options are

- `off` — No output is displayed. This is the default value for `options` exported from the Optimization app.
- `iter` — Information is displayed at each iteration.
- `diagnose` — Information is displayed at each iteration. In addition, the diagnostic lists some problem information and the options that have been changed from the defaults.
- `final` — The reason for stopping is displayed. This is the default for options created using `optimoptions`.

Both `iter` and `diagnose` display the following information:

- `Iteration` — Iteration number
- `f-count` — Cumulative number of objective function evaluations
- `Best f(x)` — Best objective function value
- `Current f(x)` — Current objective function value
- `Mean Temperature` — Mean temperature function value

Options Changes in R2016a

In this section...

“Use `optimoptions` to Set Options” on page 17-65

“Options that `optimoptions` Hides” on page 17-65

“Table of Option Names in Legacy Order” on page 17-67

“Table of Option Names in Current Order” on page 17-70

Use `optimoptions` to Set Options

Before R2016a, you set options for some Global Optimization Toolbox solvers by using a dedicated option function:

- `gaoptimset` for `ga` and `gamultiobj`
- `psoptimset` for `patternsearch`
- `saoptimset` for `simulannealbnd`

Beginning in R2016a, the recommended way to set options is to use `optimoptions`. (You already set `particleswarm` options using `optimoptions`.)

Note `GlobalSearch` and `MultiStart` use a different mechanism for setting properties. See “GlobalSearch and MultiStart Properties (Options)” on page 17-2. Some of these property names changed as solver option names changed.

Some option names changed in R2016a. See “Table of Option Names in Legacy Order” on page 17-67.

`optimoptions` “hides” some options, meaning it does not display their values. `optimoptions` displays only current names, not legacy names. For details, see “View Optimization Options”.

Options that `optimoptions` Hides

`optimoptions` does not display some options. To view the setting of any such “hidden” option, use dot notation. For details, see “View Optimization Options”. These options are listed in *italics* in the options tables in the function reference pages.

Options that optioptions Hides

Option	Description	Solvers	Reason for Hiding
<i>Cache</i>	With Cache set to 'on', patternsearch keeps a history of the mesh points it polls. At subsequent iterations, patternsearch does not poll points close to those it already polled. Use this option if patternsearch runs slowly while computing the objective function. If the objective function is stochastic, do not use this option.	patternsearch	Works poorly
<i>CacheSize</i>	Size of the history.	patternsearch	Works poorly
<i>CacheTol</i>	Largest distance from the current mesh point to any point in the history in order for patternsearch to avoid polling the current point. Use if 'Cache' option is set to 'on'.	patternsearch	Works poorly
<i>DisplayInterval</i>	Interval for iterative display. The iterative display prints one line for every DisplayInterval iterations.	particleswarm, simulannealbnd	Not generally useful
<i>FunValCheck</i>	Check whether objective function and constraints values are valid. 'on' displays an error when the objective function or constraints return a value that is complex, Inf, or NaN.	particleswarm	Not generally useful
<i>HybridInterval</i>	Interval (if not 'end' or 'never') at which HybridFcn is called.	simulannealbnd	Not generally useful
<i>InitialPenalty</i>	Initial value of penalty parameter.	ga, patternsearch	Difficult to know how to set

Option	Description	Solvers	Reason for Hiding
<i>MaxMeshSize</i>	Maximum mesh size used in a poll or search step.	patternsearch	Not generally useful
<i>MeshRotate</i>	Rotate the pattern before declaring a point to be optimum.	patternsearch	Default value is best
<i>MigrationDirection</i>	Direction of migration — see “Migration Options” on page 17-37.	ga	Not useful
<i>MigrationFraction</i>	Scalar between 0 and 1 specifying the fraction of individuals in each subpopulation that migrates to a different subpopulation — see “Migration Options” on page 17-37.	ga	Not useful
<i>MigrationInterval</i>	Positive integer specifying the number of generations that take place between migrations of individuals between subpopulations — see “Migration Options” on page 17-37	ga	Not useful
<i>PenaltyFactor</i>	Penalty update parameter.	ga, patternsearch	Difficult to know how to set
<i>PlotInterval</i>	Positive integer specifying the number of generations between consecutive calls to the plot functions.	ga, patternsearch, simulannealrnd	Not useful
<i>StallTest</i>	String describing the stopping test.	ga	Default value is best
<i>TolBind</i>	Binding tolerance. See “Constraint Parameters” on page 17-16.	patternsearch	Default value is usually best

Table of Option Names in Legacy Order

These two tables have identical information. One is in alphabetical order by legacy option name, the other is in order by current option name. The tables show values only when the values differ between legacy and current, and show only the names that differ. For changes in Optimization Toolbox solvers, see “Current and Legacy Option Names”.

* indicates GlobalSearch and MultiStart property names as well as solver option names.

Option Names in Legacy Order

Legacy Name	Current Name	Legacy Values	Current Values
CompletePoll	UseCompletePoll	'on', 'off'	true, false
CompleteSearch	UseCompleteSearch	'on', 'off'	true, false
Generations	MaxGenerations		
InitialPopulation	InitialPopulationMatrix		
InitialScores	InitialScoreMatrix		
InitialSwarm	InitialSwarmMatrix		
MaxFunEvals	MaxFunctionEvaluations		
MaxIter	MaxIterations		
MeshAccelerator	AccelerateMesh	'on', 'off'	true, false
MeshContraction	MeshContractionFactor		
MeshExpansion	MeshExpansionFactor		
MinFractionNeighbors	MinNeighborsFraction		
NonlinConAlgorithm	NonlinearConstraintAlgorithm		
* OutputFcns	* OutputFcn		
* PlotFcns	* PlotFcn		
PollingOrder	PollOrderAlgorithm		
PopInitRange	InitialPopulationRange		
SearchMethod	SearchFcn		
SelfAdjustment	SelfAdjustmentWeight		
SocialAdjustment	SocialAdjustmentWeight		
StallGenLimit	MaxStallGenerations		
StallIterLimit	MaxStallIterations		
StallTimeLimit	MaxStallTime		
TimeLimit	MaxTime		
TolCon	ConstraintTolerance		
* TolFun	* FunctionTolerance		
TolMesh	MeshTolerance		

Legacy Name	Current Name	Legacy Values	Current Values
* TolX	StepTolerance * XTolerance for GlobalSearch and MultiStart		
Vectorized	UseVectorized	'on', 'off'	true, false

Table of Option Names in Current Order

* indicates GlobalSearch and MultiStart property names as well as solver option names.

Option Names in Current Order

Current Name	Legacy Name	Current Values	Legacy Values
AccelerateMesh	MeshAccelerator	true, false	'on', 'off'
ConstraintTolerance	TolCon		
*FunctionTolerance	*TolFun		
InitialPopulationMatrix	InitialPopulation		
InitialPopulationRange	PopInitRange		
InitialScoreMatrix	InitialScores		
InitialSwarmMatrix	InitialSwarm		
MaxFunctionEvaluations	MaxFunEvals		
MaxGenerations	Generations		
MaxIterations	MaxIter		
MaxStallGenerations	StallGenLimits		
MaxStallIterations	StallIterLimit		
MaxStallTime	StallTimeLimit		
MaxTime	TimeLimit		
MeshContractionFactor	MeshContraction		
MeshExpansionFactor	MeshExpansion		
MeshTolerance	TolMesh		
MinNeighborsFraction	MinFractionNeighbors		
NonlinearConstraintAlgorithm	NonlinConAlgorithm		
*OutputFcn	*OutputFcns		
*PlotFcn	*PlotFcns		
PollOrderAlgorithm	PollingOrder		
SearchFcn	SearchMethod		
SelfAdjustmentWeight	SelfAdjustment		
SocialAdjustmentWeight	SocialAdjustment		
StepTolerance	TolX		
UseCompletePoll	CompletePoll	true, false	'on', 'off'

Current Name	Legacy Name	Current Values	Legacy Values
UseCompleteSearch	CompleteSearch	true, false	'on', 'off'
UseVectorized	Vectorized	true, false	'on', 'off'
* XTolerance	* TolX		

Functions

createOptimProblem

Create optimization problem structure

Syntax

```
problem = createOptimProblem(solverName)
problem = createOptimProblem(solverName,Name,Value)
```

Description

`problem = createOptimProblem(solverName)` creates an empty optimization problem structure for the `solverName` solver.

`problem = createOptimProblem(solverName,Name,Value)` specifies additional options using one or more name-value arguments.

Examples

Create and Run `fmincon` Problem Structure

Create a problem structure with the following specifications:

- `fmincon` solver
- "interior-point" algorithm
- Random 2-D initial point `x0`
- Rosenbrock's function as the objective
- Lower bounds of -2
- Upper bounds of 2

Rosenbrock's function for a 2-D variable x is $f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$ (for details, see "Constrained Nonlinear Problem Using Optimize Live Editor Task or Solver"). To specify the "interior-point" algorithm, create options using `optimoptions`.

```
anonrosen = @(x)(100*(x(2) - x(1)^2)^2 + (1-x(1))^2);
opts = optimoptions(@fmincon,Algorithm="interior-point");
rng default % For reproducibility
problem = createOptimProblem("fmincon",...
    x0=randn(2,1),...
    objective=anonrosen,...
    lb=[-2;-2],...
    ub=[2;2],...
    options=opts);
```

Solve the problem starting from `problem.x0` by calling `fmincon`.

```
[x,fval] = fmincon(problem)
```

Local minimum possible. Constraints satisfied.

fmincon stopped because the size of the current step is less than the value of the step size tolerance and constraints are satisfied to within the value of the constraint tolerance.

```
x = 2×1
```

```
    1.0000
    1.0000
```

```
fval = 2.0603e-11
```

Look for a better solution by calling GlobalSearch.

```
gs = GlobalSearch;
[x2,fval2] = run(gs,problem)
```

GlobalSearch stopped because it analyzed all the trial points.

All 16 local solver runs converged with a positive local solver exit flag.

```
x2 = 2×1
```

```
    1.0000
    1.0000
```

```
fval2 = 2.1093e-11
```

In this case, both fmincon and GlobalSearch reach the same solution.

Input Arguments

solverName — Optimization solver

"fmincon" | @fmincon | "fminunc" | @fminunc | "lsqnonlin" | @lsqnonlin | "lsqcurvefit" | @lsqcurvefit

Optimization solver, specified as one of the following.

- For GlobalSearch, specify "fmincon" or @fmincon.
- For MultiStart, specify "fmincon" or @fmincon, "fminunc" or @fminunc, "lsqnonlin" or @lsqnonlin, or "lsqcurvefit" or @lsqcurvefit.

Example: "fmincon"

Data Types: char | string | function_handle

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1, . . . , NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example:

```
createOptimProblem("fmincon","x0",x0,"objective",fun,"lb",zeros(size(x0)))
```

Aeq — Linear equality constraints

real matrix

Linear equality constraints, specified as a real matrix. `Aeq` is an `Me-by-nvars` matrix, where `Me` is the number of equalities.

`Aeq` encodes the `Me` linear equalities

$$\text{Aeq} * \mathbf{x} = \text{beq},$$

where \mathbf{x} is the column vector of `N` variables `x(:)`, and `beq` is a column vector with `Me` elements.

For example, to specify

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 10 \\ 2x_1 + 4x_2 + x_3 &= 20, \end{aligned}$$

give these constraints:

$$\begin{aligned} \text{Aeq} &= [1, 2, 3; 2, 4, 1]; \\ \text{beq} &= [10; 20]; \end{aligned}$$

Example: To specify that the control variables sum to 1, give the constraints `Aeq = ones(1,N)` and `beq = 1`.

Data Types: double

Aineq — Linear inequality constraints

real matrix

Linear inequality constraints, specified as a real matrix. `Aineq` is an `M-by-nvars` matrix, where `M` is the number of inequalities.

`Aineq` encodes the `M` linear inequalities

$$\text{Aineq} * \mathbf{x} \leq \text{bineq},$$

where \mathbf{x} is the column vector of `nvars` variables `x(:)`, and `bineq` is a column vector with `M` elements.

For example, to specify

$$\begin{aligned} x_1 + 2x_2 &\leq 10 \\ 3x_1 + 4x_2 &\leq 20 \\ 5x_1 + 6x_2 &\leq 30, \end{aligned}$$

give these constraints:

$$\begin{aligned} \text{Aineq} &= [1, 2; 3, 4; 5, 6]; \\ \text{bineq} &= [10; 20; 30]; \end{aligned}$$

Example: To specify that the control variables sum to 1 or less, give the constraints `Aineq = ones(1,N)` and `bineq = 1`.

Data Types: double

beq — Linear equality constraints

real vector

Linear equality constraints, specified as a real vector. `beq` is an M -element vector related to the `Aeq` matrix. If you pass `beq` as a row vector, solvers internally convert `beq` to the column vector `beq(:)`.

`beq` encodes the M linear equalities

$$Aeq \cdot x = beq,$$

where x is the column vector of N variables `x(:)`, and `Aeq` is a matrix of size M -by- N .

For example, to specify

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 10 \\ 2x_1 + 4x_2 + x_3 &= 20, \end{aligned}$$

give these constraints:

$$\begin{aligned} Aeq &= [1, 2, 3; 2, 4, 1]; \\ beq &= [10; 20]; \end{aligned}$$

Example: To specify that the control variables sum to 1, give the constraints `Aeq = ones(1,N)` and `beq = 1`.

Data Types: `double`

bineq – Linear inequality constraints

real vector

Linear inequality constraints, specified as a real vector. `bineq` is an M -element vector related to the `Aineq` matrix. If you pass `bineq` as a row vector, solvers internally convert `bineq` to the column vector `bineq(:)`.

`bineq` encodes the M linear inequalities

$$Aineq \cdot x \leq bineq,$$

where x is the column vector of N variables `x(:)`, and `Aineq` is a matrix of size M -by- N .

For example, to specify

$$\begin{aligned} x_1 + 2x_2 &\leq 10 \\ 3x_1 + 4x_2 &\leq 20 \\ 5x_1 + 6x_2 &\leq 30, \end{aligned}$$

give these constraints:

$$\begin{aligned} Aineq &= [1, 2; 3, 4; 5, 6]; \\ bineq &= [10; 20; 30]; \end{aligned}$$

Example: To specify that the control variables sum to 1 or less, give the constraints `Aineq = ones(1,N)` and `bineq = 1`.

Data Types: `double`

lb – Lower bounds

`[]` (default) | real vector or array

Lower bounds, specified as a real vector or array of doubles. `lb` represents the lower bounds element-wise in $lb \leq x \leq ub$.

Internally, `createOptimProblem` converts an array `lb` to the vector `lb(:)`.

Example: `lb = [0; -Inf; 4]` means $x(1) \geq 0$, $x(3) \geq 4$.

Data Types: `double`

nonlcon — Nonlinear constraints

function handle | function name

Nonlinear constraints, specified as a function handle or function name. `nonlcon` is a function that accepts an array `x` and returns two arrays, `c(x)` and `ceq(x)`.

- `c(x)` is the array of nonlinear inequality constraints at `x`. The solver attempts to satisfy $c(x) \leq 0$ for all entries of `c`.
- `ceq(x)` is the array of nonlinear equality constraints at `x`. The solver attempts to satisfy $ceq(x) = 0$ for all entries of `ceq`.

For example, `nonlcon` is a MATLAB function such as the following:

```
function [c,ceq] = nonlcon(x)
c = ...      % Compute nonlinear inequalities at x.
ceq = ...    % Compute nonlinear equalities at x.
```

For more information, see “Nonlinear Constraints”.

Data Types: `char` | `string` | `function_handle`

objective — Objective function

function handle | function name

Objective function, specified as a function handle or function name.

- For all solvers except `lsqnonlin` and `lsqcurvefit`, the objective function must accept an array `x` and return a scalar. If the `SpecifyObjectiveGradient` option is `true`, then the objective function must return a second output, a vector representing the gradient of the objective. For details, see `fun`.
- For `lsqnonlin`, the objective function must accept a vector `x` and return a vector. If the `SpecifyObjectiveGradient` option is `true`, then the objective function must return a second output, a matrix representing the Jacobian of the objective. For details, see `fun`.
- For `lsqcurvefit`, the objective function must accept two inputs, `x` and `xdata`, and return a vector. If the `SpecifyObjectiveGradient` option is `true`, then the objective function must return a second output, a matrix representing the Jacobian of the objective. For details, see `fun`.

Example: `@sin`

Example: `"sin"`

Data Types: `char` | `string` | `function_handle`

options — Optimization options

output of `optimoptions`

Optimization options, specified as the output of `optimoptions`.

Example: `optimoptions("fmincon","SpecifyObjectiveGradient",true)`

ub — Upper bounds

`[]` (default) | real vector or array

Upper bounds, specified as a real vector or array of doubles. `ub` represents the upper bounds element-wise in $lb \leq x \leq ub$.

Internally, `createOptimProblem` converts an array `ub` to the vector `ub(:)`.

Example: `ub = [Inf;4;10]` means $x(2) \leq 4$, $x(3) \leq 10$.

Data Types: `double`

x0 — Initial point

real vector | real array

Initial point, specified as a real vector or real array. Solvers use the number of elements in `x0` and the size of `x0` to determine the number and size of variables that `fun` accepts.

Example: `x0 = [1,2,3,4]`

Data Types: `double`

xdata — Input data for model

real vector | real array

Input data for the model, specified as a real vector or real array. The model is $ydata = fun(x, xdata)$,

where `xdata` and `ydata` are fixed arrays, and `x` is the array of parameters that `lsqcurvefit` changes to search for a minimum sum of squares.

Example: `xdata = [1,2,3,4]`

Data Types: `double`

ydata — Response data for model

real vector | real array

Response data for the model, specified as a real vector or real array. The model is $ydata = fun(x, xdata)$,

where `xdata` and `ydata` are fixed arrays, and `x` is the array of parameters that `lsqcurvefit` changes to search for a minimum sum of squares.

The `ydata` array must be the same size and shape as the array `fun(x0, xdata)`.

Example: `ydata = [1,2,3,4]`

Data Types: `double`

Output Arguments

problem — Optimization problem

structure

Optimization problem, returned as a structure. Use `problem` as the second input argument of `run`, as in the following examples:

```
x = run(gs, problem)
x = run(ms, problem, k)
```

You can also solve the problem by calling the named solver on the problem. For example, if `problem` is created for `fmincon`, enter

```
x = fmincon(problem)
```

Version History

Introduced in R2010a

See Also

`run` | `GlobalSearch` | `MultiStart`

Topics

“Create Problem Structure” on page 4-4

CustomStartPointSet

Custom start points

Description

A CustomStartPointSet is an object wrapper of a matrix whose rows represent start points for MultiStart.

Creation

Syntax

```
tpoints = CustomStartPointSet(ptmatrix)
```

Description

`tpoints = CustomStartPointSet(ptmatrix)` generates a CustomStartPointSet object from the `ptmatrix` matrix. Each row of `ptmatrix` represents one start point.

Input Arguments

ptmatrix — Start points

matrix

Start points, specified as a matrix. Each row of `ptmatrix` represents one start point.

Example: `randn(40,3)` creates 40 start points of 3 dimensions.

Data Types: `double`

Properties

NumStartPoints — Number of start points

positive integer

This property is read-only.

Number of start points, specified as a positive integer. `NumStartPoints` is the number of rows in `ptmatrix`.

Example: `40`

Data Types: `double`

StartPointsDimension — Dimension of each start point

positive integer

This property is read-only.

Dimension of each start point, specified as a positive integer. `StartPointsDimension` is the number of columns in `ptmatrix`.

`StartPointsDimension` is the same as the number of elements in `problem.x0`, the problem structure you pass to `run`.

Example: 5

Data Types: `double`

Object Functions

`list` List start points

Examples

Create `CustomStartPointSet`

Create a `CustomStartPointSet` object with 64 three-dimensional points.

```
[x,y,z] = meshgrid(1:4);  
ptmatrix = [x(:),y(:),z(:)] + [10,20,30];  
tpoints = CustomStartPointSet(ptmatrix);
```

`tpoints` is the `ptmatrix` matrix contained in a `CustomStartPointSet` object.

Extract the original matrix from the `tpoints` object by using `list`.

```
tpts = list(tpoints);
```

Check that the `tpts` output is identical to `ptmatrix`.

```
isequal(ptmatrix,tpts)
```

```
ans = logical  
     1
```

Version History

Introduced in R2010a

See Also

`MultiStart` | `RandomStartPointSet` | `list`

Topics

“`CustomStartPointSet` Object for Start Points” on page 4-11

“Workflow for `GlobalSearch` and `MultiStart`” on page 4-3

ga

Find minimum of function using genetic algorithm

Syntax

```
x = ga(fun,nvars)
x = ga(fun,nvars,A,b)
x = ga(fun,nvars,A,b,Aeq,beq)
x = ga(fun,nvars,A,b,Aeq,beq,lb,ub)
x = ga(fun,nvars,A,b,Aeq,beq,lb,ub,nonlcon)
x = ga(fun,nvars,A,b,Aeq,beq,lb,ub,nonlcon,options)
x = ga(fun,nvars,A,b,Aeq,beq,lb,ub,nonlcon,intcon)
x = ga(fun,nvars,A,b,Aeq,beq,lb,ub,nonlcon,intcon,options)
x = ga(problem)
[x,fval] = ga(____)
[x,fval,exitflag,output] = ga(____)
[x,fval,exitflag,output,population,scores] = ga(____)
```

Description

$x = \text{ga}(\text{fun}, \text{nvars})$ finds a local unconstrained minimum, x , to the objective function, fun . nvars is the dimension (number of design variables) of fun .

Note “Passing Extra Parameters” explains how to pass extra parameters to the objective function and nonlinear constraint functions, if necessary.

$x = \text{ga}(\text{fun}, \text{nvars}, A, b)$ finds a local minimum x to fun , subject to the linear inequalities $A*x \leq b$. ga evaluates the matrix product $A*x$ as if x is transposed ($A*x'$).

$x = \text{ga}(\text{fun}, \text{nvars}, A, b, Aeq, beq)$ finds a local minimum x to fun , subject to the linear equalities $Aeq*x = beq$ and $A*x \leq b$. (Set $A=[]$ and $b=[]$ if no linear inequalities exist.) ga evaluates the matrix product $Aeq*x$ as if x is transposed ($Aeq*x'$).

$x = \text{ga}(\text{fun}, \text{nvars}, A, b, Aeq, beq, lb, ub)$ defines a set of lower and upper bounds on the design variables, x , so that a solution is found in the range $lb \leq x \leq ub$. (Set $Aeq=[]$ and $beq=[]$ if no linear equalities exist.)

$x = \text{ga}(\text{fun}, \text{nvars}, A, b, Aeq, beq, lb, ub, \text{nonlcon})$ subjects the minimization to the constraints defined in nonlcon . The function nonlcon accepts x and returns vectors C and Ceq , representing the nonlinear inequalities and equalities respectively. ga minimizes the fun such that $C(x) \leq 0$ and $Ceq(x) = 0$. (Set $lb=[]$ and $ub=[]$ if no bounds exist.)

$x = \text{ga}(\text{fun}, \text{nvars}, A, b, Aeq, beq, lb, ub, \text{nonlcon}, \text{options})$ minimizes with the default optimization parameters replaced by values in options . (Set $\text{nonlcon}=[]$ if no nonlinear constraints exist.) Create options using `optimoptions`.

$x = \text{ga}(\text{fun}, \text{nvars}, A, b, Aeq, beq, lb, ub, \text{nonlcon}, \text{intcon})$ or $x = \text{ga}(\text{fun}, \text{nvars}, A, b, Aeq, beq, lb, ub, \text{nonlcon}, \text{intcon}, \text{options})$ requires that the variables listed in intcon take integer values.

Note When there are integer constraints, `ga` does not accept nonlinear equality constraints, only nonlinear inequality constraints.

`x = ga(problem)` finds the minimum for `problem`, a structure described in `problem`.

`[x,fval] = ga(___)`, for any previous input arguments, also returns `fval`, the value of the fitness function at `x`.

`[x,fval,exitflag,output] = ga(___)` also returns `exitflag`, an integer identifying the reason the algorithm terminated, and `output`, a structure that contains output from each generation and other information about the performance of the algorithm.

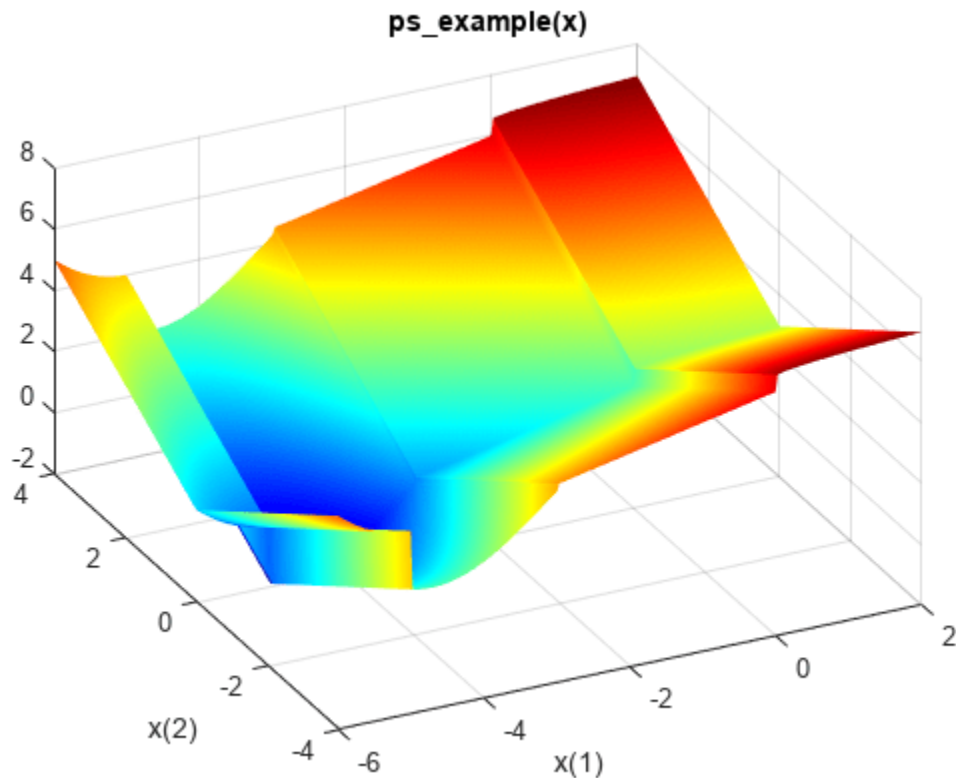
`[x,fval,exitflag,output,population,scores] = ga(___)` also returns a matrix `population`, whose rows are the final population, and a vector `scores`, the scores of the final population.

Examples

Optimize a Nonsmooth Function Using `ga`

The `ps_example.m` file is included when you run this example. Plot the function.

```
xi = linspace(-6,2,300);
yi = linspace(-4,4,300);
[X,Y] = meshgrid(xi,yi);
Z = ps_example([X(:),Y(:)]);
Z = reshape(Z,size(X));
surf(X,Y,Z,'MeshStyle','none')
colormap 'jet'
view(-26,43)
xlabel('x(1)')
ylabel('x(2)')
title('ps\_example(x)')
```



Find the minimum of this function using `ga`.

```
rng default % For reproducibility
x = ga(@ps_example,2)
```

Optimization terminated: average change in the fitness value less than options.FunctionTolerance

```
x = 1x2
```

```
   -4.6793   -0.0860
```

Minimize a Nonsmooth Function with Linear Constraints

Use the genetic algorithm to minimize the `ps_example` function on the region $x(1) + x(2) \geq 1$ and $x(2) \leq 5 + x(1)$. This function is included when you run this example.

First, convert the two inequality constraints to the matrix form $A*x \leq b$. In other words, get the x variables on the left-hand side of the inequality, and make both inequalities less than or equal:

$$-x(1) - x(2) \leq -1$$

$$-x(1) + x(2) \leq 5$$

```
A = [-1, -1;
     -1, 1];
b = [-1; 5];
```

Solve the constrained problem using `ga`.

```
rng default % For reproducibility
fun = @ps_example;
x = ga(fun, 2, A, b)
```

Optimization terminated: average change in the fitness value less than options.FunctionTolerance

```
x = 1×2
     1.0007    -0.0000
```

The constraints are satisfied to within the default value of the constraint tolerance, $1e-3$. To see this, compute $A*x' - b$, which should have negative components.

```
disp(A*x' - b)
    -0.0007
    -6.0007
```

Minimize a Nonsmooth Function with Linear Equality and Inequality Constraints

Use the genetic algorithm to minimize the `ps_example` function on the region $x(1) + x(2) \geq 1$ and $x(2) \leq 5 + x(1)$. This function is included when you run this example.

First, convert the two constraints to the matrix form $A*x \leq b$ and $Aeq*x = beq$. In other words, get the x variables on the left-hand side of the expressions, and make the inequality into less than or equal form:

```
-x(1) -x(2) <= -1
-x(1) + x(2) == 5
```

```
A = [-1 -1];
b = -1;
Aeq = [-1 1];
beq = 5;
```

Solve the constrained problem using `ga`.

```
rng default % For reproducibility
fun = @ps_example;
x = ga(fun, 2, A, b, Aeq, beq)
```

Optimization terminated: average change in the fitness value less than options.FunctionTolerance

```
x = 1×2
    -2.0005    2.9995
```

Check that the constraints are satisfied to within the default value of `ConstraintTolerance`, $1e-3$.

```
disp(A*x' - b)
    9.9999e-04
disp(Aeq*x' - beq)
    5.7738e-09
```

Optimize with Linear Constraints and Bounds

Use the genetic algorithm to minimize the `ps_example` function on the region $x(1) + x(2) \geq 1$ and $x(2) = 5 + x(1)$. The `ps_example` function is included when you run this example. In addition, set bounds $1 \leq x(1) \leq 6$ and $-3 \leq x(2) \leq 8$.

First, convert the two linear constraints to the matrix form $A*x \leq b$ and $Aeq*x = beq$. In other words, get the x variables on the left-hand side of the expressions, and make the inequality into less than or equal form:

$$-x(1) - x(2) \leq -1$$

$$-x(1) + x(2) = 5$$

```
A = [-1 -1];
b = -1;
Aeq = [-1 1];
beq = 5;
```

Set bounds `lb` and `ub`.

```
lb = [1 -3];
ub = [6 8];
```

Solve the constrained problem using `ga`.

```
rng default % For reproducibility
fun = @ps_example;
x = ga(fun,2,A,b,Aeq,beq,lb,ub)
```

Optimization terminated: average change in the fitness value less than options.FunctionTolerance

```
x = 1x2
    1.0000    6.0000
```

Check that the linear constraints are satisfied to within the default value of `ConstraintTolerance`, $1e-3$.

```
disp(A*x' - b)
   -6.0000
disp(Aeq*x' - beq)
   -3.4030e-08
```

Optimize with Nonlinear Constraints Using ga

Use the genetic algorithm to minimize the `ps_example` function on the region $2x_1^2 + x_2^2 \leq 3$ and $(x_1 + 1)^2 = (x_2/2)^4$. The `ps_example` function is included when you run this example.

To do so, use the function `ellipsecons.m` that returns the inequality constraint in the first output, `c`, and the equality constraint in the second output, `ceq`. The `ellipsecons` function is included when you run this example. Examine the `ellipsecons` code.

```
type ellipsecons
function [c,ceq] = ellipsecons(x)
c = 2*x(1)^2 + x(2)^2 - 3;
ceq = (x(1)+1)^2 - (x(2)/2)^4;
```

Include a function handle to `ellipsecons` as the `nonlcon` argument.

```
nonlcon = @ellipsecons;
fun = @ps_example;
rng default % For reproducibility
x = ga(fun,2,[],[],[],[],[],[],nonlcon)
```

Optimization finished: average change in the fitness value less than options.FunctionTolerance and

```
x = 1x2
    -0.9766    0.0362
```

Check that the nonlinear constraints are satisfied at `x`. The constraints are satisfied when `c ≤ 0` and `ceq = 0` to within the default value of `ConstraintTolerance`, `1e-3`.

```
[c,ceq] = nonlcon(x)
c = -1.0911
ceq = 5.4645e-04
```

Minimize with Nondefault Options

Use the genetic algorithm to minimize the `ps_example` function on the region $x(1) + x(2) \geq 1$ and $x(2) == 5 + x(1)$ using a constraint tolerance that is smaller than the default. The `ps_example` function is included when you run this example.

First, convert the two constraints to the matrix form $A*x \leq b$ and $Aeq*x = beq$. In other words, get the `x` variables on the left-hand side of the expressions, and make the inequality into less than or equal form:

```
-x(1) -x(2) <= -1
-x(1) + x(2) == 5
```



```
A = [-1 -1];
b = -1;
Aeq = [-1 1];
beq = 5;
```

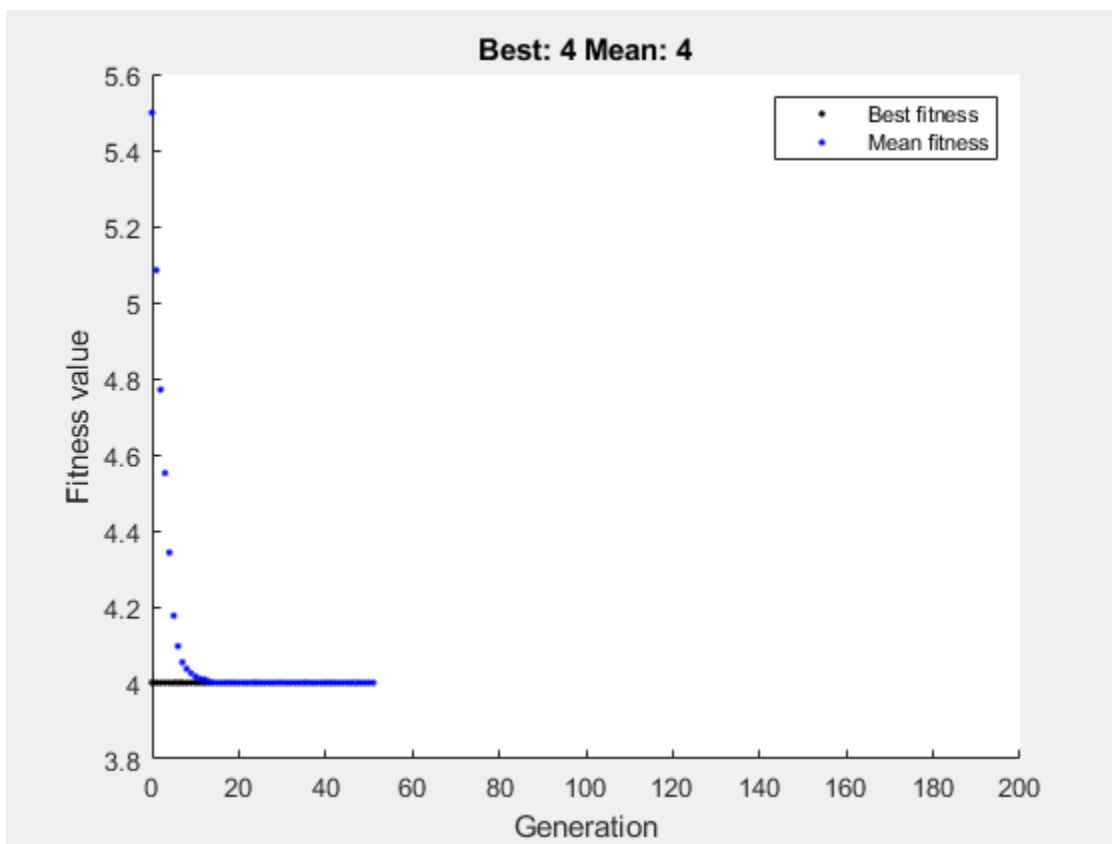
To obtain a more accurate solution, set a constraint tolerance of $1e-6$. And to monitor the solver progress, set a plot function.

```
options = optimoptions('ga','ConstraintTolerance',1e-6,'PlotFcn', @gaplotbestf);
```

Solve the minimization problem.

```
rng default % For reproducibility
fun = @ps_example;
x = ga(fun,2,A,b,Aeq,beq,[],[],[],options)
```

Optimization terminated: average change in the fitness value less than options.FunctionTolerance



```
x = 1x2
    -2.0000    3.0000
```

Check that the linear constraints are satisfied to within $1e-6$.

```
disp(A*x' - b)
    9.9838e-07
```

```
disp(Aeq*x' - beq)
-1.4900e-08
```

Minimize a Nonlinear Function with Integer Constraints

Use the genetic algorithm to minimize the `ps_example` function subject to the constraint that `x(1)` is an integer. This function is included when you run this example.

```
intcon = 1;
rng default % For reproducibility
fun = @ps_example;
A = [];
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];
nonlcon = [];
x = ga(fun,2,A,b,Aeq,beq,lb,ub,nonlcon,intcon)
```

Optimization terminated: average change in the penalty fitness value less than options.FunctionTol and constraint violation is less than options.ConstraintTolerance.

```
x = 1×2
-5.0000 -0.0834
```

Obtain the Solution and Function Value

Use the genetic algorithm to minimize an integer-constrained nonlinear problem. Obtain both the location of the minimum and the minimum function value. The objective function, `ps_example`, is included when you run this example.

```
intcon = 1;
rng default % For reproducibility
fun = @ps_example;
A = [];
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];
nonlcon = [];
[x,fval] = ga(fun,2,A,b,Aeq,beq,lb,ub,nonlcon,intcon)
```

Optimization terminated: average change in the penalty fitness value less than options.FunctionTol and constraint violation is less than options.ConstraintTolerance.

```
x = 1×2
-5.0000 -0.0834
```

```
fval = -1.8344
```

Compare this result to the solution of the problem with no constraints.

```
[x,fval] = ga(fun,2)
```

```
Optimization terminated: average change in the fitness value less than options.FunctionTolerance
```

```
x = 1×2
```

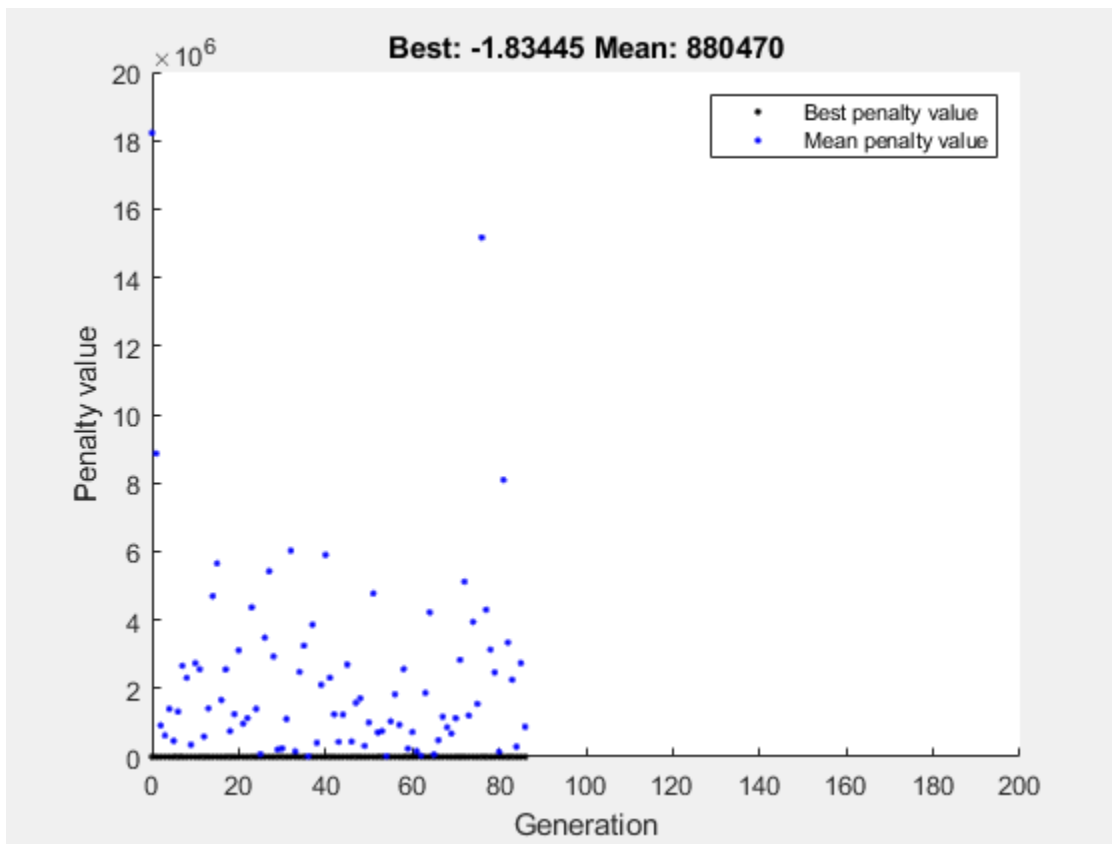
```
    -4.6906    -0.0078
```

```
fval = -1.9918
```

Obtain Diagnostic Information

Use the genetic algorithm to minimize the `ps_example` function constrained to have `x(1)` integer-valued. The `ps_example` function is included when you run this example. To understand the reason the solver stopped and how `ga` searched for a minimum, obtain the `exitflag` and `output` results. Also, plot the minimum observed objective function value as the solver progresses.

```
intcon = 1;
rng default % For reproducibility
fun = @ps_example;
A = [];
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];
nonlcon = [];
options = optimoptions('ga','PlotFcn', @gaplotbestf);
[x,fval,exitflag,output] = ga(fun,2,A,b,Aeq,beq,lb,ub,nonlcon,intcon,options)
```



Optimization terminated: average change in the penalty fitness value less than options.FunctionTolerance and constraint violation is less than options.ConstraintTolerance.

`x = 1x2`

```
-5.0000    -0.0834
```

`fval = -1.8344`

`exitflag = 1`

`output = struct with fields:`

```
    problemtype: 'integerconstraints'
```

```
    rngstate: [1x1 struct]
```

```
    generations: 86
```

```
    funccount: 3311
```

```
    message: 'Optimization terminated: average change in the penalty fitness value less than options.FunctionTolerance and constraint violation is less than options.ConstraintTolerance.'
```

```
    maxconstraint: 0
```

```
    hybridflag: []
```

Obtain Final Population and Scores

Use the genetic algorithm to minimize the `ps_example` function constrained to have `x(1)` integer-valued. The `ps_example` function is included when you run this example. Obtain all outputs, including the final population and vector of scores.

```
intcon = 1;
rng default % For reproducibility
fun = @ps_example;
A = [];
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];
nonlcon = [];
[x,fval,exitflag,output,population,scores] = ga(fun,2,A,b,Aeq,beq,lb,ub,nonlcon,intcon);
```

Optimization terminated: average change in the penalty fitness value less than options.FunctionTolerance and constraint violation is less than options.ConstraintTolerance.

Examine the first 10 members of the final population and their corresponding scores. Notice that `x(1)` is integer-valued for all these population members. The integer `ga` algorithm generates only integer-feasible populations.

```
disp(population(1:10,:))
```

```
1.0e+03 *
-0.0050 -0.0001
-0.0050 -0.0001
-1.6420  0.0027
-1.5070  0.0010
-0.4540  0.0104
-0.2530 -0.0011
-0.1210 -0.0003
-0.1040  0.1314
-0.0140 -0.0010
 0.0160 -0.0002
```

```
disp(scores(1:10))
```

```
1.0e+06 *
-0.0000
-0.0000
 2.6798
 2.2560
 0.2016
 0.0615
 0.0135
 0.0099
 0.0001
 0.0000
```

Input Arguments

fun — Objective function

function handle | function name

Objective function, specified as a function handle or function name. Write the objective function to accept a row vector of length `nvars` and return a scalar value.

When the 'UseVectorized' option is true, write `fun` to accept a `pop-by-nvars` matrix, where `pop` is the current population size. In this case, `fun` returns a vector the same length as `pop` containing the fitness function values. Ensure that `fun` does not assume any particular size for `pop`, since `ga` can pass a single member of a population even in a vectorized calculation.

Example: `fun = @(x)(x-[4,2]).^2`

Data Types: `char` | `function_handle` | `string`

nvars — Number of variables

positive integer

Number of variables, specified as a positive integer. The solver passes row vectors of length `nvars` to `fun`.

Example: 4

Data Types: `double`

A — Linear inequality constraints

real matrix

Linear inequality constraints, specified as a real matrix. `A` is an `M-by-nvars` matrix, where `M` is the number of inequalities.

`A` encodes the `M` linear inequalities

$A \cdot x \leq b$,

where `x` is the column vector of `nvars` variables `x(:)`, and `b` is a column vector with `M` elements.

For example, to specify

$x_1 + 2x_2 \leq 10$
 $3x_1 + 4x_2 \leq 20$
 $5x_1 + 6x_2 \leq 30$,

give these constraints:

`A = [1,2;3,4;5,6];`
`b = [10;20;30];`

Example: To specify that the control variables sum to 1 or less, give the constraints `A = ones(1,N)` and `b = 1`.

Data Types: `double`

b — Linear inequality constraints

real vector

Linear inequality constraints, specified as a real vector. **b** is an M-element vector related to the A matrix. If you pass **b** as a row vector, solvers internally convert **b** to the column vector **b(:)**.

b encodes the M linear inequalities

$$A*x \leq b,$$

where **x** is the column vector of N variables **x(:)**, and **A** is a matrix of size M-by-N.

For example, to specify

$$\begin{aligned} x_1 + 2x_2 &\leq 10 \\ 3x_1 + 4x_2 &\leq 20 \\ 5x_1 + 6x_2 &\leq 30, \end{aligned}$$

give these constraints:

$$\begin{aligned} A &= [1,2;3,4;5,6]; \\ b &= [10;20;30]; \end{aligned}$$

Example: To specify that the control variables sum to 1 or less, give the constraints **A = ones(1,N)** and **b = 1**.

Data Types: double

Aeq — Linear equality constraints

real matrix

Linear equality constraints, specified as a real matrix. **Aeq** is an Me-by-nvars matrix, where Me is the number of equalities.

Aeq encodes the Me linear equalities

$$Aeq*x = beq,$$

where **x** is the column vector of N variables **x(:)**, and **beq** is a column vector with Me elements.

For example, to specify

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 10 \\ 2x_1 + 4x_2 + x_3 &= 20, \end{aligned}$$

give these constraints:

$$\begin{aligned} Aeq &= [1,2,3;2,4,1]; \\ beq &= [10;20]; \end{aligned}$$

Example: To specify that the control variables sum to 1, give the constraints **Aeq = ones(1,N)** and **beq = 1**.

Data Types: double

beq — Linear equality constraints

real vector

Linear equality constraints, specified as a real vector. **beq** is an Me-element vector related to the **Aeq** matrix. If you pass **beq** as a row vector, solvers internally convert **beq** to the column vector **beq(:)**.

`beq` encodes the `Me` linear equalities

$$\text{Aeq} * x = \text{beq},$$

where x is the column vector of N variables $x(:)$, and `Aeq` is a matrix of size `Meq`-by- N .

For example, to specify

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 10 \\ 2x_1 + 4x_2 + x_3 &= 20, \end{aligned}$$

give these constraints:

$$\begin{aligned} \text{Aeq} &= [1, 2, 3; 2, 4, 1]; \\ \text{beq} &= [10; 20]; \end{aligned}$$

Example: To specify that the control variables sum to 1, give the constraints `Aeq = ones(1, N)` and `beq = 1`.

Data Types: `double`

lb — Lower bounds

`[]` (default) | real vector or array

Lower bounds, specified as a real vector or array of doubles. `lb` represents the lower bounds element-wise in $\text{lb} \leq x \leq \text{ub}$.

Internally, `ga` converts an array `lb` to the vector `lb(:)`.

Example: `lb = [0; -Inf; 4]` means $x(1) \geq 0$, $x(3) \geq 4$.

Data Types: `double`

ub — Upper bounds

`[]` (default) | real vector or array

Upper bounds, specified as a real vector or array of doubles. `ub` represents the upper bounds element-wise in $\text{lb} \leq x \leq \text{ub}$.

Internally, `ga` converts an array `ub` to the vector `ub(:)`.

Example: `ub = [Inf; 4; 10]` means $x(2) \leq 4$, $x(3) \leq 10$.

Data Types: `double`

nonlcon — Nonlinear constraints

function handle | function name

Nonlinear constraints, specified as a function handle or function name. `nonlcon` is a function that accepts a vector or array x and returns two arrays, $c(x)$ and $\text{ceq}(x)$.

- $c(x)$ is the array of nonlinear inequality constraints at x . `ga` attempts to satisfy

$$c(x) \leq 0$$

for all entries of c .

- $\text{ceq}(x)$ is the array of nonlinear equality constraints at x . `ga` attempts to satisfy

$$\text{ceq}(x) = 0$$

for all entries of `ceq`.

For example,

```
x = ga(@myfun,4,A,b,Aeq,beq,lb,ub,@mycon)
```

where `mycon` is a MATLAB function such as

```
function [c,ceq] = mycon(x)
c = ...      % Compute nonlinear inequalities at x.
ceq = ...    % Compute nonlinear equalities at x.
```

For more information, see “Nonlinear Constraints”.

To learn how to use vectorized constraints, see “Vectorized Constraints” on page 2-7.

Note `ga` does not enforce nonlinear constraints to be satisfied when the `PopulationType` option is set to `'bitString'` or `'custom'`.

If `intcon` is not empty, the second output of `nonlcon` (`ceq`) must be an empty entry (`[]`).

For information on how `ga` uses `nonlcon`, see “Nonlinear Constraint Solver Algorithms for Genetic Algorithm” on page 8-56.

Data Types: `char` | `function_handle` | `string`

options — Optimization options

output of `optimoptions` | structure

Optimization options, specified as the output of `optimoptions` or a structure.

`optimoptions` hides the options listed in *italics*. See “Options that `optimoptions` Hides” on page 17-65.

- Values in `{}` denote the default value.
- `{}`* represents the default when there are linear constraints, and for `MutationFcn` also when there are bounds.
- **I*** indicates default for integer constraints, or indicates special considerations for integer constraints.
- **NM** indicates that the option does not apply to `gamultiobj`.

Options for `ga` and `gamultiobj`

Option	Description	Values
ConstraintTolerance	Determines the feasibility with respect to nonlinear constraints. Also, <code>max(sqrt(eps), ConstraintTolerance)</code> determines feasibility with respect to linear constraints. For an options structure, use <code>TolCon</code> .	Positive scalar <code>{1e-3}</code>
CreationFcn	Function that creates the initial population. Specify as a name of a built-in creation function or a function handle. See “Population Options” on page 17-26.	<code>{'gacreationuniform'} {'gacreationlinearfeasible'}* 'gacreationnonlinearfeasible' {'gacreationuniformint'}I*</code> for <code>ga</code> <code>{'gacreationsobol'}I*</code> for <code>gamultiobj</code> Custom creation function on page 17-26
CrossoverFcn	Function that the algorithm uses to create crossover children. Specify as a name of a built-in crossover function or a function handle. See “Crossover Options” on page 17-34.	<code>{'crossoverscattered'}</code> for <code>ga</code> , <code>{'crossoverintermediate'}*</code> for <code>gamultiobj</code> <code>{'crossoverlaplace'}I*</code> <code>'crossoverheuristic'</code> <code>'crossoversinglepoint'</code> <code>'crossovertwoopt'</code> <code>'crossoverarithmetic'</code> Custom crossover function on page 17-34
CrossoverFraction	The fraction of the population at the next generation, not including elite children, that the crossover function creates.	Positive scalar <code>{0.8}</code>
Display	Level of display.	<code>'off'</code> <code>'iter'</code> <code>'diagnose'</code> <code>{'final'}</code>
DistanceMeasureFcn	Function that computes the distance measure of individuals. Specify as a name of a built-in distance measure function or a function handle. The value applies to the decision variable or design space (genotype) or to function space (phenotype). The default <code>'distancecrowding'</code> is in function space (phenotype). For <code>gamultiobj</code> only. See “Multiobjective Options” on page 17-38. For an options structure, use a function handle, not a name.	<code>{'distancecrowding'}</code> means the same as <code>{@distancecrowding, 'phenotype'}</code> <code>{@distancecrowding, 'genotype'}</code> Custom distance function on page 17-38

Option	Description	Values
EliteCount	NM Positive integer specifying how many individuals in the current generation are guaranteed to survive to the next generation. Not used in <code>gamultiobj</code> .	Positive integer $\{\text{ceil}(0.05 * \text{PopulationSize})\}$ $\{0.05 * (\text{default PopulationSize})\}$ for mixed-integer problems
FitnessLimit	NM If the fitness function attains the value of <code>FitnessLimit</code> , the algorithm halts.	Scalar $\{-\text{Inf}\}$
FitnessScalingFcn	Function that scales the values of the fitness function. Specify as a name of a built-in scaling function or a function handle. Option unavailable for <code>gamultiobj</code> .	<code>'fitscalingrank'</code> <code>'fitscalingshiftlinear'</code> <code>'fitscalingprop'</code> <code>'fitscalingtop'</code> Custom fitness scaling function on page 17-29
FunctionTolerance	The algorithm stops if the average relative change in the best fitness function value over <code>MaxStallGenerations</code> generations is less than or equal to <code>FunctionTolerance</code> . If <code>StallTest</code> is <code>'geometricWeighted'</code> , then the algorithm stops if the weighted average relative change is less than or equal to <code>FunctionTolerance</code> . For <code>gamultiobj</code> , the algorithm stops when the geometric average of the relative change in value of the spread over <code>options.MaxStallGenerations</code> generations is less than <code>options.FunctionTolerance</code> , and the final spread is less than the mean spread over the past <code>options.MaxStallGenerations</code> generations. See “ <code>gamultiobj</code> Algorithm” on page 14-5. For an options structure, use <code>TolFun</code> .	Positive scalar $\{1e-6\}$ for <code>ga</code> , $\{1e-4\}$ for <code>gamultiobj</code>
HybridFcn	I* Function that continues the optimization after <code>ga</code> terminates. Specify as a name or a function handle. Alternatively, a cell array specifying the hybrid function and its options. See “ <code>ga</code> Hybrid Function” on page 17-39. For <code>gamultiobj</code> , the only hybrid function is <code>@fgoalattain</code> . See “ <code>gamultiobj</code> Hybrid Function” on page 17-40. When the problem has integer constraints, you cannot use a hybrid function. See “When to Use a Hybrid Function” on page 8-116.	Function name or handle <code>'fminsearch'</code> <code>'patternsearch'</code> <code>'fminunc'</code> <code>'fmincon'</code> <code>{[]}</code> or 1-by-2 cell array <code>{@solver, hybridoptions}</code> , where <code>solver</code> = <code>fminsearch</code> , <code>patternsearch</code> , <code>fminunc</code> , or <code>fmincon</code> <code>{[]}</code>

Option	Description	Values
<i>InitialPenalty</i>	NM I* Initial value of the penalty parameter	Positive scalar {10}
<i>InitialPopulationMatrix</i>	Initial population used to seed the genetic algorithm. Has up to <i>PopulationSize</i> rows and <i>N</i> columns, where <i>N</i> is the number of variables. You can pass a partial population, meaning one with fewer than <i>PopulationSize</i> rows. In that case, the genetic algorithm uses <i>CreationFcn</i> to generate the remaining population members. See “Population Options” on page 17-26. For an options structure, use <i>InitialPopulation</i> .	Matrix {[]}
<i>InitialPopulationRange</i>	Matrix or vector specifying the range of the individuals in the initial population. Applies to <i>gacreationuniform</i> creation function. <i>ga</i> shifts and scales the default initial range to match any finite bounds. For an options structure, use <i>PopInitRange</i> .	Matrix or vector {[-10; 10]} for unbounded components, {[-1e4+1; 1e4+1]} for unbounded components of integer-constrained problems, {[lb;ub]} for bounded components, with the default range modified to match one-sided bounds
<i>InitialScoresMatrix</i>	Initial scores used to determine fitness. Has up to <i>PopulationSize</i> rows and <i>Nf</i> columns, where <i>Nf</i> is the number of fitness functions (1 for <i>ga</i> , greater than 1 for <i>gamultiobj</i>). You can pass a partial scores matrix, meaning one with fewer than <i>PopulationSize</i> rows. In that case, the solver fills in the scores when it evaluates the fitness functions. For an options structure, use <i>InitialScores</i> .	Column vector for single objective matrix for multiobjective {[]}
<i>MaxGenerations</i>	Maximum number of iterations before the algorithm halts. For an options structure, use <i>Generations</i> .	Positive integer {100*numberOfVariables} for <i>ga</i> , {200*numberOfVariables} for <i>gamultiobj</i>

Option	Description	Values
MaxStallGenerations	<p>The algorithm stops if the average relative change in the best fitness function value over MaxStallGenerations generations is less than or equal to FunctionTolerance. If StallTest is 'geometricWeighted', then the algorithm stops if the weighted average relative change is less than or equal to FunctionTolerance.</p> <p>For gamultiobj, the algorithm stops when the geometric average of the relative change in value of the spread over options.MaxStallGenerations generations is less than options.FunctionTolerance, and the final spread is less than the mean spread over the past options.MaxStallGenerations generations. See “gamultiobj Algorithm” on page 14-5.</p> <p>For an options structure, use StallGenLimit.</p>	Positive integer {50} for ga, {100} for gamultiobj
MaxStallTime	<p>NM The algorithm stops if there is no improvement in the objective function for MaxStallTime seconds, as measured by tic and toc.</p> <p>For an options structure, use StallTimeLimit.</p>	Positive scalar {Inf}
MaxTime	<p>The algorithm stops after running for MaxTime seconds, as measured by tic and toc. This limit is enforced after each iteration, so ga can exceed the limit when an iteration takes substantial time.</p> <p>For an options structure, use TimeLimit.</p>	Positive scalar {Inf}
MigrationDirection	Direction of migration. See “Migration Options” on page 17-37.	'both' {'forward'}
MigrationFraction	Scalar from 0 through 1 specifying the fraction of individuals in each subpopulation that migrates to a different subpopulation. See “Migration Options” on page 17-37.	Scalar {0.2}
MigrationInterval	Positive integer specifying the number of generations that take place between migrations of individuals between subpopulations. See “Migration Options” on page 17-37.	Positive integer {20}

Option	Description	Values
MutationFcn	Function that produces mutation children. Specify as a name of a built-in mutation function or a function handle. See “Mutation Options” on page 17-31.	{'mutationgaussian'} for ga without constraints {'mutationadaptfeasible'}* for gamultiobj and for ga with constraints {'mutationpower'}I* 'mutationpositivebasis' 'mutationuniform' Custom mutation function on page 17-31
NonlinearConstraintAlgorithm	Nonlinear constraint algorithm. See “Nonlinear Constraint Solver Algorithms for Genetic Algorithm” on page 8-56. Option unchangeable for gamultiobj. For an options structure, use NonlinConAlgorithm.	{'auglag'} for ga, {'penalty'} for gamultiobj
OutputFcn	Functions that ga calls at each iteration. Specify as a function handle or a cell array of function handles. See “Output Function Options” on page 17-41. For an options structure, use OutputFcns.	Function handle or cell array of function handles {[]}
ParetoFraction	Scalar from 0 through 1 specifying the fraction of individuals to keep on the first Pareto front while the solver selects individuals from higher fronts, for gamultiobj only. See “Multiobjective Options” on page 17-38.	Scalar {0.35}
PenaltyFactor	NM I* Penalty update parameter.	Positive scalar {100}
PlotFcn	Function that plots data computed by the algorithm. Specify as a name of a built-in plot function, a function handle, or a cell array of built-in names or function handles. See “Plot Options” on page 17-23. For an options structure, use PlotFcns.	ga or gamultiobj: {[]} 'gaplotdistance' 'gaplotgenealogy' 'gaplotselection' 'gaplotscorediversity' 'gaplotscores' 'gaplotstopping' 'gaplotmaxconstr' Custom plot function on page 17-23 ga only: 'gaplotbestf' 'gaplotbestindiv' 'gaplotexpectation' 'gaplotrange' gamultiobj only: 'gaplotpareto' 'gaplotparetodistance' 'gaplotrankhist' 'gaplotspread'

Option	Description	Values
<i>PlotInterval</i>	Positive integer specifying the number of generations between consecutive calls to the plot functions.	Positive integer {1}
PopulationSize	Size of the population.	Positive integer {50} when numberOfVariables <= 5, {200} otherwise {min(max(10*nvars,40),100)} for mixed-integer problems
PopulationType	Data type of the population. Must be 'doubleVector' for mixed-integer problems.	'bitstring' 'custom' {'doubleVector'} ga ignores all constraints when PopulationType is set to 'bitString' or 'custom'. See "Population Options" on page 17-26.
SelectionFcn	Function that selects parents of crossover and mutation children. Specify as a name of a built-in selection function or a function handle. gamultiobj uses only 'selectiontournament'.	{'selectionstochunif'} for ga, {'selectiontournament'} for gamultiobj 'selectionremainder' 'selectionuniform' 'selectionroulette' Custom selection function on page 17-30
StallTest	NM Stopping test type.	'geometricWeighted' {'averageChange'}
UseParallel	Compute fitness and nonlinear constraint functions in parallel. See "Vectorize and Parallel Options (User Function Evaluation)" on page 17-43 and "How to Use Parallel Processing in Global Optimization Toolbox" on page 16-11.	true {false}
UseVectorized	Specifies whether functions are vectorized. See "Vectorize and Parallel Options (User Function Evaluation)" on page 17-43 and "Vectorize the Fitness Function" on page 8-103. For an options structure, use Vectorized with the values 'on' or 'off'.	true {false}

Example: `optimoptions('ga','PlotFcn',@gaplotbestf)`

intcon – Integer variables

vector of positive integers

Integer variables, specified as a vector of positive integers taking values from 1 to nvars. Each value in intcon represents an x component that is integer-valued.

Note When `intcon` is nonempty, `nonlcon` must return empty for `ceq`. For more information on integer programming, see “Mixed Integer ga Optimization” on page 8-40.

Example: To specify that the even entries in `x` are integer-valued, set `intcon` to `2:2:nvars`

Data Types: `double`

problem — Problem description

structure

Problem description, specified as a structure containing these fields.

<code>fitnessfcn</code>	Fitness functions
<code>nvars</code>	Number of design variables
<code>Aineq</code>	A matrix for linear inequality constraints
<code>Bineq</code>	b vector for linear inequality constraints
<code>Aeq</code>	Aeq matrix for linear equality constraints
<code>Beq</code>	beq vector for linear equality constraints
<code>lb</code>	Lower bound on <code>x</code>
<code>ub</code>	Upper bound on <code>x</code>
<code>nonlcon</code>	Nonlinear constraint functions
<code>intcon</code>	Indices of integer variables
<code>rngstate</code>	Field to reset the state of the random number generator
<code>solver</code>	'ga'
<code>options</code>	Options created using <code>optimoptions</code> or an options structure

You must specify the fields `fitnessfcn`, `nvars`, and `options`. The remainder are optional for `ga`.

Data Types: `struct`

Output Arguments

x — Solution

real vector

Solution, returned as a real vector. `x` is the best point that `ga` located during its iterations.

fval — Objective function value at the solution

real number

Objective function value at the solution, returned as a real number. Generally, `fval = fun(x)`.

exitflag — Reason `ga` stopped

integer

Reason that `ga` stopped, returned as an integer.

Exit Flag	Meaning
1	Without nonlinear constraints — Average cumulative change in value of the fitness function over <code>MaxStallGenerations</code> generations is less than <code>FunctionTolerance</code> , and the constraint violation is less than <code>ConstraintTolerance</code> . With nonlinear constraints — Magnitude of the complementarity measure (see “Complementarity Measure” on page 18-34) is less than $\sqrt{\text{ConstraintTolerance}}$, the subproblem is solved using a tolerance less than <code>FunctionTolerance</code> , and the constraint violation is less than <code>ConstraintTolerance</code> .
3	Value of the fitness function did not change in <code>MaxStallGenerations</code> generations and the constraint violation is less than <code>ConstraintTolerance</code> .
4	Magnitude of step smaller than machine precision and the constraint violation is less than <code>ConstraintTolerance</code> .
5	Minimum fitness limit <code>FitnessLimit</code> reached and the constraint violation is less than <code>ConstraintTolerance</code> .
0	Maximum number of generations <code>MaxGenerations</code> exceeded.
-1	Optimization terminated by an output function or plot function.
-2	No feasible point found.
-4	Stall time limit <code>MaxStallTime</code> exceeded.
-5	Time limit <code>MaxTime</code> exceeded.

When there are integer constraints, `ga` uses the penalty fitness value instead of the fitness value for stopping criteria.

output — Information about the optimization process

structure

Information about the optimization process, returned as a structure with these fields:

- `problemtype` — Problem type, one of:
 - 'unconstrained'
 - 'boundconstraints'
 - 'linearconstraints'
 - 'nonlinearconstr'
 - 'integerconstraints'
- `rngstate` — State of the MATLAB random number generator, just before the algorithm started. You can use the values in `rngstate` to reproduce the output of `ga`. See “Reproduce Results” on page 8-67.
- `generations` — Number of generations computed.
- `funccount` — Number of evaluations of the fitness function.
- `message` — Reason the algorithm terminated.
- `maxconstraint` — Maximum constraint violation, if any.
- `hybridflag` — Exit flag from the hybrid function. Relates to the `HybridFcn` options. Not applicable to `gamultiobj`.

population — Final population

matrix

Final population, returned as a PopulationSize-by-nvars matrix. The rows of population are the individuals.

scores — Final scores

column vector

Final scores, returned as a column vector.

- For non-integer problems, the final scores are the fitness function values of the rows of population.
- For integer problems, the final scores are the penalty fitness values of the population members. See “Integer ga Algorithm” on page 8-45.

More About**Complementarity Measure**

In the Augmented Lagrangian nonlinear constraint solver, the complementarity measure is the norm of the vector whose elements are $c_i\lambda_i$, where c_i is the nonlinear inequality constraint violation, and λ_i is the corresponding Lagrange multiplier. See “Augmented Lagrangian Genetic Algorithm” on page 8-56.

Tips

- To write a function with additional parameters to the independent variables that can be called by `ga`, see “Passing Extra Parameters”.
- For problems that use the population type `Double Vector` (the default), `ga` does not accept functions whose inputs are of type `complex`. To solve problems involving complex data, write your functions so that they accept real vectors, by separating the real and imaginary parts.

Algorithms

For a description of the genetic algorithm, see “How the Genetic Algorithm Works” on page 8-15.

For a description of the mixed integer programming algorithm, see “Integer ga Algorithm” on page 8-45.

For a description of the nonlinear constraint algorithms, see “Nonlinear Constraint Solver Algorithms for Genetic Algorithm” on page 8-56.

Alternative Functionality**App**

The **Optimize** Live Editor task provides a visual interface for `ga`.

Version History

Introduced before R2006a

R2019b: ga Performs Fewer Fitness Function Evaluations

Behavior changed in R2019b

When the fitness function is deterministic, `ga` does not reevaluate the fitness function on elite (current best) individuals. You can control this behavior by accessing the new `state.EvalElites` field and modifying it in a custom output function or custom plot function. Similarly, when the initial population has duplicate members, `ga` evaluates each unique member only once. You can control this behavior in a custom output function or custom plot function by accessing and modifying the new `state.HaveDuplicates` field. For details, see “Custom Output Function for Genetic Algorithm” on page 8-105 or Custom Plot Function on page 8-59.

For details about the two new fields, see “The State Structure” on page 17-25.

References

- [1] Goldberg, David E., *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison-Wesley, 1989.
- [2] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. “A Globally Convergent Augmented Lagrangian Algorithm for Optimization with General Constraints and Simple Bounds”, *SIAM Journal on Numerical Analysis*, Volume 28, Number 2, pages 545-572, 1991.
- [3] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. “A Globally Convergent Augmented Lagrangian Barrier Algorithm for Optimization with General Inequality Constraints and Simple Bounds”, *Mathematics of Computation*, Volume 66, Number 217, pages 261-288, 1997.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set the 'UseParallel' option to `true`.

```
options = optimoptions('solvername','UseParallel',true)
```

For more information, see “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11.

See Also

`gamultiobj` | `optimoptions` | `particleswarm` | `patternsearch` | **Optimize**

Topics

“Genetic Algorithm”
 “Get Started with Global Optimization Toolbox”
 “Solver-Based Optimization Problem Setup”

gamultiobj

Find Pareto front of multiple fitness functions using genetic algorithm

Syntax

```
x = gamultiobj(fun,nvars)
x = gamultiobj(fun,nvars,A,b)
x = gamultiobj(fun,nvars,A,b,Aeq,beq)
x = gamultiobj(fun,nvars,A,b,Aeq,beq,lb,ub)
x = gamultiobj(fun,nvars,A,b,Aeq,beq,lb,ub,nonlcon)
x = gamultiobj(fun,nvars,A,b,Aeq,beq,lb,ub,options)
x = gamultiobj(fun,nvars,A,b,Aeq,beq,lb,ub,nonlcon,options)
x = gamultiobj(fun,nvars,A,b,Aeq,beq,lb,ub,nonlcon,intcon)
x = gamultiobj(fun,nvars,A,b,Aeq,beq,lb,ub,nonlcon,intcon,options)
x = gamultiobj(problem)
[x,fval] = gamultiobj(____)
[x,fval,exitflag,output] = gamultiobj(____)
[x,fval,exitflag,output,population,scores] = gamultiobj(____)
```

Description

`x = gamultiobj(fun,nvars)` finds x on the “Pareto Front” on page 18-58 of the objective functions defined in `fun`. `nvars` is the dimension of the optimization problem (number of decision variables). The solution x is local, which means it might not be on the global Pareto front.

Note “Passing Extra Parameters” explains how to pass extra parameters to the objective function and nonlinear constraint functions, if necessary.

`x = gamultiobj(fun,nvars,A,b)` finds a local Pareto set x subject to the linear inequalities $A * x \leq b$. See “Linear Inequality Constraints”. `gamultiobj` supports linear constraints only for the default `PopulationType` option ('doubleVector').

`x = gamultiobj(fun,nvars,A,b,Aeq,beq)` finds a local Pareto set x subject to the linear equalities $Aeq * x = beq$ and the linear inequalities $A * x \leq b$, see “Linear Equality Constraints”. (Set $A = []$ and $b = []$ if no inequalities exist.) `gamultiobj` supports linear constraints only for the default `PopulationType` option ('doubleVector').

`x = gamultiobj(fun,nvars,A,b,Aeq,beq,lb,ub)` defines a set of lower and upper bounds on the design variables x so that a local Pareto set is found in the range $lb \leq x \leq ub$, see “Bound Constraints”. Use empty matrices for `Aeq` and `beq` if no linear equality constraints exist. `gamultiobj` supports bound constraints only for the default `PopulationType` option ('doubleVector').

`x = gamultiobj(fun,nvars,A,b,Aeq,beq,lb,ub,nonlcon)` finds a Pareto set subject to the constraints defined in `nonlcon`. The function `nonlcon` accepts x and returns vectors c and ceq , representing the nonlinear inequalities and equalities respectively. `gamultiobj` minimizes `fun` such that $c(x) \leq 0$ and $ceq(x) = 0$. (Set $lb = []$ and $ub = []$ if no bounds exist.) `gamultiobj` supports nonlinear constraints only for the default `PopulationType` option ('doubleVector').

`x = gamultiobj(fun,nvars,A,b,Aeq,beq,lb,ub,options)` or `x = gamultiobj(fun,nvars,A,b,Aeq,beq,lb,ub,nonlcon,options)` finds a Pareto set `x` with the default optimization parameters replaced by values in `options`. Create `options` using `optimoptions` (recommended) or a structure.

`x = gamultiobj(fun,nvars,A,b,Aeq,beq,lb,ub,nonlcon,intcon)` or `x = gamultiobj(fun,nvars,A,b,Aeq,beq,lb,ub,nonlcon,intcon,options)` requires that the variables listed in `intcon` take integer values.

Note When there are integer constraints, `gamultiobj` does not accept nonlinear equality constraints, only nonlinear inequality constraints.

`x = gamultiobj(problem)` finds the Pareto set for `problem`, where `problem` is a structure described in `problem`.

`[x,fval] = gamultiobj(____)`, for any input variables, returns a matrix `fval`, the value of all the fitness functions defined in `fun` for all the solutions in `x`. `fval` has `nf` columns, where `nf` is the number of objectives, and has the same number of rows as `x`.

`[x,fval,exitflag,output] = gamultiobj(____)` returns `exitflag`, an integer identifying the reason the algorithm stopped, and `output`, a structure that contains information about the optimization process.

`[x,fval,exitflag,output,population,scores] = gamultiobj(____)` returns `population`, whose rows are the final population, and `scores`, the scores of the final population.

Examples

Simple Multiobjective Problem

Find the Pareto front for a simple multiobjective problem. There are two objectives and two decision variables `x`.

```
fitnessfcn = @(x)[norm(x)^2,0.5*norm(x(:)-[2;-1])^2+2];
```

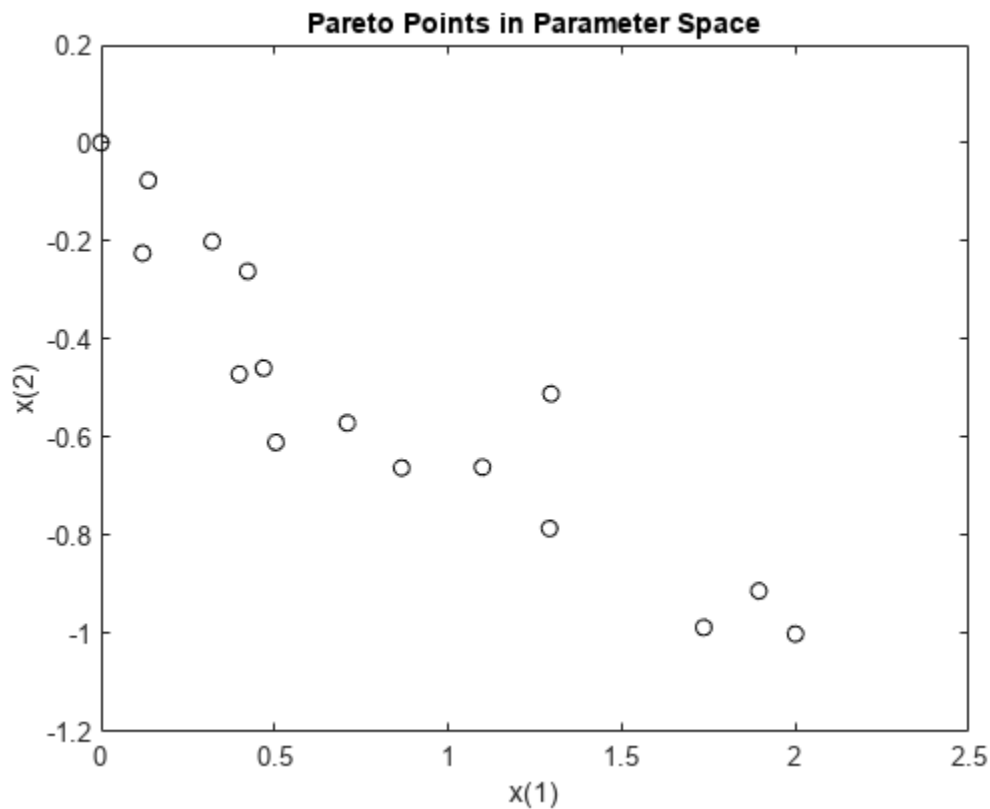
Find the Pareto front for this objective function.

```
rng default % For reproducibility
x = gamultiobj(fitnessfcn,2);
```

Optimization terminated: average change in the spread of Pareto solutions less than options.Function

Plot the solution points.

```
plot(x(:,1),x(:,2),'ko')
xlabel('x(1)')
ylabel('x(2)')
title('Pareto Points in Parameter Space')
```



To see the effect of a linear constraint on this problem, see “Multiobjective Problem with Linear Constraint” on page 18-38.

Multiobjective Problem with Linear Constraint

This example shows how to find the Pareto front for a multiobjective problem in the presence of a linear constraint.

There are two objective functions and two decision variables x .

```
fitnessfcn = @(x)[norm(x)^2,0.5*norm(x(:)-[2;-1])^2+2];
```

The linear constraint is $x(1) + x(2) \leq 1/2$.

```
A = [1,1];
b = 1/2;
```

Find the Pareto front.

```
rng default % For reproducibility
x = gamultiobj(fitnessfcn,2,A,b);
```

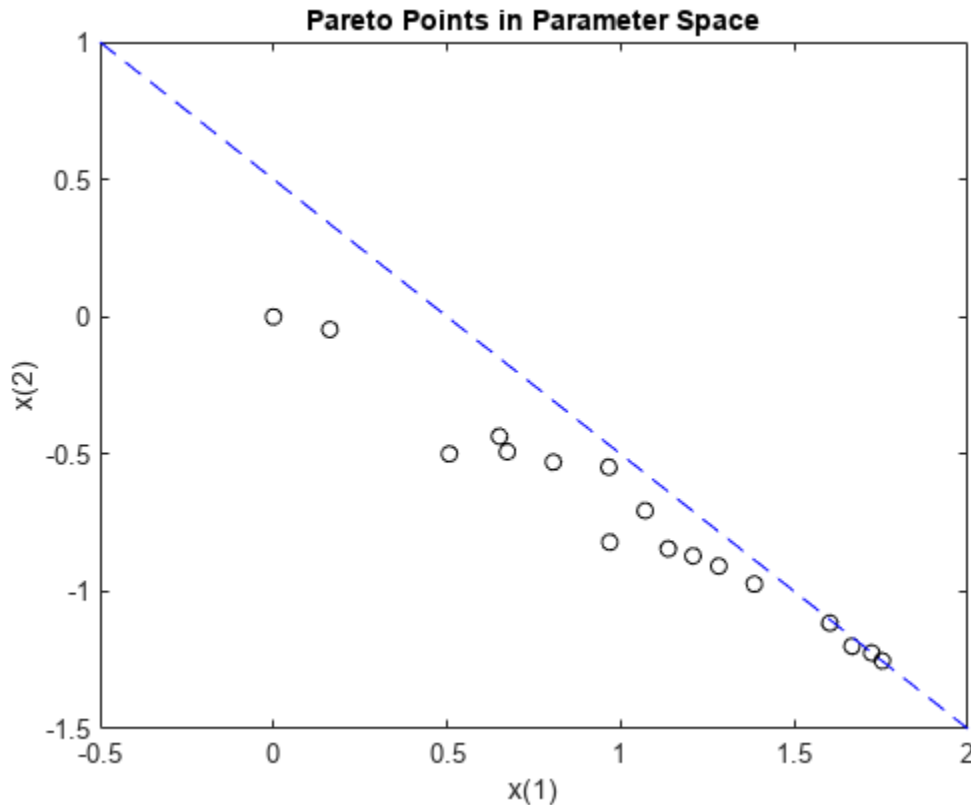
Optimization terminated: average change in the spread of Pareto solutions less than options.Function

Plot the constrained solution and the linear constraint.

```

plot(x(:,1),x(:,2),'ko')
t = linspace(-1/2,2);
y = 1/2 - t;
hold on
plot(t,y,'b--')
xlabel('x(1)')
ylabel('x(2)')
title('Pareto Points in Parameter Space')
hold off

```



To see the effect of removing the linear constraint from this problem, see “Simple Multiobjective Problem” on page 18-37.

Multiobjective Optimization with Bound Constraints

Find the Pareto front for the two fitness functions $\sin(x)$ and $\cos(x)$ on the interval $0 \leq x \leq 2\pi$.

```

fitnessfcn = @(x)[sin(x),cos(x)];
nvars = 1;
lb = 0;
ub = 2*pi;
rng default % for reproducibility
x = gamultiobj(fitnessfcn,nvars,[],[],[],[],lb,ub)

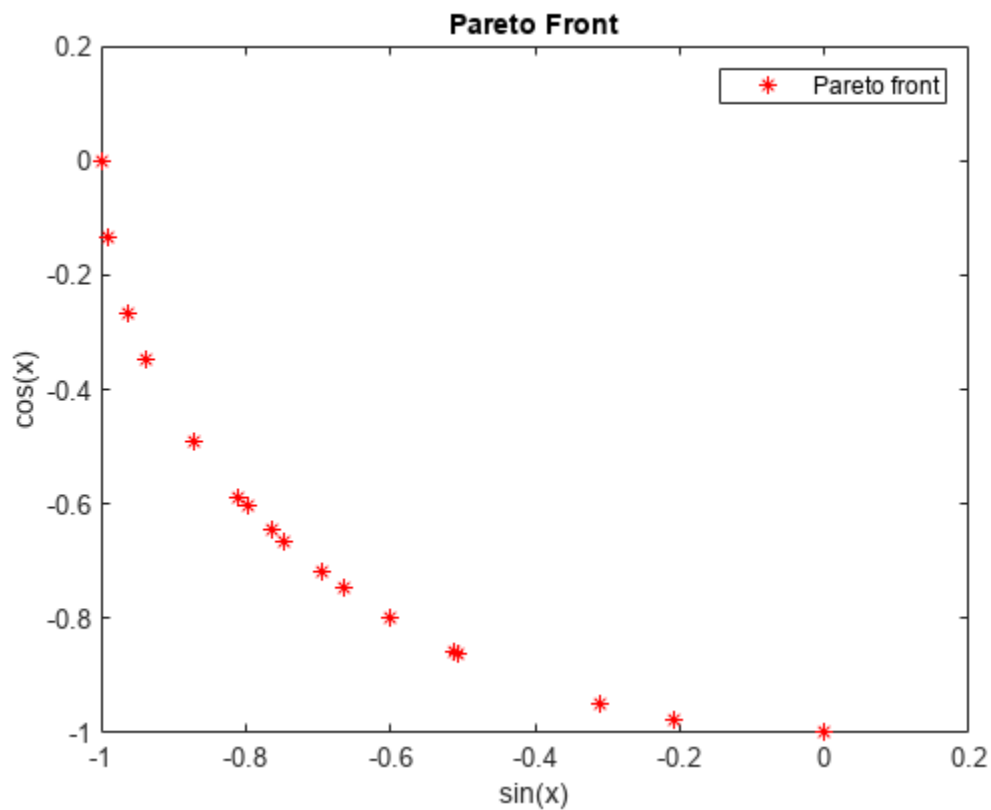
```

Optimization terminated: average change in the spread of Pareto solutions less than options.Function

```
x = 18x1
4.7124
4.7124
3.1415
3.6733
3.9845
3.4582
3.9098
4.4409
4.0846
3.8686
⋮
```

Plot the solution. `gamultiobj` finds points along the entire Pareto front.

```
plot(sin(x),cos(x),'r*')
xlabel('sin(x)')
ylabel('cos(x)')
title('Pareto Front')
legend('Pareto front')
```



Disconnected Pareto Front

Find and plot the Pareto front for the two-objective Schaffer's second function. This function has a disconnected Pareto front.

Copy this code to a function file on your MATLAB® path.

```
function y = schaffer2(x) % y has two columns

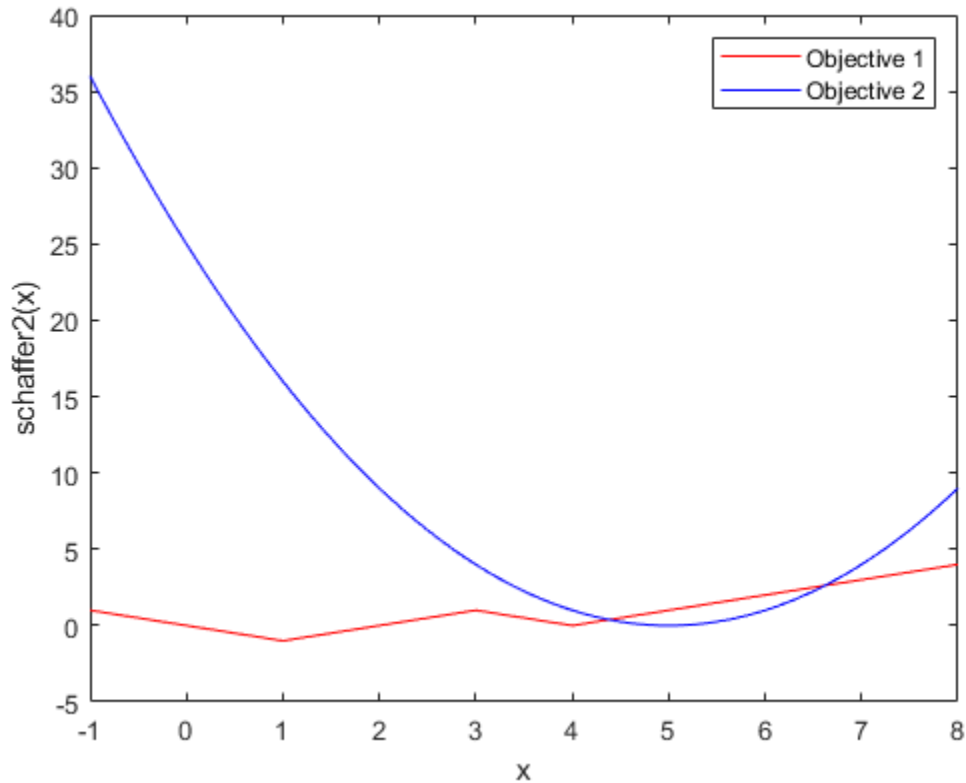
% Initialize y for two objectives and for all x
y = zeros(length(x),2);

% Evaluate first objective.
% This objective is piecewise continuous.
for i = 1:length(x)
    if x(i) <= 1
        y(i,1) = -x(i);
    elseif x(i) <=3
        y(i,1) = x(i) -2;
    elseif x(i) <=4
        y(i,1) = 4 - x(i);
    else
        y(i,1) = x(i) - 4;
    end
end

% Evaluate second objective
y(:,2) = (x -5).^2;
```

Plot the two objectives.

```
x = -1:0.1:8;
y = schaffer2(x);
plot(x,y(:,1),'r',x,y(:,2),'b');
xlabel x
ylabel 'schaffer2(x)'
legend('Objective 1','Objective 2')
```



The two objective functions compete for x in the ranges $[1, 3]$ and $[4, 5]$. But, the Pareto-optimal front consists of only two disconnected regions, corresponding to the x in the ranges $[1, 2]$ and $[4, 5]$. There are disconnected regions because the region $[2, 3]$ is inferior to $[4, 5]$. In that range, objective 1 has the same values, but objective 2 is smaller.

Set bounds to keep population members in the range $-5 \leq x \leq 10$.

```
lb = -5;
ub = 10;
```

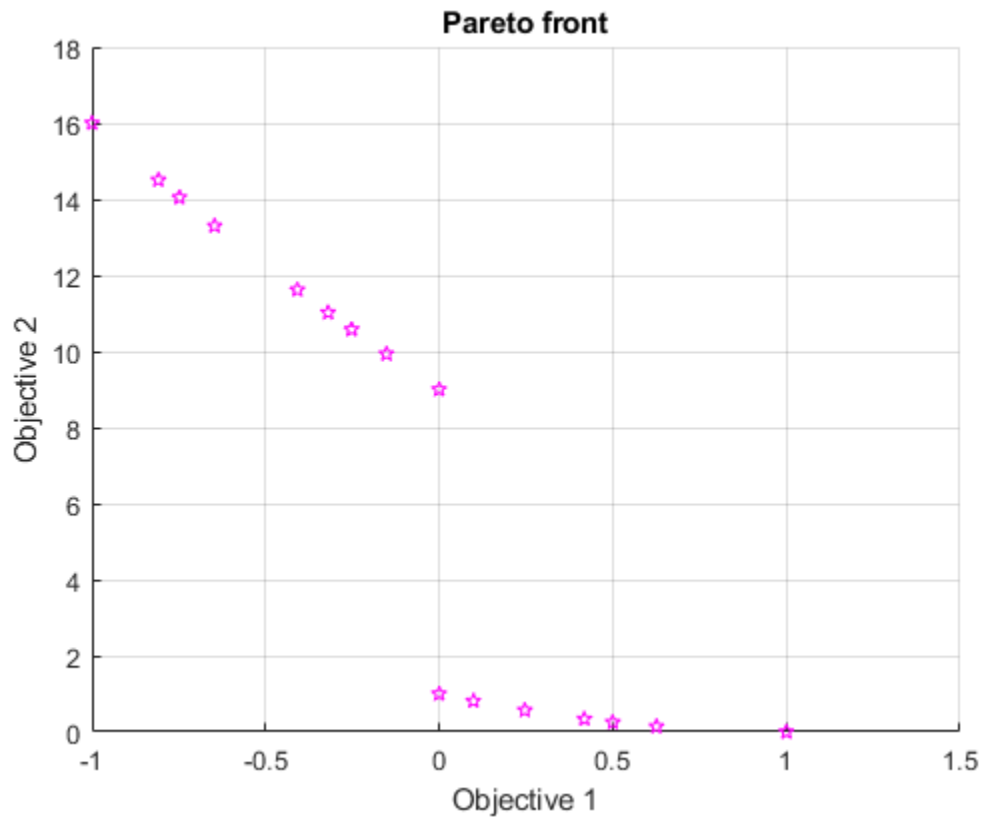
Set options to view the Pareto front as `gamultiobj` runs.

```
options = optimoptions('gamultiobj','PlotFcn',@gaplotpareto);
```

Call `gamultiobj`.

```
rng default % For reproducibility
[x,fval,exitflag,output] = gamultiobj(@schaffer2,1,[],[],[],[],lb,ub,options);
```

Optimization terminated: maximum number of generations exceeded.



Integer gamultiobj

Create a two-objective function in two problem variables.

```
rng default % For reproducibility
M = diag([-1 -1]) + randn(2)/4; % Two problem variables
fun = @(x)[(x').^2 / 30 + M*x']; % Two objectives
```

Specify that the second variable must be an integer.

```
intcon = 2;
```

Specify problem bounds, the `gaplotpareto` plot function, and a population size of 100.

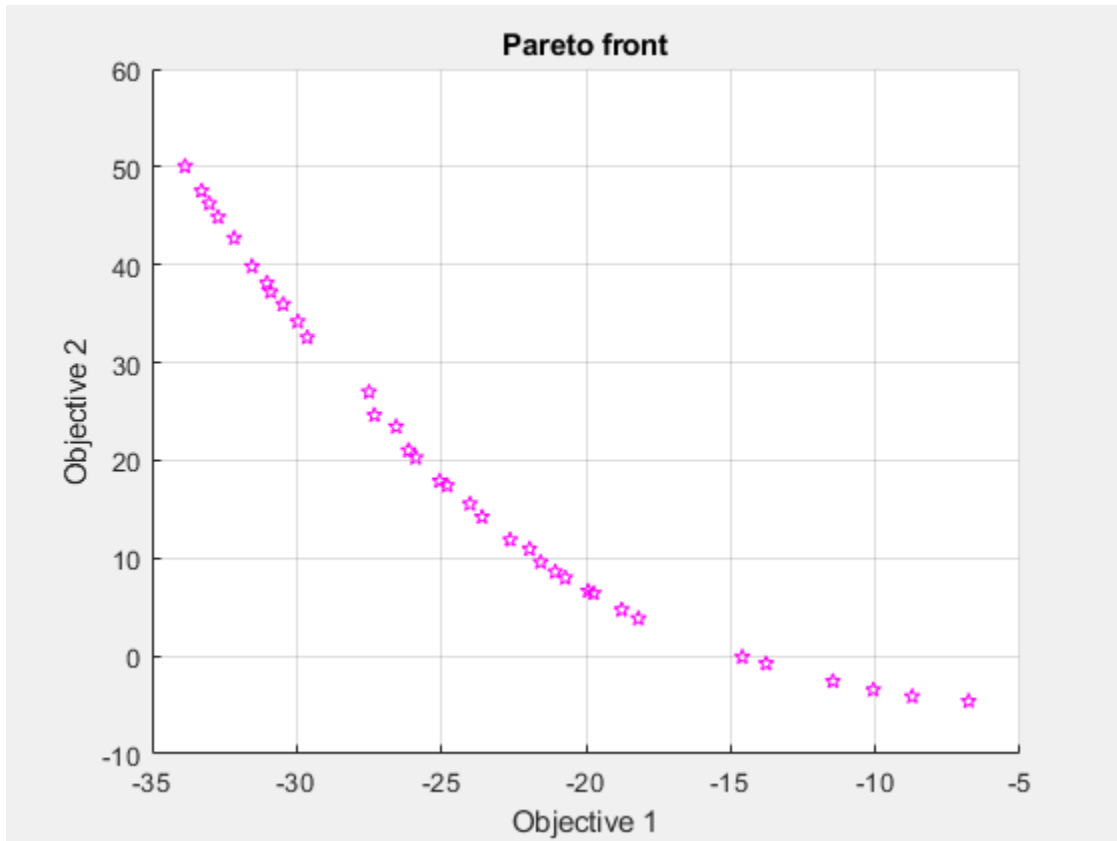
```
lb = [0 0];
ub = [100 50];
options = optimoptions("gamultiobj", "PlotFcn", "gaplotpareto", ...
    "PopulationSize", 100);
```

Find the Pareto set for the problem.

```
nvars = 2;
A = [];
b = [];
Aeq = [];
beq = [];
```

```
nonlcon = [];
[x,fval] = gamultiobj(fun,nvars,A,b,Aeq,beq,lb,ub,nonlcon,intcon,options);
```

Optimization terminated: average change in the spread of Pareto solutions less than options.Function



List ten of the solutions, and notice that the second variable is integer-valued.

```
x(1:10,:)
```

```
ans = 10x2
```

```
8.3393 28.0000
12.9927 49.0000
7.1611 27.0000
7.0210 18.0000
0.0004 12.0000
9.0989 44.0000
9.3974 29.0000
0.5537 17.0000
6.4010 17.0000
7.0531 31.0000
```

Obtain All Outputs from gamultiobj

Run a simple multiobjective problem and obtain all available outputs.

Set the random number generator for reproducibility.

```
rng default
```

Set the fitness functions to `kur_multiobjective`, a function that has three control variables and returns two fitness function values.

```
fitnessfcn = @kur_multiobjective;
nvars = 3;
```

The `kur_multiobjective` function has the following code.

```
function y = kur_multiobjective(x)
%KUR_MULTIOBJECTIVE Objective function for a multiobjective problem.
% The Pareto-optimal set for this two-objective problem is nonconvex as
% well as disconnected. The function KUR_MULTIOBJECTIVE computes two
% objectives and returns a vector y of size 2-by-1.
%
% Reference: Kalyanmoy Deb, "Multi-Objective Optimization using
% Evolutionary Algorithms", John Wiley & Sons ISBN 047187339
%
% Copyright 2007 The MathWorks, Inc.

% Initialize for two objectives
y = zeros(2,1);

% Compute first objective
for i = 1:2
    y(1) = y(1) - 10*exp(-0.2*sqrt(x(i)^2 + x(i+1)^2));
end

% Compute second objective
for i = 1:3
    y(2) = y(2) + abs(x(i))^0.8 + 5*sin(x(i)^3);
end
```

Set lower and upper bounds on all variables.

```
ub = [5 5 5];
lb = -ub;
```

Find the Pareto front and all other outputs for this problem.

```
[x,fval,exitflag,output,population,scores] = gamultiobj(fitnessfcn,nvars, ...
    [],[],[],[],lb,ub);
```

Optimization terminated: average change in the spread of Pareto solutions less than options.Function

Examine the sizes of some of the returned variables.

```
size_x = size(x)
size_population = size(population)
size_scores = size(scores)
```

```
size_x =
```

```
18      3

sizepopulation =
50      3

sisescores =
50      2
```

The returned Pareto front contains 18 points. There are 50 members of the final population. Each `population` row has three dimensions, corresponding to the three decision variables. Each `sisescores` row has two dimensions, corresponding to the two fitness functions.

Input Arguments

fun — Fitness functions to optimize

function handle | function name

Fitness functions to optimize, specified as a function handle or function name.

`fun` is a function that accepts a real row vector of doubles `x` of length `nvars` and returns a real vector `F(x)` of objective function values. For details on writing `fun`, see “Compute Objective Functions” on page 2-2.

If you set the `UseVectorized` option to `true`, then `fun` accepts a matrix of size `n-by-nvars`, where the matrix represents `n` individuals. `fun` returns a matrix of size `n-by-m`, where `m` is the number of objective functions. See “Vectorize the Fitness Function” on page 8-103.

Example: `@(x)[sin(x),cos(x)]`

Data Types: `char` | `function_handle` | `string`

nvars — Number of variables

positive integer

Number of variables, specified as a positive integer. The solver passes row vectors of length `nvars` to `fun`.

Example: 4

Data Types: `double`

A — Linear inequality constraints

real matrix

Linear inequality constraints, specified as a real matrix. `A` is an `M-by-nvars` matrix, where `M` is the number of inequalities.

`A` encodes the `M` linear inequalities

$A*x \leq b$,

where x is the column vector of `nvars` variables $x(:)$, and b is a column vector with M elements.

For example, give constraints $A = [1,2;3,4;5,6]$ and $b = [10;20;30]$ to specify these sums:

$$\begin{aligned}x_1 + 2x_2 &\leq 10 \\3x_1 + 4x_2 &\leq 20 \\5x_1 + 6x_2 &\leq 30.\end{aligned}$$

Example: To set the sum of the x -components to 1 or less, take $A = \text{ones}(1,N)$ and $b = 1$.

Data Types: `double`

b – Linear inequality constraints

real vector

Linear inequality constraints, specified as a real vector. b is an M -element vector related to the A matrix. If you pass b as a row vector, solvers internally convert b to the column vector $b(:)$.

b encodes the M linear inequalities

$$A*x \leq b,$$

where x is the column vector of `nvars` variables $x(:)$, and A is a matrix of size M -by-`nvars`.

For example, give constraints $A = [1,2;3,4;5,6]$ and $b = [10;20;30]$ to specify these sums:

$$\begin{aligned}x_1 + 2x_2 &\leq 10 \\3x_1 + 4x_2 &\leq 20 \\5x_1 + 6x_2 &\leq 30.\end{aligned}$$

Example: To set the sum of the x -components to 1 or less, take $A = \text{ones}(1,N)$ and $b = 1$.

Data Types: `double`

Aeq – Linear equality constraints

real matrix

Linear equality constraints, specified as a real matrix. Aeq is an Me -by-`nvars` matrix, where Me is the number of equalities.

Aeq encodes the Me linear equalities

$$Aeq*x = beq,$$

where x is the column vector of `nvars` variables $x(:)$, and beq is a column vector with Me elements.

For example, give constraints $Aeq = [1,2,3;2,4,1]$ and $beq = [10;20]$ to specify these sums:

$$\begin{aligned}x_1 + 2x_2 + 3x_3 &= 10 \\2x_1 + 4x_2 + x_3 &= 20.\end{aligned}$$

Example: To set the sum of the x -components to 1, take $Aeq = \text{ones}(1,N)$ and $beq = 1$.

Data Types: `double`

beq – Linear equality constraints

real vector

Linear equality constraints, specified as a real vector. `beq` is an `Me`-element vector related to the `Aeq` matrix. If you pass `beq` as a row vector, solvers internally convert `beq` to the column vector `beq(:)`.

`beq` encodes the `Me` linear equalities

$$\text{Aeq} * x = \text{beq},$$

where `x` is the column vector of `nvars` variables `x(:)`, and `Aeq` is a matrix of size `Meq`-by-`N`.

For example, give constraints `Aeq = [1,2,3;2,4,1]` and `beq = [10;20]` to specify these sums:

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 10 \\ 2x_1 + 4x_2 + x_3 &= 20. \end{aligned}$$

Example: To set the sum of the `x`-components to 1, take `Aeq = ones(1,N)` and `beq = 1`.

Data Types: `double`

lb — Lower bounds

real vector | real array

Lower bounds, specified as a real vector or real array. If `numel(lb) = nvars`, then `lb` specifies that `x(i) >= lb(i)` for all `i`.

If `numel(lb) < nvars`, then `lb` specifies that `x(i) >= lb(i)` for `1 <= i <= numel(lb)`.

In this case, solvers issue a warning.

Example: To specify all `x`-components as positive, set `lb = zeros(nvars,1)`.

Data Types: `double`

ub — Upper bounds

real vector | real array

Upper bounds, specified as a real vector or real array. If `numel(ub) = nvars`, then `ub` specifies that `x(i) <= ub(i)` for all `i`.

If `numel(ub) < nvars`, then `ub` specifies that `x(i) <= ub(i)` for `1 <= i <= numel(ub)`.

In this case, solvers issue a warning.

Example: To specify all `x`-components as less than one, set `ub = ones(nvars,1)`.

Data Types: `double`

nonlcon — Nonlinear constraints

function handle | function name

Nonlinear constraints, specified as a function handle or function name. `nonlcon` is a function that accepts a row vector `x` and returns two row vectors, `c(x)` and `ceq(x)`.

- `c(x)` is the row vector of nonlinear inequality constraints at `x`. The `gamultiobj` function attempts to satisfy `c(x) <= 0` for all entries of `c`.
- `ceq(x)` is the row vector nonlinear equality constraints at `x`. The `gamultiobj` function attempts to satisfy `ceq(x) = 0` for all entries of `ceq`.

If you set the `UseVectorized` option to `true`, then `nonlcon` accepts a matrix of size `n-by-nvars`, where the matrix represents `n` individuals. `nonlcon` returns a matrix of size `n-by-mc` in the first argument, where `mc` is the number of nonlinear inequality constraints. `nonlcon` returns a matrix of size `n-by-mceq` in the second argument, where `mceq` is the number of nonlinear equality constraints. See “Vectorize the Fitness Function” on page 8-103.

For example, `x = gamultiobj(@myfun,nvars,A,b,Aeq,beq,lb,ub,@mycon)`, where `mycon` is a MATLAB function such as the following:

```
function [c,ceq] = mycon(x)
c = ...      % Compute nonlinear inequalities at x.
ceq = ...    % Compute nonlinear equalities at x.
```

For more information, see “Nonlinear Constraints”.

Data Types: `char` | `function_handle` | `string`

options — Optimization options

output of `optimoptions` | structure

Optimization options, specified as the output of `optimoptions` or a structure. See option details in “Genetic Algorithm Options” on page 17-23.

`optimoptions` hides the options listed in *italics*. See “Options that `optimoptions` Hides” on page 17-65.

- Values in `{}` denote the default value.
- `{}`* represents the default when there are linear constraints, and for `MutationFcn` also when there are bounds.
- **I*** indicates that the solver handles options for integer constraints differently.
- **NM** indicates that the option does not apply to `gamultiobj`.

Options for `ga` and `gamultiobj`

Option	Description	Values
ConstraintTolerance	Determines the feasibility with respect to nonlinear constraints. Also, <code>max(sqrt(eps), ConstraintTolerance)</code> determines feasibility with respect to linear constraints. For an options structure, use <code>TolCon</code> .	Positive scalar <code>{1e-3}</code>
CreationFcn	Function that creates the initial population. Specify as a name of a built-in creation function or a function handle. See “Population Options” on page 17-26.	<code>{'gacreationuniform'} {'gacreationlinearfeasible'}* 'gacreationnonlinearfeasible' {'gacreationuniformint'}I*</code> for <code>ga</code> <code>{'gacreationsobol'}I*</code> for <code>gamultiobj</code> Custom creation function on page 17-26
CrossoverFcn	Function that the algorithm uses to create crossover children. Specify as a name of a built-in crossover function or a function handle. See “Crossover Options” on page 17-34.	<code>{'crossoverscattered'}</code> for <code>ga</code> , <code>{'crossoverintermediate'}*</code> for <code>gamultiobj</code> <code>{'crossoverlaplace'}I*</code> <code>'crossoverheuristic'</code> <code>'crossoversinglepoint'</code> <code>'crossovertwopoint'</code> <code>'crossoverarithmetic'</code> Custom crossover function on page 17-34
CrossoverFraction	The fraction of the population at the next generation, not including elite children, that the crossover function creates.	Positive scalar <code>{0.8}</code>
Display	Level of display.	<code>'off'</code> <code>'iter'</code> <code>'diagnose'</code> <code>{'final'}</code>
DistanceMeasureFcn	Function that computes the distance measure of individuals. Specify as a name of a built-in distance measure function or a function handle. The value applies to the decision variable or design space (genotype) or to function space (phenotype). The default <code>'distancecrowding'</code> is in function space (phenotype). For <code>gamultiobj</code> only. See “Multiobjective Options” on page 17-38. For an options structure, use a function handle, not a name.	<code>{'distancecrowding'}</code> means the same as <code>{@distancecrowding, 'phenotype'}</code> <code>{@distancecrowding, 'genotype'}</code> Custom distance function on page 17-38

Option	Description	Values
EliteCount	NM Positive integer specifying how many individuals in the current generation are guaranteed to survive to the next generation. Not used in gamultiobj.	Positive integer $\{\text{ceil}(0.05 * \text{PopulationSize})\}$ $\{0.05 * (\text{default PopulationSize})\}$ for mixed-integer problems
FitnessLimit	NM If the fitness function attains the value of FitnessLimit, the algorithm halts.	Scalar $\{-\text{Inf}\}$
FitnessScalingFcn	Function that scales the values of the fitness function. Specify as a name of a built-in scaling function or a function handle. Option unavailable for gamultiobj.	<code>'fitscalingrank'</code> <code>'fitscalingshiftlinear'</code> <code>'fitscalingprop'</code> <code>'fitscalingtop'</code> Custom fitness scaling function on page 17-29
FunctionTolerance	The algorithm stops if the average relative change in the best fitness function value over MaxStallGenerations generations is less than or equal to FunctionTolerance. If StallTest is 'geometricWeighted', then the algorithm stops if the weighted average relative change is less than or equal to FunctionTolerance. For gamultiobj, the algorithm stops when the geometric average of the relative change in value of the spread over options.MaxStallGenerations generations is less than options.FunctionTolerance, and the final spread is less than the mean spread over the past options.MaxStallGenerations generations. See “gamultiobj Algorithm” on page 14-5. For an options structure, use TolFun.	Positive scalar $\{1e-6\}$ for ga, $\{1e-4\}$ for gamultiobj
HybridFcn	I* Function that continues the optimization after ga terminates. Specify as a name or a function handle. Alternatively, a cell array specifying the hybrid function and its options. See “ga Hybrid Function” on page 17-39. For gamultiobj, the only hybrid function is @fgoalattain. See “gamultiobj Hybrid Function” on page 17-40. When the problem has integer constraints, you cannot use a hybrid function. See “When to Use a Hybrid Function” on page 8-116.	Function name or handle <code>'fminsearch'</code> <code>'patternsearch'</code> <code>'fminunc'</code> <code>'fmincon'</code> <code>{[]}</code> or 1-by-2 cell array <code>{@solver, hybridoptions}</code> , where solver = fminsearch, patternsearch, fminunc, or fmincon <code>{[]}</code>

Option	Description	Values
<i>InitialPenalty</i>	NM I* Initial value of the penalty parameter	Positive scalar {10}
<i>InitialPopulationMatrix</i>	Initial population used to seed the genetic algorithm. Has up to <i>PopulationSize</i> rows and <i>N</i> columns, where <i>N</i> is the number of variables. You can pass a partial population, meaning one with fewer than <i>PopulationSize</i> rows. In that case, the genetic algorithm uses <i>CreationFcn</i> to generate the remaining population members. See “Population Options” on page 17-26. For an options structure, use <i>InitialPopulation</i> .	Matrix {[]}
<i>InitialPopulationRange</i>	Matrix or vector specifying the range of the individuals in the initial population. Applies to <i>gacreationuniform</i> creation function. <i>ga</i> shifts and scales the default initial range to match any finite bounds. For an options structure, use <i>PopInitRange</i> .	Matrix or vector {[-10;10]} for unbounded components, {[-1e4+1;1e4+1]} for unbounded components of integer-constrained problems, {[lb;ub]} for bounded components, with the default range modified to match one-sided bounds
<i>InitialScoresMatrix</i>	Initial scores used to determine fitness. Has up to <i>PopulationSize</i> rows and <i>Nf</i> columns, where <i>Nf</i> is the number of fitness functions (1 for <i>ga</i> , greater than 1 for <i>gamultiobj</i>). You can pass a partial scores matrix, meaning one with fewer than <i>PopulationSize</i> rows. In that case, the solver fills in the scores when it evaluates the fitness functions. For an options structure, use <i>InitialScores</i> .	Column vector for single objective matrix for multiobjective {[]}
<i>MaxGenerations</i>	Maximum number of iterations before the algorithm halts. For an options structure, use <i>Generations</i> .	Positive integer {100*numberOfVariables} for <i>ga</i> , {200*numberOfVariables} for <i>gamultiobj</i>

Option	Description	Values
MaxStallGenerations	<p>The algorithm stops if the average relative change in the best fitness function value over MaxStallGenerations generations is less than or equal to FunctionTolerance. If StallTest is 'geometricWeighted', then the algorithm stops if the weighted average relative change is less than or equal to FunctionTolerance.</p> <p>For gamultiobj, the algorithm stops when the geometric average of the relative change in value of the spread over options.MaxStallGenerations generations is less than options.FunctionTolerance, and the final spread is less than the mean spread over the past options.MaxStallGenerations generations. See “gamultiobj Algorithm” on page 14-5.</p> <p>For an options structure, use StallGenLimit.</p>	Positive integer {50} for ga, {100} for gamultiobj
MaxStallTime	<p>NM The algorithm stops if there is no improvement in the objective function for MaxStallTime seconds, as measured by tic and toc.</p> <p>For an options structure, use StallTimeLimit.</p>	Positive scalar {Inf}
MaxTime	<p>The algorithm stops after running for MaxTime seconds, as measured by tic and toc. This limit is enforced after each iteration, so ga can exceed the limit when an iteration takes substantial time.</p> <p>For an options structure, use TimeLimit.</p>	Positive scalar {Inf}
MigrationDirection	Direction of migration. See “Migration Options” on page 17-37.	'both' {'forward'}
MigrationFraction	Scalar from 0 through 1 specifying the fraction of individuals in each subpopulation that migrates to a different subpopulation. See “Migration Options” on page 17-37.	Scalar {0.2}
MigrationInterval	Positive integer specifying the number of generations that take place between migrations of individuals between subpopulations. See “Migration Options” on page 17-37.	Positive integer {20}

Option	Description	Values
MutationFcn	Function that produces mutation children. Specify as a name of a built-in mutation function or a function handle. See “Mutation Options” on page 17-31.	{'mutationgaussian'} for ga without constraints {'mutationadaptfeasible'}* for gamultiobj and for ga with constraints {'mutationpower'}I* 'mutationpositivebasis' 'mutationuniform' Custom mutation function on page 17-31
NonlinearConstraintAlgorithm	Nonlinear constraint algorithm. See “Nonlinear Constraint Solver Algorithms for Genetic Algorithm” on page 8-56. Option unchangeable for gamultiobj. For an options structure, use NonlinConAlgorithm.	{'auglag'} for ga, {'penalty'} for gamultiobj
OutputFcn	Functions that ga calls at each iteration. Specify as a function handle or a cell array of function handles. See “Output Function Options” on page 17-41. For an options structure, use OutputFcns.	Function handle or cell array of function handles {[]}
ParetoFraction	Scalar from 0 through 1 specifying the fraction of individuals to keep on the first Pareto front while the solver selects individuals from higher fronts, for gamultiobj only. See “Multiobjective Options” on page 17-38.	Scalar {0.35}
PenaltyFactor	NM I* Penalty update parameter.	Positive scalar {100}
PlotFcn	Function that plots data computed by the algorithm. Specify as a name of a built-in plot function, a function handle, or a cell array of built-in names or function handles. See “Plot Options” on page 17-23. For an options structure, use PlotFcns.	ga or gamultiobj: {[]} 'gaplotdistance' 'gaplotgenealogy' 'gaplotselection' 'gaplotscorediversity' 'gaplotscores' 'gaplotstopping' 'gaplotmaxconstr' Custom plot function on page 17-23 ga only: 'gaplotbestf' 'gaplotbestindiv' 'gaplotexpectation' 'gaplotrange' gamultiobj only: 'gaplotpareto' 'gaplotparetodistance' 'gaplotrankhist' 'gaplotspread'

Option	Description	Values
<i>PlotInterval</i>	Positive integer specifying the number of generations between consecutive calls to the plot functions.	Positive integer {1}
PopulationSize	Size of the population.	Positive integer {50} when numberOfVariables <= 5, {200} otherwise {min(max(10*nvars,40),100)} for mixed-integer problems
PopulationType	Data type of the population. Must be 'doubleVector' for mixed-integer problems.	'bitstring' 'custom' {'doubleVector'} ga ignores all constraints when PopulationType is set to 'bitString' or 'custom'. See "Population Options" on page 17-26.
SelectionFcn	Function that selects parents of crossover and mutation children. Specify as a name of a built-in selection function or a function handle. gamultiobj uses only 'selectiontournament'.	{'selectionstochunif'} for ga, {'selectiontournament'} for gamultiobj 'selectionremainder' 'selectionuniform' 'selectionroulette' Custom selection function on page 17-30
<i>StallTest</i>	NM Stopping test type.	'geometricWeighted' {'averageChange'}
UseParallel	Compute fitness and nonlinear constraint functions in parallel. See "Vectorize and Parallel Options (User Function Evaluation)" on page 17-43 and "How to Use Parallel Processing in Global Optimization Toolbox" on page 16-11.	true {false}
UseVectorized	Specifies whether functions are vectorized. See "Vectorize and Parallel Options (User Function Evaluation)" on page 17-43 and "Vectorize the Fitness Function" on page 8-103. For an options structure, use Vectorized with the values 'on' or 'off'.	true {false}

Example: `optimoptions('gamultiobj','PlotFcn',@gaplotpareto)`

intcon – Integer variables

vector of positive integers

Integer variables, specified as a vector of positive integers taking values from 1 to nvars. Each value in intcon represents an x component that is integer-valued.

Note When `intcon` is nonempty, `nonlcon` must return empty for `ceq`.

Example: To specify that the even entries in `x` are integer-valued, set `intcon` to `2:2:nvars`

Data Types: `double`

problem — Problem description

structure

Problem description, specified as a structure containing these fields.

<code>fitnessfcn</code>	Fitness functions
<code>nvars</code>	Number of design variables
<code>Aineq</code>	A matrix for linear inequality constraints
<code>Bineq</code>	b vector for linear inequality constraints
<code>Aeq</code>	Aeq matrix for linear equality constraints
<code>Beq</code>	beq vector for linear equality constraints
<code>lb</code>	Lower bound on <code>x</code>
<code>ub</code>	Upper bound on <code>x</code>
<code>nonlcon</code>	Nonlinear constraint functions
<code>intcon</code>	Indices of integer variables
<code>rngstate</code>	Field to reset the state of the random number generator
<code>solver</code>	'gamultiobj'
<code>options</code>	Options created using <code>optimoptions</code> or an options structure

You must specify the fields `fitnessfcn`, `nvars`, and `options`. The remainder are optional for `gamultiobj`.

Data Types: `struct`

Output Arguments

x — Pareto points

`m`-by-`nvars` array

Pareto points, returned as an `m`-by-`nvars` array, where `m` is the number of points on the Pareto front. Each row of `x` represents one point on the Pareto front.

fval — Function values on Pareto front

`m`-by-`nf` array

Function values on the Pareto front, returned as an `m`-by-`nf` array. `m` is the number of points on the Pareto front, and `nf` is the number of fitness functions. Each row of `fval` represents the function values at one Pareto point in `x`.

exitflag — Reason `gamultiobj` stopped

integer

Reason `gamultiobj` stopped, returned as an integer.

exitflag Value	Stopping Condition
1	Geometric average of the relative change in value of the spread over <code>options.MaxStallGenerations</code> generations is less than <code>options.FunctionTolerance</code> , and the final spread is less than the mean spread over the past <code>options.MaxStallGenerations</code> generations
0	Maximum number of generations exceeded
-1	Optimization terminated by an output function or plot function
-2	No feasible point found
-5	Time limit exceeded

output — Information about the optimization process

structure

Information about the optimization process, returned as a structure with these fields.

output Field	Meaning
<code>problemtype</code>	Type of problem: <ul style="list-style-type: none"> 'unconstrained' — No constraints 'boundconstraints' — Only bound constraints 'linearconstraints' — Linear constraints, with or without bound constraints 'nonlinearconstr' — Nonlinear constraints, with or without other types of constraints
<code>rngstate</code>	State of the MATLAB random number generator, just before the algorithm started. You can use the values in <code>rngstate</code> to reproduce the output of <code>gamultiobj</code> . See “Reproduce Results” on page 8-67.
<code>generations</code>	Total number of generations, excluding <code>HybridFcn</code> iterations.
<code>funccount</code>	Total number of function evaluations.
<code>message</code>	<code>gamultiobj</code> exit message.
<code>averagedistance</code>	Average “distance,” which by default is the standard deviation of the norm of the difference between Pareto front members and their mean.
<code>spread</code>	Combination of the “distance,” and a measure of the movement of the points on the Pareto front between the final two iterations.
<code>maxconstraint</code>	Maximum constraint violation at the final Pareto set.

population — Final population

n-by-nvars array

Final population, returned as an n-by-nvars array, where n is the number of members of the population.

scores — Scores of the final population

n-by-nf array

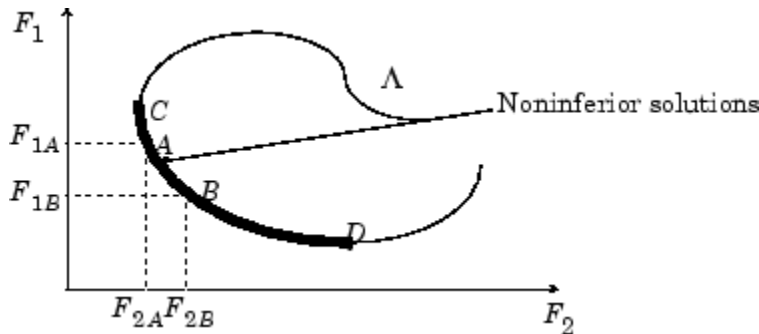
Scores of the final population, returned as an n-by-nf array. n is the number of members of the population, and nf is the number of fitness functions.

When there are nonlinear constraints, `gamultiobj` sets the scores of infeasible population members to `Inf`.

More About

Pareto Front

A Pareto front is a set of points in parameter space (the space of decision variables) that have noninferior fitness function values.



In other words, for each point on the Pareto front, you can improve one fitness function only by degrading another. For details, see “What Is Multiobjective Optimization?” on page 14-2

As in “Local vs. Global Optima”, it is possible for a Pareto front to be local, but not global. “Local” means that the Pareto points can be noninferior compared to nearby points, but points farther away in parameter space could have lower function values in every component.

Algorithms

`gamultiobj` uses a controlled, elitist genetic algorithm (a variant of NSGA-II [1]). An elitist GA always favors individuals with better fitness value (rank). A controlled elitist GA also favors individuals that can help increase the diversity of the population even if they have a lower fitness value. It is important to maintain the diversity of population for convergence to an optimal Pareto front. Diversity is maintained by controlling the elite members of the population as the algorithm progresses. Two options, `ParetoFraction` and `DistanceMeasureFcn`, control the elitism. `ParetoFraction` limits the number of individuals on the Pareto front (elite members). The distance function, selected by `DistanceMeasureFcn`, helps to maintain diversity on a front by favoring individuals that are relatively far away on the front. The algorithm stops if the spread, a measure of the movement of the Pareto front, is small. For details, see “`gamultiobj` Algorithm” on page 14-5.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for `gamultiobj`.

Version History

Introduced in R2007b

References

[1] Deb, Kalyanmoy. *Multi-Objective Optimization Using Evolutionary Algorithms*. Chichester, England: John Wiley & Sons, 2001.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set the 'UseParallel' option to true.

```
options = optimoptions('solvername','UseParallel',true)
```

For more information, see “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11.

See Also

ga | optimoptions | paretosearch | **Optimize**

Topics

“Pareto Front for Two Objectives” on page 14-19

“Performing a Multiobjective Optimization Using the Genetic Algorithm” on page 14-48

“What Is Multiobjective Optimization?” on page 14-2

“gamultiobj Options and Syntax: Differences from ga” on page 14-18

gaoptimget

(Not recommended) Obtain values of genetic algorithm options structure

Note `gaoptimget` is not recommended. Instead, query options using dot notation. For more information, see “Compatibility Considerations”.

Syntax

```
val = gaoptimget(options, 'name')
val = gaoptimget(options, 'name', default)
```

Description

`val = gaoptimget(options, 'name')` returns the value of the parameter `name` from the genetic algorithm options structure `options`. `gaoptimget(options, 'name')` returns an empty matrix `[]` if the value of `name` is not specified in `options`. It is only necessary to type enough leading characters of `name` to uniquely identify it. `gaoptimget` ignores case in parameter names.

`val = gaoptimget(options, 'name', default)` returns the `'name'` parameter, but will return the default value if the `name` parameter is not specified (or is `[]`) in `options`.

Version History

Introduced before R2006a

R2018b: `gaoptimget` is not recommended

Not recommended starting in R2018b

To query options, the `gaoptimget`, `psoptimget`, and `saoptimget` functions are not recommended. Instead, use dot notation. For example, to see the setting of the `Display` option in `opts`,

```
displayopt = opts.Display
% instead of
displayopt = gaoptimget(opts, 'Display')
```

Using automatic code completions, dot notation takes fewer keystrokes: `displayopt = opts.D`
Tab.

There are no plans to remove `gaoptimget`, `psoptimget`, and `saoptimget` at this time.

See Also

`ga` | `gamultiobj`

Topics

“Genetic Algorithm Options” on page 17-23

gaoptimset

(Not recommended) Create genetic algorithm options structure

Note `gaoptimset` is not recommended. Use `optimoptions` instead. For more information, see “Compatibility Considerations”.

Syntax

```
gaoptimset
options = gaoptimset(Name,Value)
options = gaoptimset
options = gaoptimset(@ga)
options = gaoptimset(@gamultiobj)
options = gaoptimset(olddopts,Name,Value)
options = gaoptimset(olddopts,newopts)
```

Description

`gaoptimset` with no input or output arguments displays a complete list of options with their valid values. Generally, the default values appear in brackets {}.

Note The indicated default values are not defaults for all problem types. For an updated list, evaluate `optimoptions('ga')` or `optimoptions('gamultiobj')`.

`options = gaoptimset(Name,Value)` creates a structure named `options` and sets the value of 'Name1' to `Value1`, 'Name2' to `Value2`, and so on. `gaoptimset` sets any unspecified options to [], meaning solvers use the default option values. You can specify a option name using only enough leading characters to define the name uniquely. `gaoptimset` ignores case for option names. For example, 'Display', 'display', and 'Disp' are equivalent option names.

`options = gaoptimset` (with no input arguments) creates a structure named `options` that contains the options for the genetic algorithm. In this case, `gaoptimset` sets all option values to [], indicating to use default values.

`options = gaoptimset(@ga)` or `options = gaoptimset(@gamultiobj)` creates an `options` structure containing options with explicit default values for the `ga` or `gamultiobj` solver, respectively.

`options = gaoptimset(olddopts,Name,Value)` creates a copy of `olddopts`, modifying the specified options with the specified values.

`options = gaoptimset(olddopts,newopts)` combines an existing `options` structure, `olddopts`, with a new `options` structure, `newopts`. Any options in `newopts` with nonempty values overwrite the corresponding options in `olddopts`.

Examples

View Genetic Algorithm Options

To see all the options available for `ga` and `gamultiobj` in an options structure, run `gamultiobj` with no input or output arguments.

`gaoptimset`

```

    PopulationType: [ 'bitstring'      | 'custom'      | {'doubleVector'} ]
    PopInitRange:  [ matrix            | {[-10;10]} ]
    PopulationSize: [ positive scalar ]
    EliteCount:    [ positive scalar | {0.05*PopulationSize} ]
    CrossoverFraction: [ positive scalar | {0.8} ]

    ParetoFraction: [ positive scalar | {0.35} ]

    MigrationDirection: [ 'both'          | {'forward'} ]
    MigrationInterval:  [ positive scalar | {20} ]
    MigrationFraction:  [ positive scalar | {0.2} ]

    Generations: [ positive scalar ]
    TimeLimit:   [ positive scalar | {Inf} ]
    FitnessLimit: [ scalar          | {-Inf} ]
    StallGenLimit: [ positive scalar ]
    StallTest:    [ 'geometricWeighted' | {'averageChange'} ]
    StallTimeLimit: [ positive scalar | {Inf} ]
    TolFun:       [ positive scalar ]

    TolCon: [ positive scalar | {1e-6} ]

    InitialPopulation: [ matrix          | {[]} ]
    InitialScores:    [ column vector  | {[]} ]

    NonlinConAlgorithm: [ 'penalty' | {'auglag'} ]
    InitialPenalty:    [ positive scalar | {10} ]
    PenaltyFactor:     [ positive scalar | {100} ]

    CreationFcn: [ function_handle | @gacreationuniform | @gacreationlinearfeasible ]
    FitnessScalingFcn: [ function_handle | @fitscalingshiftlinear | @fitscalingprop |
    @fitscalingtop | {@fitscalingrank} ]
    SelectionFcn: [ function_handle | @selectionremainder | @selectionuniform |
    @selectionroulette | @selectiontournament | @selectionstochunif ]
    CrossoverFcn: [ function_handle | @crossoverheuristic | @crossoverintermediate |
    @crossoversinglepoint | @crossovertwopoint | @crossoverarithmetic |
    @crossoverscattered ]
    MutationFcn: [ function_handle | @mutationuniform | @mutationadaptfeasible |
    @mutationgaussian ]
    DistanceMeasureFcn: [ function_handle | {@distancecrowding} ]
    HybridFcn: [ @fminsearch | @patternsearch | @fminunc | @fmincon | {} ]

    Display: [ 'off' | 'iter' | 'diagnose' | {'final'} ]
    OutputFns: [ function_handle | {} ]
    PlotFns: [ function_handle | @gaplotbestf | @gaplotbestindiv | @gaplotdistance |
    @gaplotexpectation | @gaplotgenealogy | @gaplotselection | @gaplottra
    @gaplotcorediversity | @gaplotcores | @gaplotstopping |
    @gaplotmaxconstr | @gaplotrankhist | @gaplotpareto | @gaplotspread |
    @gaplotparetodistance | {} ]
    PlotInterval: [ positive scalar | {1} ]

```

```
Vectorized: [ 'on' | {'off'} ]  
UseParallel: [ logical scalar | true | {false} ]
```

Set Nondefault Genetic Algorithm Options

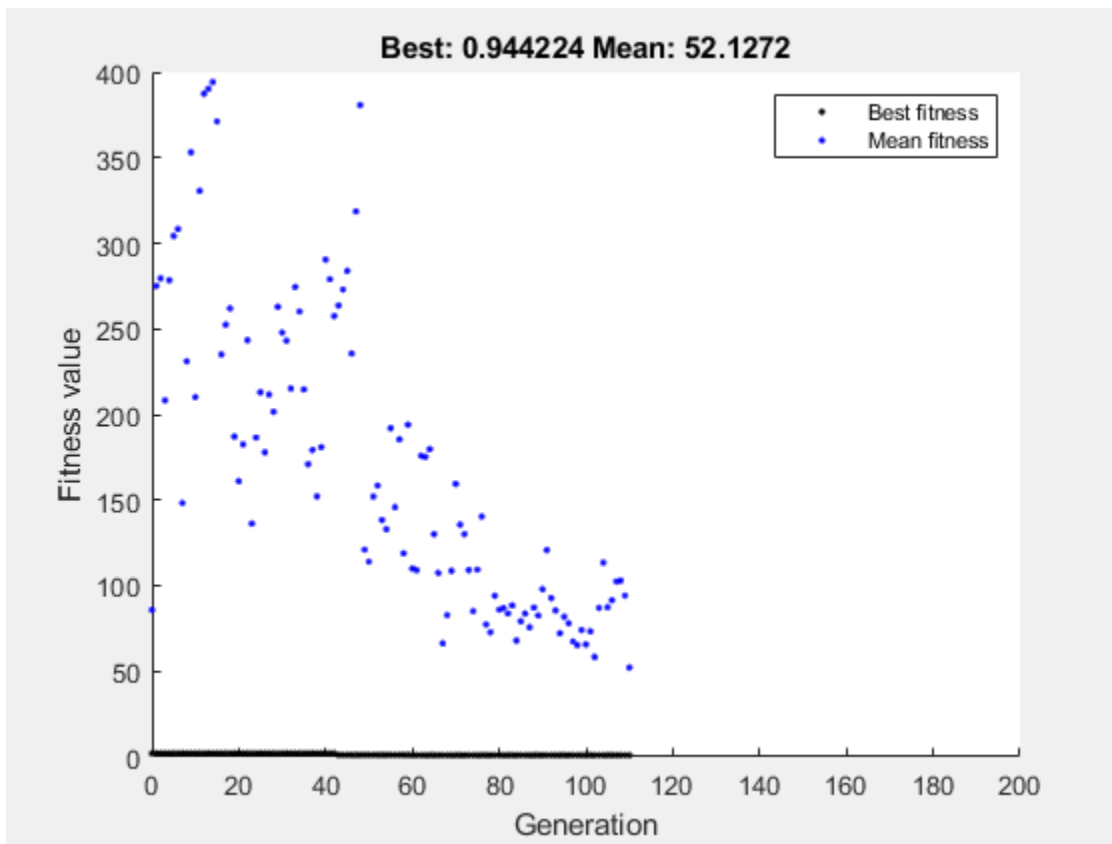
Set options for `ga` to have a starting population that includes the point `[1, 1]` and to use the `gaplotbestf` plot function.

```
options = gaoptimset('InitialPopulation',[1 1],...  
    'PlotFcns',@gaplotbestf);
```

Find a local minimum of the function `rastriginsfcn`, which is available when you run this example, using the specified options.

```
rng default % For reproducibility  
nvar = 2;  
A = [];  
b = [];  
Aeq = [];  
beq = [];  
lb = [];  
ub = [];  
nlconst = [];  
[x,fval] = ga(@rastriginsfcn,nvar,A,b,Aeq,beq,lb,ub,nlconst,options)
```

Optimization terminated: average change in the fitness value less than options.FunctionTolerance



```
x = 1x2
```

```
0.0227 0.0656
```

```
fval = 0.9442
```

Create Default ga Options

Use `gaoptimset` to create an option structure for `ga` with the values set to their defaults.

```
options = gaoptimset(@ga)
```

```
options = struct with fields:
    PopulationType: 'doubleVector'
    PopInitRange: []
    PopulationSize: '50 when numberOfVariables <= 5, else 200'
    EliteCount: '0.05*PopulationSize'
    CrossoverFraction: 0.8000
    ParetoFraction: []
    MigrationDirection: 'forward'
    MigrationInterval: 20
    MigrationFraction: 0.2000
    Generations: '100*numberOfVariables'
    TimeLimit: Inf
```



```

    FitnessLimit: -Inf
    StallGenLimit: 50
        StallTest: 'averageChange'
    StallTimeLimit: Inf
        TolFun: 1.0000e-06
        TolCon: 1.0000e-03
    InitialPopulation: []
    InitialScores: []
    NonlinConAlgorithm: 'auglag'
    InitialPenalty: 10
    PenaltyFactor: 100
    PlotInterval: 1
    CreationFcn: []
    FitnessScalingFcn: @fitscalingrank
    SelectionFcn: []
    CrossoverFcn: []
    MutationFcn: []
    DistanceMeasureFcn: []
    HybridFcn: []
    Display: 'final'
    PlotFcns: []
    OutputFcns: []
    Vectorized: 'off'
    IntegerTolerance: 1.0000e-05
    UseParallel: 0

```

Modify Genetic Algorithm Options

Create genetic algorithm options to return an iterative display.

```
oldopts = gaoptimset('Display','iter');
```

Modify `oldopts` to include the `gaplotbestf` plot function and a population size of 250.

```
options = gaoptimset(oldopts,'PlotFcns',@gaplotbestf,...
    'PopulationSize',250)
```

`options` = struct with fields:

```

    PopulationType: []
    PopInitRange: []
    PopulationSize: 250
    EliteCount: []
    CrossoverFraction: []
    ParetoFraction: []
    MigrationDirection: []
    MigrationInterval: []
    MigrationFraction: []
    Generations: []
    TimeLimit: []
    FitnessLimit: []
    StallGenLimit: []
    StallTest: []
    StallTimeLimit: []
    TolFun: []
    TolCon: []

```

```
InitialPopulation: []
InitialScores: []
NonlinConAlgorithm: []
InitialPenalty: []
PenaltyFactor: []
PlotInterval: []
CreationFcn: []
FitnessScalingFcn: []
SelectionFcn: []
CrossoverFcn: []
MutationFcn: []
DistanceMeasureFcn: []
HybridFcn: []
Display: 'iter'
PlotFcns: @gaplotbestf
OutputFcns: []
Vectorized: []
IntegerTolerance: []
UseParallel: []
```

Combine Genetic Algorithm Options

Create two sets of genetic algorithm options, `oldopts` and `newopts`. Specify an iterative display and the `gaplotbestf` plot function for `oldopts`. Specify no display and a population size of 300 for `newopts`.

```
oldopts = gaoptimset('Display','iter','PlotFcns',@gaplotbestf);
newopts = gaoptimset('Display','off','PopulationSize',300);
```

Combine these options with `newopts` taking precedence.

```
options = gaoptimset(oldopts,newopts)
```

```
options = struct with fields:
```

```
PopulationType: []
PopInitRange: []
PopulationSize: 300
EliteCount: []
CrossoverFraction: []
ParetoFraction: []
MigrationDirection: []
MigrationInterval: []
MigrationFraction: []
Generations: []
TimeLimit: []
FitnessLimit: []
StallGenLimit: []
StallTest: []
StallTimeLimit: []
TolFun: []
TolCon: []
InitialPopulation: []
InitialScores: []
NonlinConAlgorithm: []
```

```

    InitialPenalty: []
    PenaltyFactor: []
    PlotInterval: []
    CreationFcn: []
    FitnessScalingFcn: []
    SelectionFcn: []
    CrossoverFcn: []
    MutationFcn: []
    DistanceMeasureFcn: []
    HybridFcn: []
    Display: 'off'
    PlotFcns: @gaplotbestf
    OutputFcns: []
    Vectorized: []
    IntegerTolerance: []
    UseParallel: []

```

Notice that the value of the `Display` option is the value specified in `newopts`.

Input Arguments

oldopts — Optimization options

structure

Optimization options, specified as a structure, such as the output of `gaoptimset`.

Data Types: `struct`

newopts — Optimization options

structure

Optimization options, specified as a structure, such as the output of `gaoptimset`. Any options in `newopts` with nonempty values overwrite the corresponding options in `oldopts` in this syntax:

```
options = gaoptimset(oldopts,newopts)
```

Data Types: `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

`gaoptimset` creates a structure. In the following table of option names, use the name at the bottom of the description after the phrase "For an options structure," if specified. The first listed name (Option) is for the `optimoptions` function, which is the recommended function for setting options. For example, to specify the tolerance on nonlinear constraint violation in `optimoptions` you set `'ConstraintTolerance'`, but for `gaoptimset` you set `'TolCon'`.

Example: `options = gaoptimset('Display','off','PlotFcns',@gaplotbestf)`

In the following table,

- Values in braces {} denote the default value.
- {}* represents the default when the problem has linear constraints, and when `MutationFcn` has bounds.
- **I*** indicates that `ga` handles options for integer constraints differently; this notation does not apply to `gamultiobj`.
- **NM** indicates that the option does not apply to `gamultiobj`.

`optimoptions` hides the options listed in *italics*; see “Options that `optimoptions` Hides” on page 17-65.

Options for `ga` and `gamultiobj`

Option	Description	Values
ConstraintTolerance	Determines the feasibility with respect to nonlinear constraints. Also, $\max(\text{sqrt}(\text{eps}), \text{ConstraintTolerance})$ determines feasibility with respect to linear constraints. For an options structure, use <code>TolCon</code> .	Positive scalar <code>{1e-3}</code>
CreationFcn	Function that creates the initial population. Specify as a name of a built-in creation function or a function handle. See “Population Options” on page 17-26.	<code>{'gacreationuniform'}</code> <code>{'gacreationlinearfeasible'}</code> * <code>'gacreationnonlinearfeasible'</code> <code>{'gacreationuniformint'}</code> I* for <code>ga</code> <code>{'gacreationsobol'}</code> I* for <code>gamultiobj</code> Custom creation function on page 17-26
CrossoverFcn	Function that the algorithm uses to create crossover children. Specify as a name of a built-in crossover function or a function handle. See “Crossover Options” on page 17-34.	<code>{'crossoverscattered'}</code> for <code>ga</code> , <code>{'crossoverintermediate'}</code> * for <code>gamultiobj</code> <code>{'crossoverlaplace'}</code> I* <code>'crossoverheuristic'</code> <code>'crossoversinglepoint'</code> <code>'crossovertwopoint'</code> <code>'crossoverarithmetic'</code> Custom crossover function on page 17-34
CrossoverFraction	The fraction of the population at the next generation, not including elite children, that the crossover function creates.	Positive scalar <code>{0.8}</code>
Display	Level of display.	<code>'off'</code> <code>'iter'</code> <code>'diagnose'</code> <code>{'final'}</code>
DistanceMeasureFcn	Function that computes the distance measure of individuals. Specify as a name of a built-in distance measure function or a function handle. The value applies to the decision variable or design space (genotype) or to function space (phenotype). The default <code>'distancecrowding'</code> is in function space (phenotype). For <code>gamultiobj</code> only. See “Multiobjective Options” on page 17-38. For an options structure, use a function handle, not a name.	<code>{'distancecrowding'}</code> means the same as <code>{@distancecrowding, 'phenotype'}</code> <code>{@distancecrowding, 'genotype'}</code> Custom distance function on page 17-38

Option	Description	Values
EliteCount	NM Positive integer specifying how many individuals in the current generation are guaranteed to survive to the next generation. Not used in <code>gamultiobj</code> .	Positive integer $\{\text{ceil}(0.05 * \text{PopulationSize})\}$ $\{0.05 * (\text{default PopulationSize})\}$ for mixed-integer problems
FitnessLimit	NM If the fitness function attains the value of <code>FitnessLimit</code> , the algorithm halts.	Scalar $\{-\text{Inf}\}$
FitnessScalingFcn	Function that scales the values of the fitness function. Specify as a name of a built-in scaling function or a function handle. Option unavailable for <code>gamultiobj</code> .	<code>'fitscalingrank'</code> <code>'fitscalingshiftlinear'</code> <code>'fitscalingprop'</code> <code>'fitscalingtop'</code> Custom fitness scaling function on page 17-29
FunctionTolerance	The algorithm stops if the average relative change in the best fitness function value over <code>MaxStallGenerations</code> generations is less than or equal to <code>FunctionTolerance</code> . If <code>StallTest</code> is <code>'geometricWeighted'</code> , then the algorithm stops if the weighted average relative change is less than or equal to <code>FunctionTolerance</code> . For <code>gamultiobj</code> , the algorithm stops when the geometric average of the relative change in value of the spread over <code>options.MaxStallGenerations</code> generations is less than <code>options.FunctionTolerance</code> , and the final spread is less than the mean spread over the past <code>options.MaxStallGenerations</code> generations. See “ <code>gamultiobj</code> Algorithm” on page 14-5. For an options structure, use <code>TolFun</code> .	Positive scalar $\{1e-6\}$ for <code>ga</code> , $\{1e-4\}$ for <code>gamultiobj</code>
HybridFcn	I* Function that continues the optimization after <code>ga</code> terminates. Specify as a name or a function handle. Alternatively, a cell array specifying the hybrid function and its options. See “ <code>ga</code> Hybrid Function” on page 17-39. For <code>gamultiobj</code> , the only hybrid function is <code>@fgoalattain</code> . See “ <code>gamultiobj</code> Hybrid Function” on page 17-40. When the problem has integer constraints, you cannot use a hybrid function. See “When to Use a Hybrid Function” on page 8-116.	Function name or handle <code>'fminsearch'</code> <code>'patternsearch'</code> <code>'fminunc'</code> <code>'fmincon'</code> <code>{[]}</code> or 1-by-2 cell array <code>{@solver, hybridoptions}</code> , where <code>solver</code> = <code>fminsearch</code> , <code>patternsearch</code> , <code>fminunc</code> , or <code>fmincon</code> <code>{[]}</code>

Option	Description	Values
<i>InitialPenalty</i>	NM I* Initial value of the penalty parameter	Positive scalar {10}
<i>InitialPopulationMatrix</i>	Initial population used to seed the genetic algorithm. Has up to <i>PopulationSize</i> rows and <i>N</i> columns, where <i>N</i> is the number of variables. You can pass a partial population, meaning one with fewer than <i>PopulationSize</i> rows. In that case, the genetic algorithm uses <i>CreationFcn</i> to generate the remaining population members. See “Population Options” on page 17-26. For an options structure, use <i>InitialPopulation</i> .	Matrix {}
<i>InitialPopulationRange</i>	Matrix or vector specifying the range of the individuals in the initial population. Applies to <i>gacreationuniform</i> creation function. <i>ga</i> shifts and scales the default initial range to match any finite bounds. For an options structure, use <i>PopInitRange</i> .	Matrix or vector {[-10;10]} for unbounded components, {[-1e4+1;1e4+1]} for unbounded components of integer-constrained problems, {[lb;ub]} for bounded components, with the default range modified to match one-sided bounds
<i>InitialScoresMatrix</i>	Initial scores used to determine fitness. Has up to <i>PopulationSize</i> rows and <i>Nf</i> columns, where <i>Nf</i> is the number of fitness functions (1 for <i>ga</i> , greater than 1 for <i>gamultiobj</i>). You can pass a partial scores matrix, meaning one with fewer than <i>PopulationSize</i> rows. In that case, the solver fills in the scores when it evaluates the fitness functions. For an options structure, use <i>InitialScores</i> .	Column vector for single objective matrix for multiobjective {}
<i>MaxGenerations</i>	Maximum number of iterations before the algorithm halts. For an options structure, use <i>Generations</i> .	Positive integer {100*numberOfVariables} for <i>ga</i> , {200*numberOfVariables} for <i>gamultiobj</i>

Option	Description	Values
MaxStallGenerations	<p>The algorithm stops if the average relative change in the best fitness function value over MaxStallGenerations generations is less than or equal to FunctionTolerance. If StallTest is 'geometricWeighted', then the algorithm stops if the weighted average relative change is less than or equal to FunctionTolerance.</p> <p>For gamultiobj, the algorithm stops when the geometric average of the relative change in value of the spread over options.MaxStallGenerations generations is less than options.FunctionTolerance, and the final spread is less than the mean spread over the past options.MaxStallGenerations generations. See “gamultiobj Algorithm” on page 14-5.</p> <p>For an options structure, use StallGenLimit.</p>	Positive integer {50} for ga, {100} for gamultiobj
MaxStallTime	<p>NM The algorithm stops if there is no improvement in the objective function for MaxStallTime seconds, as measured by tic and toc.</p> <p>For an options structure, use StallTimeLimit.</p>	Positive scalar {Inf}
MaxTime	<p>The algorithm stops after running for MaxTime seconds, as measured by tic and toc. This limit is enforced after each iteration, so ga can exceed the limit when an iteration takes substantial time.</p> <p>For an options structure, use TimeLimit.</p>	Positive scalar {Inf}
MigrationDirection	Direction of migration. See “Migration Options” on page 17-37.	'both' {'forward'}
MigrationFraction	Scalar from 0 through 1 specifying the fraction of individuals in each subpopulation that migrates to a different subpopulation. See “Migration Options” on page 17-37.	Scalar {0.2}
MigrationInterval	Positive integer specifying the number of generations that take place between migrations of individuals between subpopulations. See “Migration Options” on page 17-37.	Positive integer {20}

Option	Description	Values
MutationFcn	Function that produces mutation children. Specify as a name of a built-in mutation function or a function handle. See “Mutation Options” on page 17-31.	{'mutationgaussian'} for ga without constraints {'mutationadaptfeasible'}* for gamultiobj and for ga with constraints {'mutationpower'}I* 'mutationpositivebasis' 'mutationuniform' Custom mutation function on page 17-31
NonlinearConstraintAlgorithm	Nonlinear constraint algorithm. See “Nonlinear Constraint Solver Algorithms for Genetic Algorithm” on page 8-56. Option unchangeable for gamultiobj. For an options structure, use NonlinConAlgorithm.	{'auglag'} for ga, {'penalty'} for gamultiobj
OutputFcn	Functions that ga calls at each iteration. Specify as a function handle or a cell array of function handles. See “Output Function Options” on page 17-41. For an options structure, use OutputFcns.	Function handle or cell array of function handles {[]}
ParetoFraction	Scalar from 0 through 1 specifying the fraction of individuals to keep on the first Pareto front while the solver selects individuals from higher fronts, for gamultiobj only. See “Multiobjective Options” on page 17-38.	Scalar {0.35}
PenaltyFactor	NM I* Penalty update parameter.	Positive scalar {100}
PlotFcn	Function that plots data computed by the algorithm. Specify as a name of a built-in plot function, a function handle, or a cell array of built-in names or function handles. See “Plot Options” on page 17-23. For an options structure, use PlotFcns.	ga or gamultiobj: {[]} 'gaplotdistance' 'gaplotgenealogy' 'gaplotselection' 'gaplotscorediversity' 'gaplotscores' 'gaplotstopping' 'gaplotmaxconstr' Custom plot function on page 17-23 ga only: 'gaplotbestf' 'gaplotbestindiv' 'gaplotexpectation' 'gaplotrange' gamultiobj only: 'gaplotpareto' 'gaplotparetodistance' 'gaplotrankhist' 'gaplotspread'

Option	Description	Values
<i>PlotInterval</i>	Positive integer specifying the number of generations between consecutive calls to the plot functions.	Positive integer {1}
PopulationSize	Size of the population.	Positive integer {50} when numberOfVariables <= 5, {200} otherwise {min(max(10*nvars,40),100)} for mixed-integer problems
PopulationType	Data type of the population. Must be 'doubleVector' for mixed-integer problems.	'bitstring' 'custom' {'doubleVector'} ga ignores all constraints when PopulationType is set to 'bitString' or 'custom'. See "Population Options" on page 17-26.
SelectionFcn	Function that selects parents of crossover and mutation children. Specify as a name of a built-in selection function or a function handle. gamultiobj uses only 'selectiontournament'.	{'selectionstochunif'} for ga, {'selectiontournament'} for gamultiobj 'selectionremainder' 'selectionuniform' 'selectionroulette' Custom selection function on page 17-30
<i>StallTest</i>	NM Stopping test type.	'geometricWeighted' {'averageChange'}
UseParallel	Compute fitness and nonlinear constraint functions in parallel. See "Vectorize and Parallel Options (User Function Evaluation)" on page 17-43 and "How to Use Parallel Processing in Global Optimization Toolbox" on page 16-11.	true {false}
UseVectorized	Specifies whether functions are vectorized. See "Vectorize and Parallel Options (User Function Evaluation)" on page 17-43 and "Vectorize the Fitness Function" on page 8-103. For an options structure, use Vectorized with the values 'on' or 'off'.	true {false}

Output Arguments

options — Optimization options

structure

Optimization options, returned as a structure.

Version History

Introduced before R2006a

R2018b: gaoptimset is not recommended

Not recommended starting in R2018b

To set options, the `gaoptimset`, `psoptimset`, and `saoptimset` functions are not recommended. Instead, use `optimoptions`.

The main difference between using `optimoptions` and the other functions is that you include the solver name as the first argument in `optimoptions`. For example, to set an iterative display in `ga`:

```
options = optimoptions('ga','Display','iter');  
% instead of  
options = gaoptimset('Display','iter');
```

The other difference is that some option names have changed. You can continue to use the old names in `optimoptions`. For details, see “Options Changes in R2016a” on page 17-65.

`optimoptions` offers these advantages over the other functions:

- `optimoptions` provides better automatic code suggestions and completions, especially in the Live Editor.
- You can use a single option-setting function instead of a variety of functions.

There are no plans to remove `gaoptimset`, `psoptimset`, and `saoptimset` at this time.

See Also

`ga` | `gamultiobj` | `optimoptions`

Topics

“Genetic Algorithm Options” on page 17-23

GlobalOptimSolution

Optimization solution

Description

A `GlobalOptimSolution` object contains information on a local minimum, including location, objective function value, and start point or points that lead to the minimum.

`GlobalSearch` and `MultiStart` generate a vector of `GlobalOptimSolution` objects. The vector is ordered by objective function value, from lowest (best) to highest (worst). `GlobalSearch` and `MultiStart` combine solutions that coincide with previously found solutions to within tolerances. For `GlobalSearch` details, see **Update Solution Set** in “When `fmincon` Runs” on page 4-37. For `MultiStart` details, see “Create `GlobalOptimSolution` Object” on page 4-39.

Creation

When you execute `run` and request the “solutions” on page 18-0 output, `GlobalSearch` and `MultiStart` create `GlobalOptimSolution` objects as output.

Properties

Exitflag — Exit condition of local solver

integer

Exit condition of the local solver, returned as an integer. Generally, a positive `Exitflag` corresponds to a local optimum, and a zero or negative `Exitflag` corresponds to an unsuccessful search for a local minimum.

For the exact meaning of each `Exitflag`, see the `exitflag` description in the appropriate local solver function reference page:

- `fmincon` `exitflag`
- `fminunc` `exitflag`
- `lsqcurvefit` `exitflag`
- `lsqnonlin` `exitflag`

Data Types: double

Fval — Objective function value

real scalar

Objective function value, returned as a real scalar. For the `lsqnonlin` and `lsqcurvefit` solvers, `Fval` is the sum of squares of the residual.

Data Types: double

Output — Output structure returned by the local solver

structure

Output structure returned by the local solver. For details, see the `output` description in the appropriate local solver function reference page:

- `fmincon` output
- `fminunc` output
- `lsqcurvefit` output
- `lsqnonlin` output

Data Types: `struct`

X — Local solution

array

Local solution, returned as an array with the same dimensions as `problem.x0`.

Data Types: `double`

X0 — Start points that lead to current solution

cell array

Start points that lead to current solution, returned as a cell array. Control the distance between points considered as distinct by setting the `FunctionTolerance` and `XTolerance` properties of the global solver.

Data Types: `cell`

Examples

Obtain Multiple Local Solutions

Use `MultiStart` to create a vector of `GlobalOptimSolution` objects for a problem with multiple local minima.

```
rng default % For reproducibility
ms = MultiStart;
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3]);
[xmin,fmin,flag,outpt,allmins] = run(ms,problem,30);
```

`MultiStart` completed the runs from all start points.

All 30 local solver runs converged with a positive local solver exit flag.

`allmins` is a vector of `GlobalOptimSolution` objects.

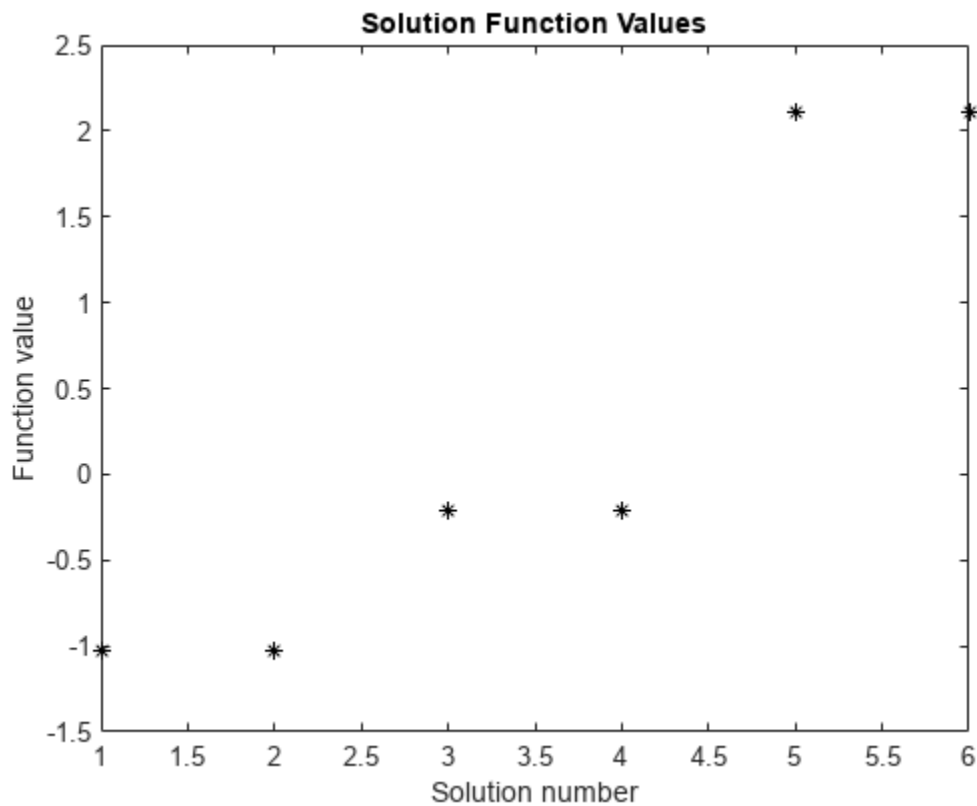
```
disp(allmins)
```

```
1x6 GlobalOptimSolution array with properties:
```

```
 X
 Fval
 Exitflag
 Output
 X0
```

Plot the objective function values at the returned solutions.

```
plot(arrayfun(@(x)x.Fval,allmins),'k*')  
xlabel('Solution number')  
ylabel('Function value')  
title('Solution Function Values')
```



To examine the initial points that lead to the various solutions, see “Visualize the Basins of Attraction” on page 4-24.

Version History

Introduced in R2010a

See Also

MultiStart | run | GlobalSearch

Topics

“Visualize the Basins of Attraction” on page 4-24

“How GlobalSearch and MultiStart Work” on page 4-34

GlobalSearch

Find global minimum

Description

A GlobalSearch object contains properties (options) that affect how run repeatedly runs a local solver to generate a GlobalOptimSolution object. When run, the solver attempts to locate a solution that has the lowest objective function value.

Creation

Syntax

```
gs = GlobalSearch
gs = GlobalSearch(Name,Value)
gs = GlobalSearch(oldGS,Name,Value)
gs = GlobalSearch(ms)
```

Description

`gs = GlobalSearch` creates `gs`, a GlobalSearch solver with its properties set to the defaults.

`gs = GlobalSearch(Name,Value)` sets properties using name-value pairs.

`gs = GlobalSearch(oldGS,Name,Value)` creates a copy of the `oldGS` GlobalSearch solver, and sets properties using name-value pairs.

`gs = GlobalSearch(ms)` creates `gs`, a GlobalSearch solver, with common property values from the `ms` MultiStart solver.

Properties

BasinRadiusFactor — Basin radius decrease factor

0.2 (default) | scalar from 0 through 1

Basin radius decrease factor, specified as a scalar from 0 through 1. A basin radius decreases after `MaxWaitCycle` consecutive start points are within the basin. The basin radius decreases by a factor of $1 - \text{BasinRadiusFactor}$.

Set `BasinRadiusFactor` to 0 to disable updates of the basin radius.

Example: 0.5

Data Types: double

Display — Level of display to the Command Window

'final' (default) | 'iter' | 'off'

Level of display to the Command Window, specified as one of the following character vectors or strings:

- 'final' - Report summary results after run finishes.
- 'iter' - Report results after the initial `fmincon` run, after Stage 1, after every 200 start points, and after every run of `fmincon`, in addition to the final summary.
- 'off' - No display.

Example: 'iter'

Data Types: char | string

DistanceThresholdFactor — Multiplier for determining trial point is in existing basin

0.75 (default) | nonnegative scalar

Multiplier for determining whether a trial point is in an existing basin of attraction, specified as a nonnegative scalar. For details, see “Examine Stage 2 Trial Point to See if `fmincon` Runs” on page 4-36.

Example: 0.5

Data Types: double

FunctionTolerance — Tolerance on function values for considering solutions equal

1e-6 (default) | nonnegative scalar

Tolerance on function values for considering solutions equal, specified as a nonnegative scalar. Solvers consider two solutions identical if they are within `XTolerance` relative distance of each other and have objective function values within `FunctionTolerance` relative difference of each other. If both conditions are not met, solvers report the solutions as distinct. Set `FunctionTolerance` to 0 to obtain the results of every local solver run. Set `FunctionTolerance` to a larger value to have fewer results. For `GlobalSearch` details, see **Update Solution Set** in “When `fmincon` Runs” on page 4-37. For `MultiStart` details, see “Create `GlobalOptimSolution` Object” on page 4-39.

Example: 1e-4

Data Types: double

MaxTime — Maximum time in seconds that GlobalSearch runs

Inf (default) | positive scalar

Maximum time in seconds that `GlobalSearch` runs, specified as a positive scalar. `GlobalSearch` and its local solvers halt when `MaxTime` seconds have passed since the beginning of the run, as measured by `tic` and `toc`.

`MaxTime` does not interrupt local solvers during a run, so the total time can exceed `MaxTime`.

Example: 180 stops the solver the first time a local solver call finishes after 180 seconds.

Data Types: double

MaxWaitCycle — Algorithm control parameter

20 (default) | positive integer

Algorithm control parameter, specified as a positive integer.

- If the observed penalty function of `MaxWaitCycle` consecutive trial points is at least the penalty threshold, then raise the penalty threshold (see `PenaltyThresholdFactor`).

- If `MaxWaitCycle` consecutive trial points are in a basin, then update that basin's radius (see `BasinRadiusFactor`).

Example: 40

Data Types: double

NumStageOnePoints — Number of Stage 1 points

200 (default) | positive integer

Number of Stage 1 points, specified as a positive integer. For details, see “Obtain Stage 1 Start Point, Run” on page 4-36.

Example: 1000

Data Types: double

NumTrialPoints — Number of potential start points

1000 (default) | positive integer

Number of potential start points, specified as a positive integer.

Example: 3e4

Data Types: double

OutputFcn — Report on solver progress or halt solver

[] (default) | function handle | cell array of function handles

Report on solver progress or halt solver, specified as a function handle or cell array of function handles. Output functions run after each local solver call. They also run when the global solver starts and ends. Write output functions using the syntax described in “OutputFcn” on page 17-3. See “GlobalSearch Output Function” on page 4-27.

Data Types: cell | function_handle

PenaltyThresholdFactor — Increase in penalty threshold

0.2 (default) | positive scalar

Increase in the penalty threshold, specified as a positive scalar. For details, see React to Large Counter Values on page 4-38.

Example: 0.4

Data Types: double

PlotFcn — Plot solver progress

[] (default) | function handle | cell array of function handles

Plot solver progress, specified as a function handle or cell array of function handles. Plot functions run after each local solver call. They also run when the global solver starts and ends. Write plot functions using the syntax described in “OutputFcn” on page 17-3.

There are two built-in plot functions:

- `@gsplotbestf` plots the best objective function value.
- `@gsplotfunccount` plots the number of function evaluations.

See “MultiStart Plot Function” on page 4-30.

Example: @gsplotbestf

Data Types: cell | function_handle

StartPointsToRun — Start points to run

'all' (default) | 'bounds' | 'bounds-ineqs'

Start points to run, specified as:

- 'all' — Run all start points.
- 'bounds' — Run only start points that satisfy bounds.
- 'bounds-ineqs' — Run only start points that satisfy bounds and inequality constraints.

GlobalSearch checks the StartPointsToRun property only during Stage 2 of the GlobalSearch algorithm (the main loop). For more information, see “GlobalSearch Algorithm” on page 4-35.

Example: 'bounds' runs only points that satisfy all bounds.

Data Types: char | string

XTolerance — Tolerance on distance for considering solutions equal

1e-6 (default) | nonnegative scalar

Tolerance on distance for considering solutions equal, specified as a nonnegative scalar. Solvers consider two solutions identical if they are within XTolerance relative distance of each other and have objective function values within FunctionTolerance relative difference of each other. If both conditions are not met, solvers report the solutions as distinct. Set XTolerance to 0 to obtain the results of every local solver run. Set XTolerance to a larger value to have fewer results. For GlobalSearch details, see **Update Solution Set** in “When fmincon Runs” on page 4-37. For MultiStart details, see “Create GlobalOptimSolution Object” on page 4-39.

Example: 2e-4

Data Types: double

Object Functions

run Run multiple-start solver

Examples

Run GlobalSearch on Multidimensional Problem

Create an optimization problem that has several local minima, and try to find the global minimum using GlobalSearch. The objective is the six-hump camel back problem (see “Run the Solver” on page 4-13).

```
rng default % For reproducibility
gs = GlobalSearch;
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3]);
x = run(gs,problem)
```

GlobalSearch stopped because it analyzed all the trial points.

All 8 local solver runs converged with a positive local solver exit flag.

```
x = 1x2
```

```
-0.0898    0.7127
```

You can request the objective function value at x when you call run by using the following syntax:

```
[x,fval] = run(gs,problem)
```

However, if you neglected to request fval, you can still compute the objective function value at x.

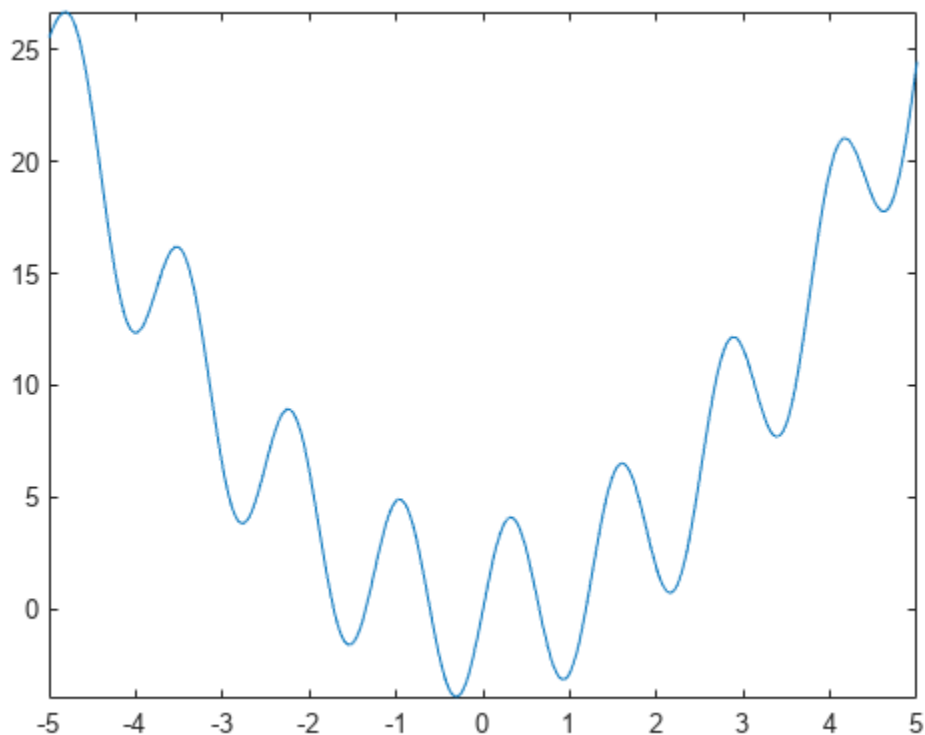
```
fval = sixmin(x)
```

```
fval = -1.0316
```

Run GlobalSearch on 1-D Problem

Consider a function with several local minima.

```
fun = @(x) x.^2 + 4*sin(5*x);  
fplot(fun,[-5,5])
```



To search for the global minimum, run `GlobalSearch` using the `fmincon` 'sqp' algorithm.

```
rng default % For reproducibility
opts = optimoptions(@fmincon,'Algorithm','sqp');
problem = createOptimProblem('fmincon','objective',...
    fun,'x0',3,'lb',-5,'ub',5,'options',opts);
gs = GlobalSearch;
[x,f] = run(gs,problem)
```

`GlobalSearch` stopped because it analyzed all the trial points.

All 23 local solver runs converged with a positive local solver exit flag.

```
x = -0.3080
```

```
f = -3.9032
```

GlobalSearch Using Common Properties from MultiStart

Create a nondefault `MultiStart` object.

```
ms = MultiStart('FunctionTolerance',2e-4,'UseParallel',true)
```

```
ms =
  MultiStart with properties:

    UseParallel: 1
      Display: 'final'
FunctionTolerance: 2.0000e-04
      MaxTime: Inf
      OutputFcn: []
      PlotFcn: []
StartPointsToRun: 'all'
      XTolerance: 1.0000e-06
```

Create a `GlobalSearch` object that uses the available properties from `ms`.

```
gs = GlobalSearch(ms)
```

```
gs =
  GlobalSearch with properties:

      NumTrialPoints: 1000
      BasinRadiusFactor: 0.2000
DistanceThresholdFactor: 0.7500
      MaxWaitCycle: 20
      NumStageOnePoints: 200
PenaltyThresholdFactor: 0.2000
      Display: 'final'
FunctionTolerance: 2.0000e-04
      MaxTime: Inf
      OutputFcn: []
      PlotFcn: []
StartPointsToRun: 'all'
      XTolerance: 1.0000e-06
```

gs has the same nondefault value of `FunctionTolerance` as `ms`. But `gs` does not use the `UseParallel` property.

Update GlobalSearch Properties

Create a `GlobalSearch` object with a `FunctionTolerance` of $1e-4$.

```
gs = GlobalSearch('FunctionTolerance',1e-4)
```

```
gs =  
GlobalSearch with properties:  
  
    NumTrialPoints: 1000  
    BasinRadiusFactor: 0.2000  
    DistanceThresholdFactor: 0.7500  
        MaxWaitCycle: 20  
    NumStageOnePoints: 200  
    PenaltyThresholdFactor: 0.2000  
        Display: 'final'  
    FunctionTolerance: 1.0000e-04  
        MaxTime: Inf  
        OutputFcn: []  
        PlotFcn: []  
    StartPointsToRun: 'all'  
        XTolerance: 1.0000e-06
```

Update the `XTolerance` property to $1e-3$ and the `StartPointsToRun` property to `'bounds'`.

```
gs = GlobalSearch(gs, 'XTolerance',1e-3, 'StartPointsToRun', 'bounds')
```

```
gs =  
GlobalSearch with properties:  
  
    NumTrialPoints: 1000  
    BasinRadiusFactor: 0.2000  
    DistanceThresholdFactor: 0.7500  
        MaxWaitCycle: 20  
    NumStageOnePoints: 200  
    PenaltyThresholdFactor: 0.2000  
        Display: 'final'  
    FunctionTolerance: 1.0000e-04  
        MaxTime: Inf  
        OutputFcn: []  
        PlotFcn: []  
    StartPointsToRun: 'bounds'  
        XTolerance: 1.0000e-03
```

You can also update properties one at a time by using dot notation.

```
gs.MaxTime = 1800
```

```
gs =  
GlobalSearch with properties:
```

```
    NumTrialPoints: 1000
    BasinRadiusFactor: 0.2000
DistanceThresholdFactor: 0.7500
    MaxWaitCycle: 20
    NumStageOnePoints: 200
PenaltyThresholdFactor: 0.2000
    Display: 'final'
FunctionTolerance: 1.0000e-04
    MaxTime: 1800
    OutputFcn: []
    PlotFcn: []
StartPointsToRun: 'bounds'
XTolerance: 1.0000e-03
```

Algorithms

For a detailed description of the algorithm, see “GlobalSearch Algorithm” on page 4-35. Ugray et al. [1] describe both the algorithm and the scatter-search method of generating trial points.

Version History

Introduced in R2010a

References

- [1] Ugray, Zsolt, Leon Lasdon, John Plummer, Fred Glover, James Kelly, and Rafael Martí. *Scatter Search and Local NLP Solvers: A Multistart Framework for Global Optimization*. INFORMS Journal on Computing, Vol. 19, No. 3, 2007, pp. 328-340.

See Also

GlobalOptimSolution | MultiStart | run

Topics

- “Example of Run with GlobalSearch” on page 4-13
“Workflow for GlobalSearch and MultiStart” on page 4-3

list

List start points

Syntax

```
points = list(tpoints)
points = list(rs,problem)
```

Description

`points = list(tpoints)` returns the points inside the `tpoints` `CustomStartPointSet` object.

`points = list(rs,problem)` generates and returns points described by the `rs` `RandomStartPointSet` object and `problem`.

Examples

Create CustomStartPointSet

Create a `CustomStartPointSet` object with 64 three-dimensional points.

```
[x,y,z] = meshgrid(1:4);
ptmatrix = [x(:),y(:),z(:)] + [10,20,30];
tpoints = CustomStartPointSet(ptmatrix);
```

`tpoints` is the `ptmatrix` matrix contained in a `CustomStartPointSet` object.

Extract the original matrix from the `tpoints` object by using `list`.

```
tpts = list(tpoints);
```

Check that the `tpts` output is identical to `ptmatrix`.

```
isequal(ptmatrix,tpts)
```

```
ans = logical
      1
```

Create RandomStartPointSet

Create a `RandomStartPointSet` object for 40 points.

```
rs = RandomStartPointSet('NumStartPoints',40);
```

Create a problem with 3-D variables, lower bounds of 0, and upper bounds of [10,20,30].

```
problem = createOptimProblem('fmincon','x0',rand(3,1),'lb',zeros(3,1),'ub',[10,20,30]);
```

Generate a random set of 40 points consistent with the problem.

```
points = list(rs,problem);
```

Examine the maximum and minimum generated components.

```
largest = max(max(points))
```

```
largest = 29.8840
```

```
smallest = min(min(points))
```

```
smallest = 0.1390
```

Input Arguments

tpoints — Start points

CustomStartPointSet object

Start points, specified as a CustomStartPointSet object. `list` extracts the points into a matrix where each row is one start point.

Example: `tpoints = CustomStartPointSet([1:5;4:8].^2)`

rs — Start points description

RandomStartPointSet object

Start points description, specified as a RandomStartPointSet object. `list` generates start points using the NumStartPoints (number of points) and ArtificialBound (artificial bounds) properties of `rs`. `list` uses the `x0` field in `problem` to determine the number of variables in the start points. `list` uses the bounds in `problem` as follows:

- `list` generates points uniformly within bounds.
- If a component has no bounds, `list` uses a lower bound of `-ArtificialBound` and an upper bound of `ArtificialBound`.
- If a component has a lower bound `lb` but no upper bound, `list` uses an upper bound of `lb + 2*ArtificialBound`.
- Similarly, if a component has an upper bound `ub` but no lower bound, `list` uses a lower bound of `ub - 2*ArtificialBound`.

problem — Problem description

problem structure

Problem description, specified as a problem structure. Create a problem structure using `createOptimProblem`. `list` uses only the lower and upper bounds in `problem`, as described in `rs`, and uses the `x0` field in `problem` to determine the number of variables.

Data Types: `struct`

Output Arguments

points — Start points

k-by-n matrix

Start points, returned as a k-by-n matrix. Each row of the matrix represents one start point.

- If you list a `CustomStartPointSet`, then k is the `NumStartPoints` property, and n is the `StartPointsDimension` property.
- If you list a `RandomStartPointSet`, then k is the `NumStartPoints` property, and n is inferred from the `x0` field of the problem structure.

Version History

Introduced in R2010a

See Also

`CustomStartPointSet` | `RandomStartPointSet` | `MultiStart`

Topics

“Workflow for `GlobalSearch` and `MultiStart`” on page 4-3

MultiStart

Find multiple local minima

Description

A `MultiStart` object contains properties (options) that affect how `run` repeatedly runs a local solver to generate a `GlobalOptimSolution` object. When run, the solver attempts to find multiple local solutions to a problem by starting from various points.

Creation

Syntax

```
ms = MultiStart
ms = MultiStart(Name,Value)
ms = MultiStart(oldMS,Name,Value)
ms = MultiStart(gs)
```

Description

`ms = MultiStart` creates `ms`, a `MultiStart` solver with its properties set to the defaults.

`ms = MultiStart(Name,Value)` sets properties using name-value pairs.

`ms = MultiStart(oldMS,Name,Value)` creates a copy of the `oldMS` `MultiStart` solver, and sets properties using name-value pairs.

`ms = MultiStart(gs)` creates `ms`, a `MultiStart` solver, with common parameter values from the `gs` `GlobalSearch` solver.

Properties

Display — Level of display to the Command Window

```
'final' (default) | 'iter' | 'off'
```

Level of display to the Command Window, specified as one of the following character vectors or strings:

- `'final'` - Report summary results after run finishes.
- `'iter'` - Report results after the initial `fmincon` run, after Stage 1, after every 200 start points, and after every run of `fmincon`, in addition to the final summary.
- `'off'` - No display.

Example: `'iter'`

Data Types: `char` | `string`

FunctionTolerance — Tolerance on function values for considering solutions equal

1e-6 (default) | nonnegative scalar

Tolerance on function values for considering solutions equal, specified as a nonnegative scalar. Solvers consider two solutions identical if they are within `XTolerance` relative distance of each other and have objective function values within `FunctionTolerance` relative difference of each other. If both conditions are not met, solvers report the solutions as distinct. Set `FunctionTolerance` to 0 to obtain the results of every local solver run. Set `FunctionTolerance` to a larger value to have fewer results. For `GlobalSearch` details, see **Update Solution Set** in “When `fmincon` Runs” on page 4-37. For `MultiStart` details, see “Create `GlobalOptimSolution` Object” on page 4-39.

Example: 1e-4

Data Types: double

MaxTime — Maximum time in seconds that MultiStart runs

Inf (default) | positive scalar

Maximum time in seconds that `MultiStart` runs, specified as a positive scalar. `MultiStart` and its local solvers halt when `MaxTime` seconds have passed since the beginning of the run, as measured by `tic` and `toc`.

`MaxTime` does not interrupt local solvers during a run, so the total time can exceed `MaxTime`.

Example: 180 stops the solver the first time a local solver call finishes after 180 seconds.

Data Types: double

OutputFcn — Report on solver progress or halt solver

[] (default) | function handle | cell array of function handles

Report on solver progress or halt solver, specified as a function handle or cell array of function handles. Output functions run after each local solver call. They also run when the global solver starts and ends. Write output functions using the syntax described in “`OutputFcn`” on page 17-3. See “`GlobalSearch` Output Function” on page 4-27.

Data Types: cell | function_handle

PlotFcn — Plot solver progress

[] (default) | function handle | cell array of function handles

Plot solver progress, specified as a function handle or cell array of function handles. Plot functions run after each local solver call. They also run when the global solver starts and ends. Write plot functions using the syntax described in “`OutputFcn`” on page 17-3.

There are two built-in plot functions:

- `@gsplotbestf` plots the best objective function value.
- `@gsplotfuncount` plots the number of function evaluations.

See “`MultiStart` Plot Function” on page 4-30.

Example: `@gsplotbestf`

Data Types: cell | function_handle

StartPointsToRun — Start points to run

'all' (default) | 'bounds' | 'bounds-ineqs'

Start points to run, specified as:

- 'all' — Run all start points.
- 'bounds' — Run only start points that satisfy bounds.
- 'bounds-ineqs' — Run only start points that satisfy bounds and inequality constraints.

Example: 'bounds' runs only points that satisfy all bounds.

Data Types: char | string

UseParallel — Distribute local solver calls to multiple processors

false (default) | true

Distribute local solver calls to multiple processors, specified as false or true.

- false — Do not run in parallel.
- true — Distribute the local solver calls to multiple processors.

Example: true

Data Types: logical

XTolerance — Tolerance on distance for considering solutions equal

1e-6 (default) | nonnegative scalar

Tolerance on distance for considering solutions equal, specified as a nonnegative scalar. Solvers consider two solutions identical if they are within `XTolerance` relative distance of each other and have objective function values within `FunctionTolerance` relative difference of each other. If both conditions are not met, solvers report the solutions as distinct. Set `XTolerance` to 0 to obtain the results of every local solver run. Set `XTolerance` to a larger value to have fewer results. For `GlobalSearch` details, see **Update Solution Set** in “When `fmincon` Runs” on page 4-37. For `MultiStart` details, see “Create `GlobalOptimSolution` Object” on page 4-39.

Example: 2e-4

Data Types: double

Object Functions

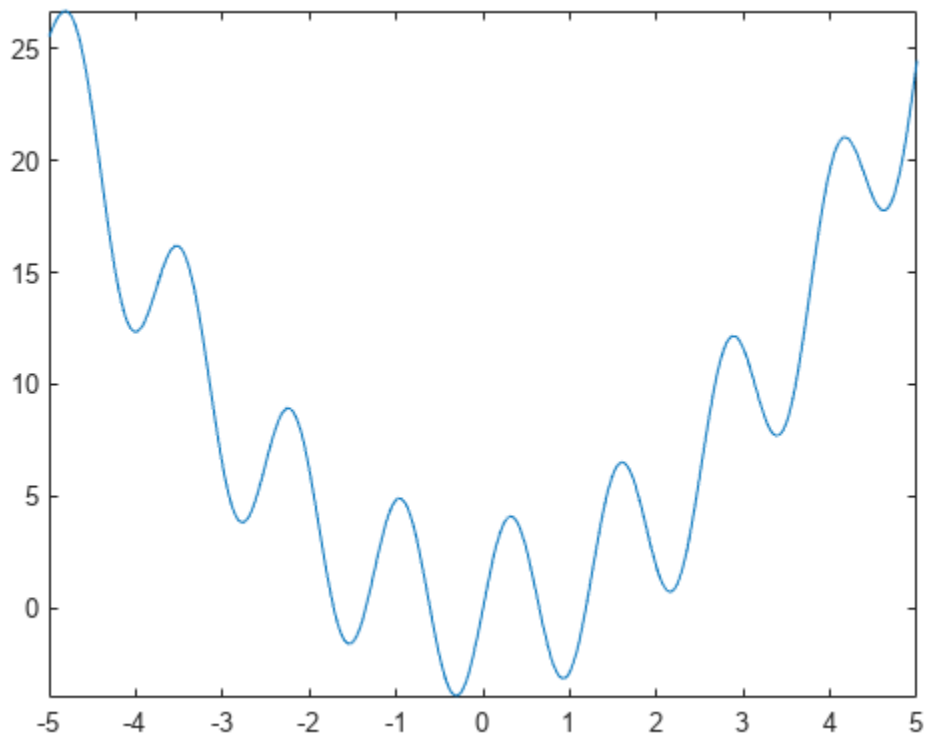
`run` Run multiple-start solver

Examples

Run MultiStart

Consider a function with several local minima.

```
fun = @(x) x.^2 + 4*sin(5*x);  
fplot(fun, [-5,5])
```



To search for the global minimum, run MultiStart on 20 instances of the problem using the `fmincon` 'sqp' algorithm.

```
rng default % For reproducibility
opts = optimoptions(@fmincon,'Algorithm','sqp');
problem = createOptimProblem('fmincon','objective',...
    fun,'x0',3,'lb',-5,'ub',5,'options',opts);
ms = MultiStart;
[x,f] = run(ms,problem,20)
```

MultiStart completed the runs from all start points.

All 20 local solver runs converged with a positive local solver exit flag.

```
x = -0.3080
```

```
f = -3.9032
```

Default MultiStart Object

Create a MultiStart object with default properties.

```
ms = MultiStart
```

```
ms =
    MultiStart with properties:
```

```

    UseParallel: 0
      Display: 'final'
FunctionTolerance: 1.0000e-06
    MaxTime: Inf
    OutputFcn: []
    PlotFcn: []
StartPointsToRun: 'all'
    XTolerance: 1.0000e-06

```

Nondefault MultiStart Object

Create a `MultiStart` object with looser tolerances than default, so the solver returns fewer solutions that are close to each other. Also, have `MultiStart` run only initial points that are feasible with respect to bounds and inequality constraints.

```
ms = MultiStart('FunctionTolerance',2e-4,'XTolerance',5e-3,...
    'StartPointsToRun','bounds-ineqs')
```

```
ms =
  MultiStart with properties:

    UseParallel: 0
      Display: 'final'
FunctionTolerance: 2.0000e-04
    MaxTime: Inf
    OutputFcn: []
    PlotFcn: []
StartPointsToRun: 'bounds-ineqs'
    XTolerance: 0.0050

```

MultiStart Using Common Properties from GlobalSearch

Create a nondefault `GlobalSearch` object.

```
gs = GlobalSearch('FunctionTolerance',2e-4,'NumTrialPoints',2000)
```

```
gs =
  GlobalSearch with properties:

    NumTrialPoints: 2000
    BasinRadiusFactor: 0.2000
DistanceThresholdFactor: 0.7500
    MaxWaitCycle: 20
    NumStageOnePoints: 200
PenaltyThresholdFactor: 0.2000
      Display: 'final'
FunctionTolerance: 2.0000e-04
    MaxTime: Inf
    OutputFcn: []
    PlotFcn: []

```

```

StartPointsToRun: 'all'
XTolerance: 1.0000e-06

```

Create a `MultiStart` object that uses the available properties from `gs`.

```
ms = MultiStart(gs)
```

```

ms =
  MultiStart with properties:

    UseParallel: 0
      Display: 'final'
FunctionTolerance: 2.0000e-04
      MaxTime: Inf
      OutputFcn: []
      PlotFcn: []
StartPointsToRun: 'all'
      XTolerance: 1.0000e-06

```

`ms` has the same nondefault value of `FunctionTolerance` as `gs`. But `ms` does not use the `NumTrialPoints` property.

Update MultiStart Properties

Create a `MultiStart` object with a `FunctionTolerance` of `1e-4`.

```
ms = MultiStart('FunctionTolerance',1e-4)
```

```

ms =
  MultiStart with properties:

    UseParallel: 0
      Display: 'final'
FunctionTolerance: 1.0000e-04
      MaxTime: Inf
      OutputFcn: []
      PlotFcn: []
StartPointsToRun: 'all'
      XTolerance: 1.0000e-06

```

Update the `XTolerance` property to `1e-3`, and the `StartPointsToRun` property to `'bounds'`.

```
ms = MultiStart(ms, 'XTolerance',1e-3, 'StartPointsToRun', 'bounds')
```

```

ms =
  MultiStart with properties:

    UseParallel: 0
      Display: 'final'
FunctionTolerance: 1.0000e-04
      MaxTime: Inf
      OutputFcn: []
      PlotFcn: []

```

```
StartPointsToRun: 'bounds'  
XTolerance: 1.0000e-03
```

You can also update properties one at a time by using dot notation.

```
ms.MaxTime = 1800
```

```
ms =  
  MultiStart with properties:  
  
    UseParallel: 0  
      Display: 'final'  
FunctionTolerance: 1.0000e-04  
      MaxTime: 1800  
    OutputFcn: []  
      PlotFcn: []  
StartPointsToRun: 'bounds'  
    XTolerance: 1.0000e-03
```

Algorithms

For a detailed description of the algorithm, see “MultiStart Algorithm” on page 4-38.

Version History

Introduced in R2010a

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set the 'UseParallel' option to true.

```
options = optimoptions('solvername','UseParallel',true)
```

For more information, see “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11.

See Also

[GlobalSearch](#) | [GlobalOptimSolution](#) | [CustomStartPointSet](#) | [RandomStartPointSet](#) | [run](#)

Topics

“Global or Multiple Starting Point Search”

“Parallel Computing”

“Workflow for GlobalSearch and MultiStart” on page 4-3

packfcn

Combine objective and nonlinear constraint functions

Syntax

```
objconstr = packfcn(obj,nlconst)
```

Description

`objconstr = packfcn(obj,nlconst)` combines the objective function `obj` and nonlinear constraint function `nlconst` into a function `objconstr`. The function `objconstr(x)` returns a structure suitable for a combined `surrogateopt` objective and constraint function. For information on converting between the `surrogateopt` structure syntax and other solvers, see “Convert Nonlinear Constraints Between `surrogateopt` Form and Other Solver Forms” on page 11-74.

Examples

Combine Objective and Constraint

Combine the objective and constraint from the example “Constrained Nonlinear Problem Using Optimize Live Editor Task or Solver” into a form suitable for `surrogateopt`.

Create the objective function as an anonymous function `ros(x)`.

```
ros = @(x)100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;
```

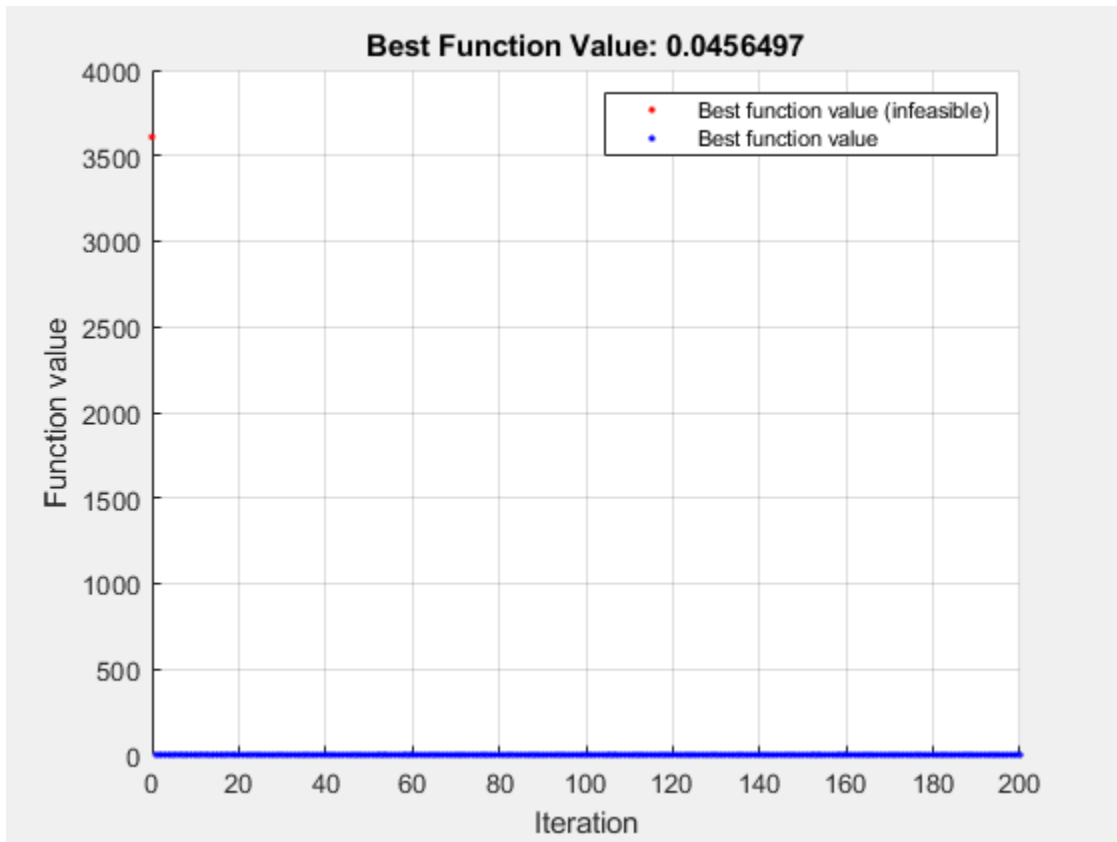
Create the nonlinear constraint helper function `unitdisk`, which appears at the end of this example on page 18-98. Save the helper function with the name `unitdisk.m` in the current folder.

Combine the objective and nonlinear constraint functions into one function suitable for `surrogateopt`.

```
objconstr = packfcn(ros,@unitdisk);
```

Specify bounds and solve the problem using `surrogateopt`.

```
lb = [-2 -2];
ub = -lb;
[x,fval] = surrogateopt(objconstr,lb,ub)
```



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
x = 1x2
```

```
    0.7870    0.6177
```

```
fval = 0.0456
```

This code creates the unitdisk helper function.

```
function [c,ceq] = unitdisk(x)
c = x(1)^2 + x(2)^2 - 1;
ceq = [ ];
end
```

Input Arguments

obj — Objective function

function handle | function name

Objective function, specified as a function handle or function name.

The resulting function objconstr contains the field Fval.

```
objconstr.Fval = obj
```

Data Types: char | string | function_handle

nlconst — Nonlinear constraint function

function handle | function name

Nonlinear constraint function, specified as a function handle or function name. Generally, the nonlinear constraint function returns two outputs.

```
[c,ceq] = nlconst(x)
```

The output `c` is a vector or array whose entries represent the inequality constraints $c(x) \leq 0$. The output `ceq` is a vector or array whose entries represent the equality constraints $c(x) = 0$. `packfcn` discards the `ceq` output.

The resulting function `objconstr` contains the field `Ineq`.

```
objconstr.Ineq = c
```

Data Types: char | string | function_handle

Output Arguments

objconstr — Combined objective and constraint function

function handle

Combined objective and constraint function, returned as a function handle. The function `objconstr(x)` returns a structure with the fields `Fval` and `Ineq`.

- `objconstr.Fval(x)` is the objective function `obj(x)`.
- `objconstr.Ineq(x)` is the nonlinear inequality constraint function `c(x)`, the first output of `nlconst(x)`.

Version History

Introduced in R2020a

See Also

`surrogateopt`

Topics

“Convert Nonlinear Constraints Between `surrogateopt` Form and Other Solver Forms” on page 11-74

paretosearch

Find points in Pareto set

Syntax

```
x = paretosearch(fun,nvars)
x = paretosearch(fun,nvars,A,b)
x = paretosearch(fun,nvars,A,b,Aeq,beq)
x = paretosearch(fun,nvars,A,b,Aeq,beq,lb,ub)
x = paretosearch(fun,nvars,A,b,Aeq,beq,lb,ub,nonlcon)
x = paretosearch(fun,nvars,A,b,Aeq,beq,lb,ub,nonlcon,options)
x = paretosearch(problem)
[x,fval] = paretosearch(____)
[x,fval,exitflag,output] = paretosearch(____)
[x,fval,exitflag,output,residuals] = paretosearch(____)
```

Description

`x = paretosearch(fun,nvars)` finds nondominated points of the multiobjective function `fun`. The `nvars` argument is the dimension of the optimization problem (number of decision variables).

`x = paretosearch(fun,nvars,A,b)` finds nondominated points subject to the linear inequalities $A*x \leq b$. See “Linear Inequality Constraints”.

`x = paretosearch(fun,nvars,A,b,Aeq,beq)` finds nondominated points subject to the linear constraints $Aeq*x = beq$ and $A*x \leq b$. If no linear inequalities exist, set `A = []` and `b = []`.

`x = paretosearch(fun,nvars,A,b,Aeq,beq,lb,ub)` defines a set of lower and upper bounds on the design variables in `x`, so that `x` is always in the range $lb \leq x \leq ub$. If no linear equalities exist, set `Aeq = []` and `beq = []`. If `x(i)` has no lower bound, set `lb(i) = -Inf`. If `x(i)` has no upper bound, set `ub(i) = Inf`.

`x = paretosearch(fun,nvars,A,b,Aeq,beq,lb,ub,nonlcon)` applies the nonlinear inequalities `c(x)` defined in `nonlcon`. The `paretosearch` function finds nondominated points such that $c(x) \leq 0$. If no bounds exist, set `lb = []`, `ub = []`, or both.

Note Currently, `paretosearch` does not support nonlinear equality constraints $ceq(x) = 0$.

`x = paretosearch(fun,nvars,A,b,Aeq,beq,lb,ub,nonlcon,options)` finds nondominated points with the optimization options specified in `options`. Use `optimoptions` to set these options. If there are no nonlinear inequality or equality constraints, set `nonlcon = []`.

`x = paretosearch(problem)` finds the nondominated points for `problem`, where `problem` is a structure described in `problem`.

`[x,fval] = paretosearch(____)`, for any input variables, returns the matrix `fval`, the value of all the objective functions in `fun` for all the solutions (rows) in `x`. The output `fval` has `nf` columns, where `nf` is the number of objectives, and has the same number of rows as `x`.

`[x,fval,exitflag,output] = paretosearch(___)` also returns `exitflag`, an integer identifying the reason the algorithm stopped, and `output`, a structure that contains information about the solution process.

`[x,fval,exitflag,output,residuals] = paretosearch(___)` also returns `residuals`, a structure containing the constraint values at the solution points `x`.

Examples

Find Pareto Front

Find points on the Pareto front of a two-objective function of a two-dimensional variable.

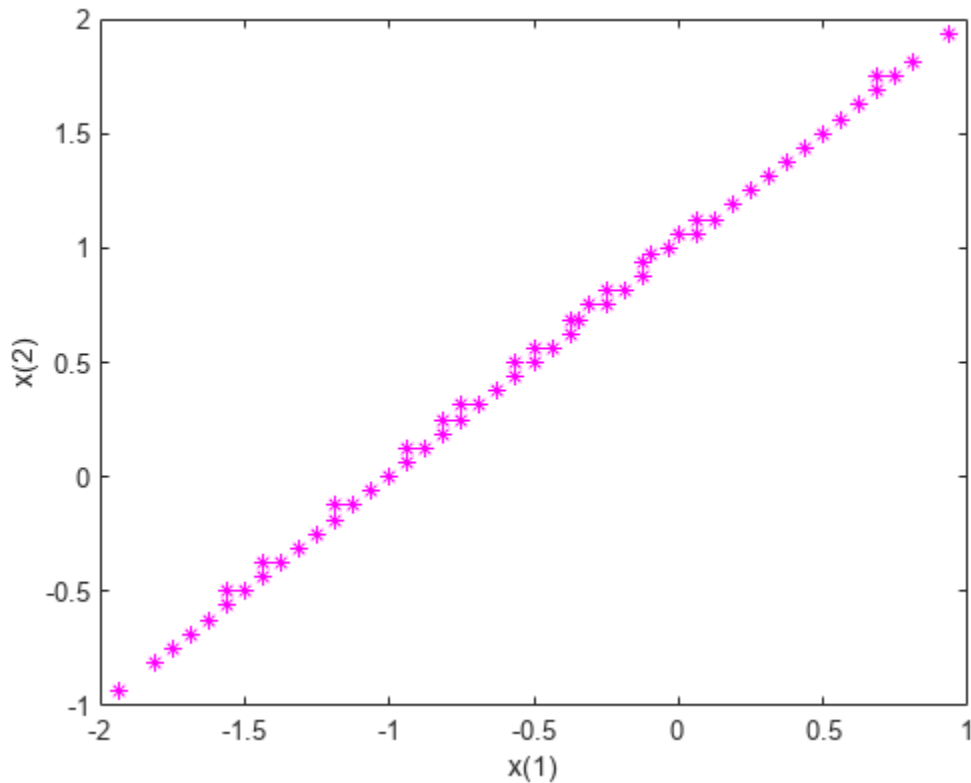
```
fun = @(x)[norm(x-[1,2])^2;norm(x+[2,1])^2];  
rng default % For reproducibility  
x = paretosearch(fun,2);
```

Pareto set found that satisfies the constraints.

Optimization completed because the relative change in the volume of the Pareto set is less than 'options.ParetoSetChangeTolerance' and constraints are satisfied to within 'options.ConstraintTolerance'.

Plot the solution as a scatter plot.

```
plot(x(:,1),x(:,2),'m*')  
xlabel('x(1)')  
ylabel('x(2)')
```



Theoretically, the solution of this problem is a straight line from $[-2, -1]$ to $[1, 2]$. `paretosearch` returns evenly-spaced points close to this line.

Create Pareto Front with Linear Constraints

Create a Pareto front for a two-objective problem in two dimensions subject to the linear constraint $x(1) + x(2) \leq 1$.

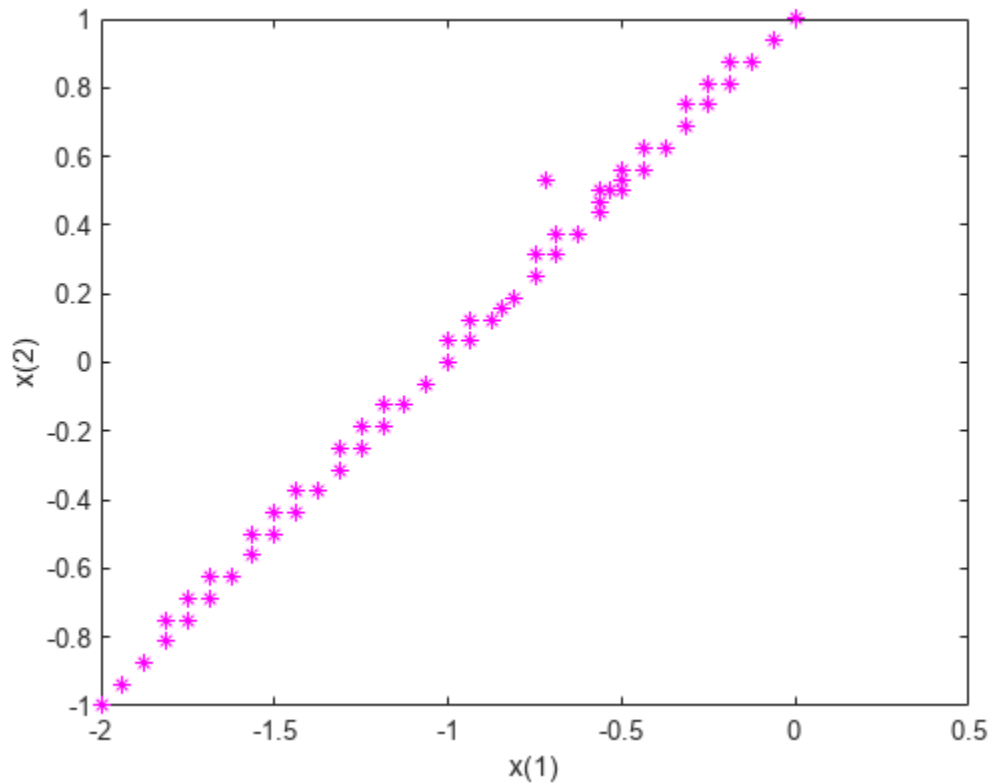
```
fun = @(x)[norm(x-[1,2])^2;norm(x+[2,1])^2];
A = [1,1];
b = 1;
rng default % For reproducibility
x = paretosearch(fun,2,A,b);
```

Pareto set found that satisfies the constraints.

Optimization completed because the relative change in the volume of the Pareto set is less than 'options.ParetoSetChangeTolerance' and constraints are satisfied to within 'options.ConstraintTolerance'.

Plot the solution as a scatter plot.

```
plot(x(:,1),x(:,2),'m*')
xlabel('x(1)')
ylabel('x(2)')
```



Theoretically, the solution of this problem is a straight line from $[-2, -1]$ to $[0, 1]$. `paretosearch` returns evenly-spaced points close to this line.

Create Pareto Front with Bounds

Create a Pareto front for a two-objective problem in two dimensions subject to the bounds $x(1) \geq 0$ and $x(2) \leq 1$.

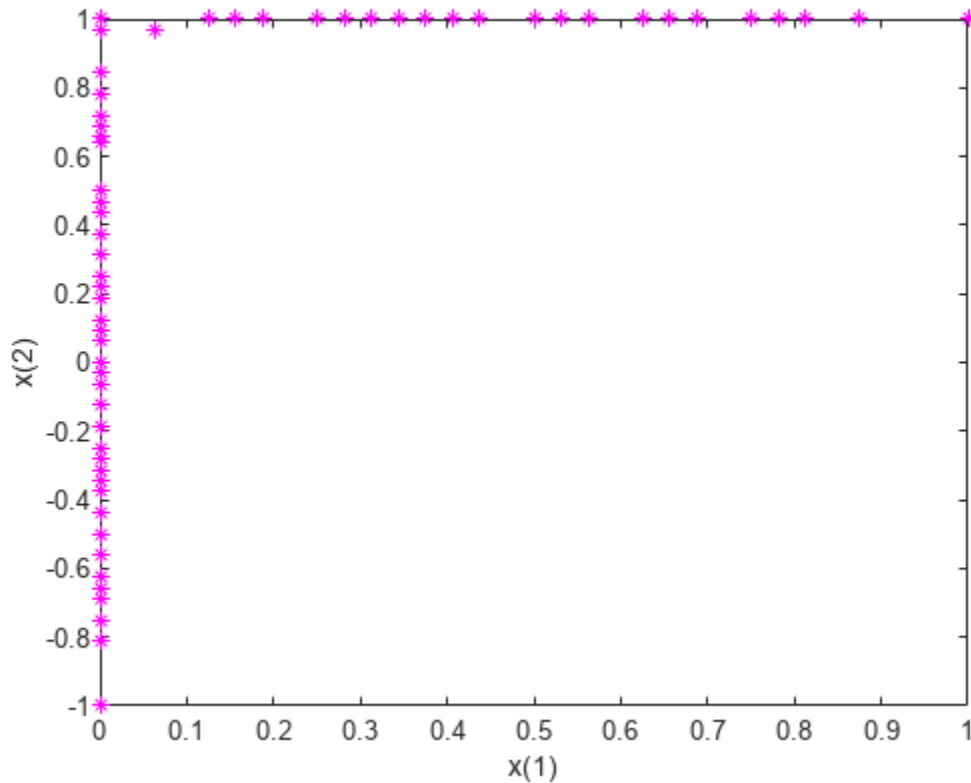
```
fun = @(x)[norm(x-[1,2])^2;norm(x+[2,1])^2];
lb = [0,-Inf]; % x(1) >= 0
ub = [Inf,1]; % x(2) <= 1
rng default % For reproducibility
x = paretosearch(fun,2,[],[],[],[],lb,ub);
```

Pareto set found that satisfies the constraints.

Optimization completed because the relative change in the volume of the Pareto set is less than 'options.ParetoSetChangeTolerance' and constraints are satisfied to within 'options.ConstraintTolerance'.

Plot the solution as a scatter plot.

```
plot(x(:,1),x(:,2),'m*')
xlabel('x(1)')
ylabel('x(2)')
```



All of the solution points are on the constraint boundaries $x(1) = 0$ or $x(2) = 1$.

Create Pareto Front with Nonlinear Constraints

Create a Pareto front for a two-objective problem in two dimensions subject to bounds $-1.1 \leq x(i) \leq 1.1$ and the nonlinear constraint $\text{norm}(x)^2 \leq 1.2$. The nonlinear constraint function appears at the end of this example, and works if you run this example as a live script. To run this example otherwise, include the nonlinear constraint function as a file on your MATLAB® path.

To better see the effect of the nonlinear constraint, set options to use a large Pareto set size.

```
rng default % For reproducibility
fun = @(x)[norm(x-[1,2])^2;norm(x+[2,1])^2];
lb = [-1.1,-1.1];
ub = [1.1,1.1];
options = optimoptions('paretosearch','ParetoSetSize',200);
x = paretosearch(fun,2,[],[],[],[],lb,ub,@circlecons,options);
```

Pareto set found that satisfies the constraints.

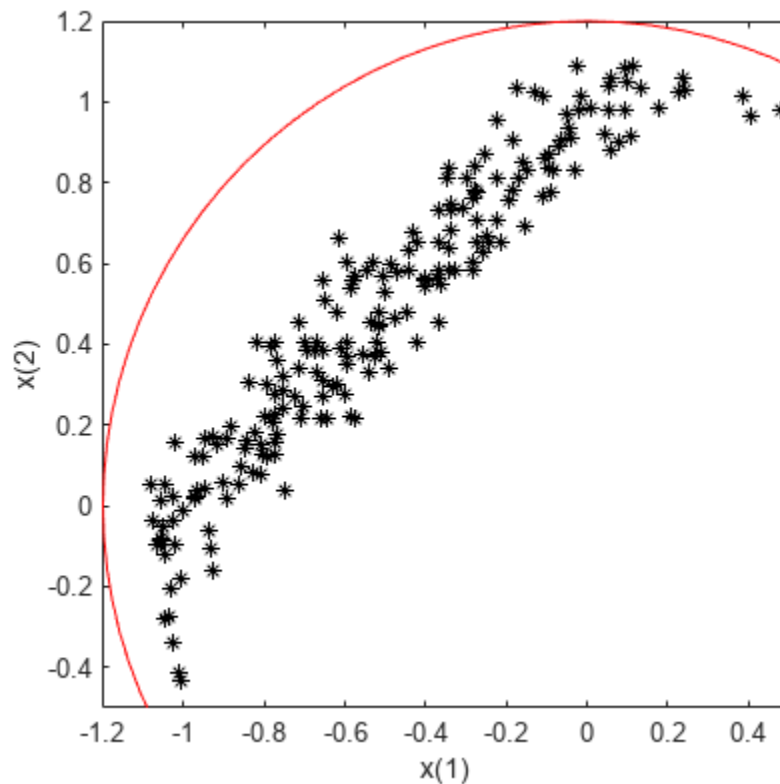
Optimization completed because the relative change in the volume of the Pareto set is less than 'options.ParetoSetChangeTolerance' and constraints are satisfied to within 'options.ConstraintTolerance'.

Plot the solution as a scatter plot. Include a plot of the circular constraint boundary.


```

figure
plot(x(:,1),x(:,2),'k*')
xlabel('x(1)')
ylabel('x(2)')
hold on
rectangle('Position',[-1.2 -1.2 2.4 2.4], 'Curvature',1, 'EdgeColor','r')
xlim([-1.2,0.5])
ylim([-0.5,1.2])
axis square
hold off

```



The solution points that have positive $x(1)$ values or negative $x(2)$ values are close to the nonlinear constraint boundary.

```

function [c,ceq] = circlecons(x)
ceq = [];
c = norm(x)^2 - 1.2;
end

```

Find Pareto Front Using Options

To monitor the progress of paretosearch, specify the 'psplotparetof' plot function.

```

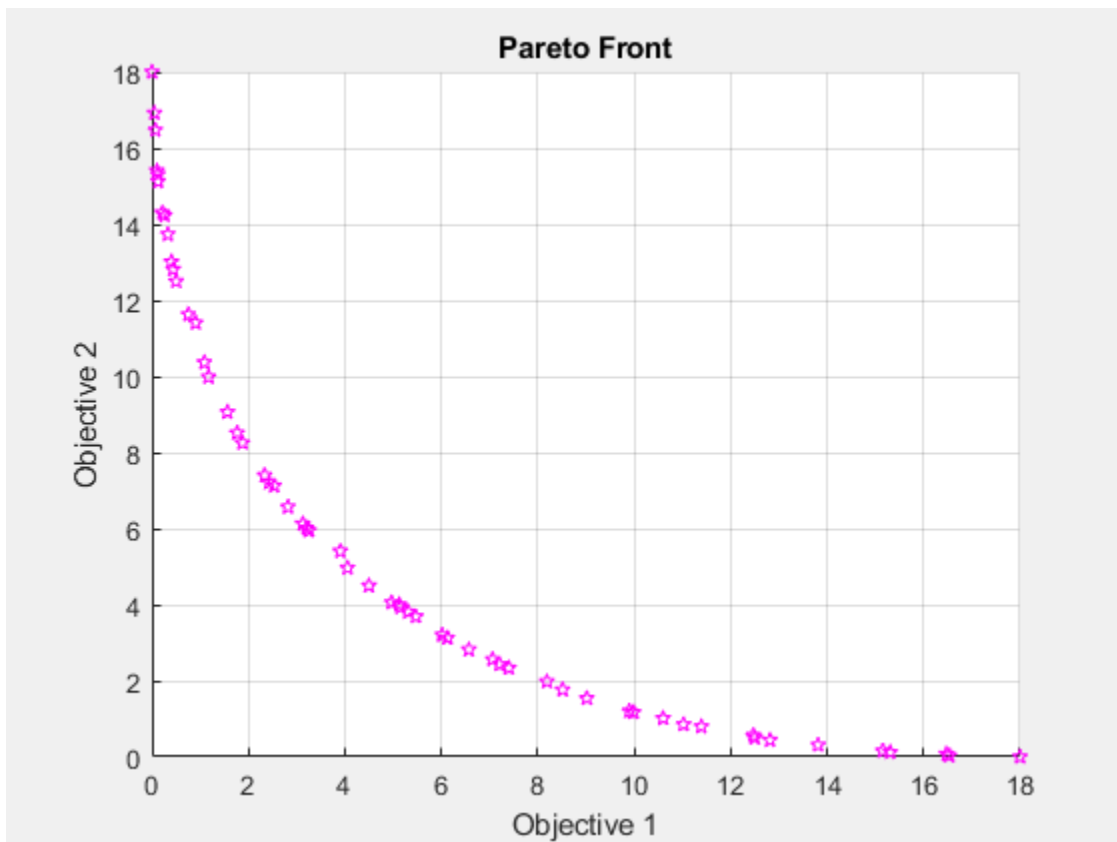
fun = @(x)[norm(x-[1,2])^2;norm(x+[2,1])^2];
options = optimoptions('paretosearch','PlotFcn','psplotparetof');

```

```
lb = [-4,-4];
ub = -lb;
x = paretosearch(fun,2,[],[],[],[],lb,ub,[],options);
```

Pareto set found that satisfies the constraints.

Optimization completed because the relative change in the volume of the Pareto set is less than 'options.ParetoSetChangeTolerance' and constraints are satisfied to within 'options.ConstraintTolerance'.



The solution looks like a quarter-circular arc with radius 18, which can be shown to be the analytical solution.

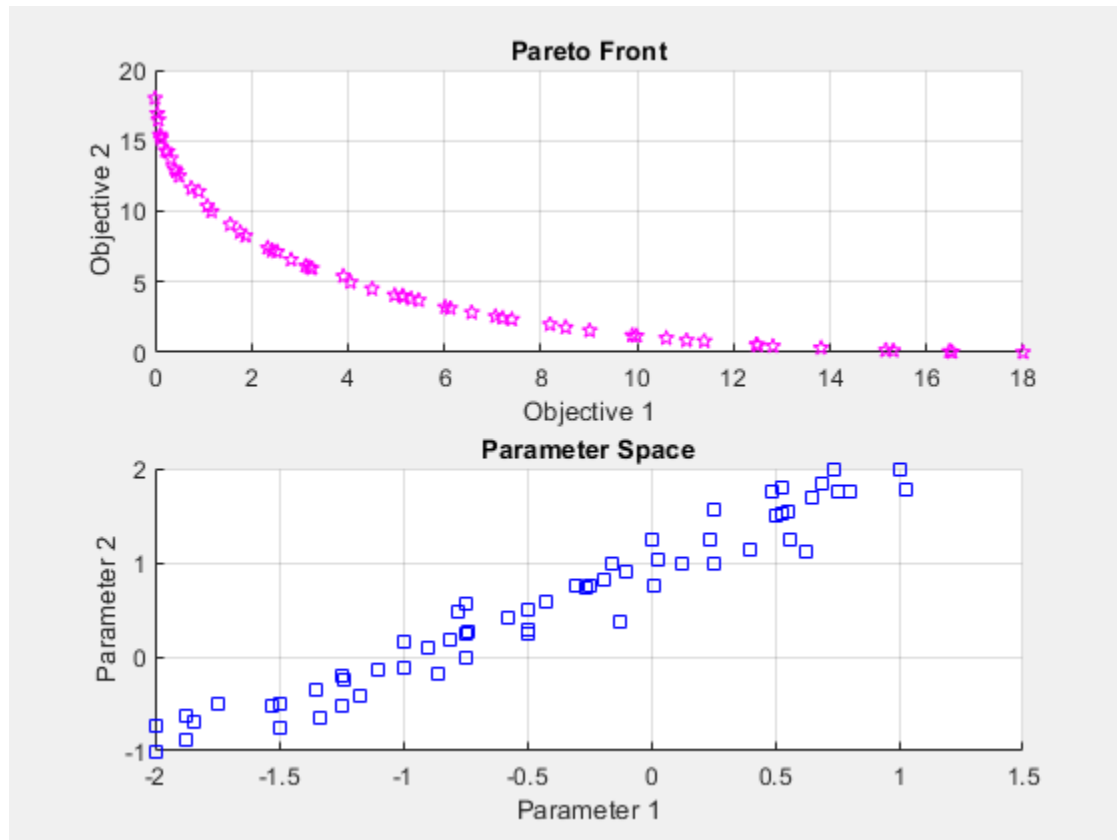
Find Pareto Front in Function Space and Parameter Space

Obtain the Pareto front in both function space and parameter space by calling `paretosearch` with both the `x` and `fval` outputs. Set options to plot the Pareto set in both function space and parameter space.

```
fun = @(x)[norm(x-[1,2])^2;norm(x+[2,1])^2];
lb = [-4,-4];
ub = -lb;
options = optimoptions('paretosearch','PlotFcn',{'psplotparetof' 'psplotparetox'});
rng default % For reproducibility
[x,fval] = paretosearch(fun,2,[],[],[],[],lb,ub,[],options);
```

Pareto set found that satisfies the constraints.

Optimization completed because the relative change in the volume of the Pareto set is less than 'options.ParetoSetChangeTolerance' and constraints are satisfied to within 'options.ConstraintTolerance'.



The analytical solution in objective function space is a quarter-circular arc of radius 18. In parameter space, the analytical solution is a straight line from $[-2, -1]$ to $[1, 2]$. The solution points are close to the analytical curves.

Monitor Pareto Set Solution

Set options to monitor the Pareto set solution process. Also, obtain more outputs from paretosearch to enable you to understand the solution process.

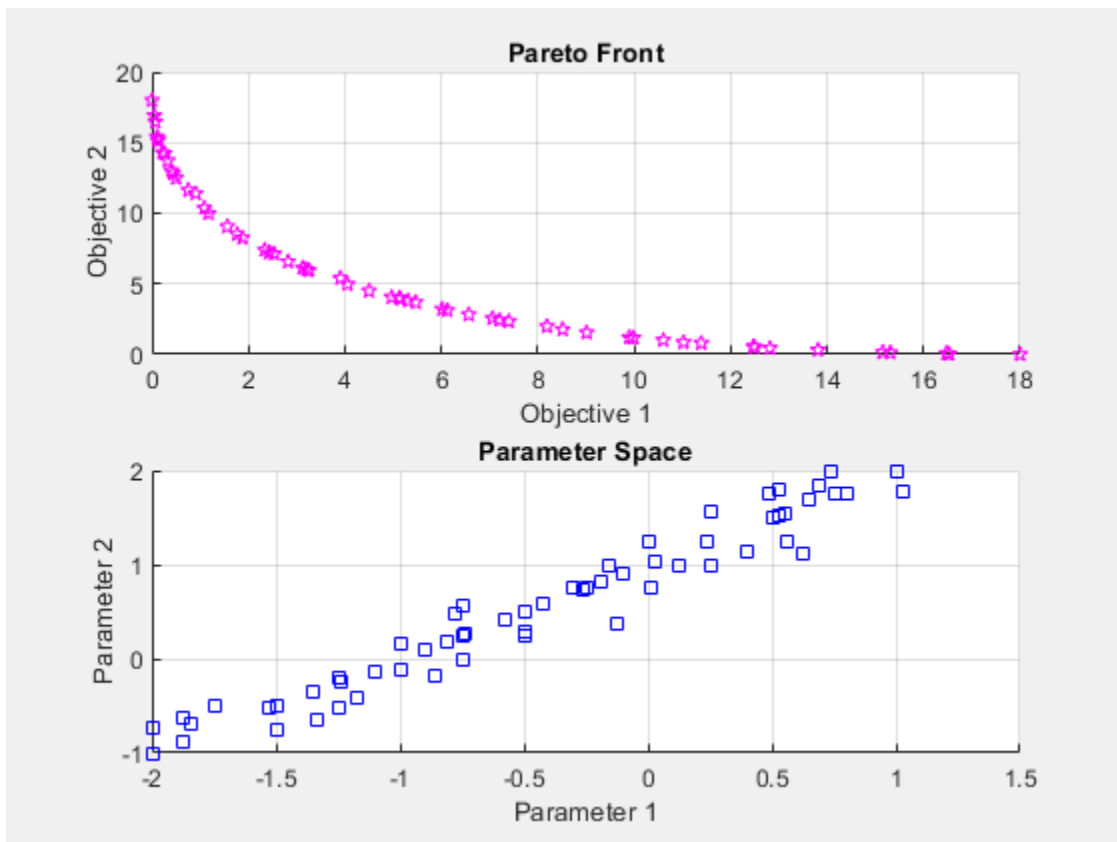
```
options = optimoptions('paretosearch','Display','iter',...
    'PlotFcn',{'psplotparetof' 'psplotparetox'});
fun = @(x)[norm(x-[1,2])^2;norm(x+[2,1])^2];
lb = [-4,-4];
ub = -lb;
rng default % For reproducibility
[x,fval,exitflag,output] = paretosearch(fun,2,[],[],[],[],lb,ub,[],options);
```

Iter	F-count	NumSolutions	Spread	Volume
0	60	11	-	3.7872e+02

1	386	12	7.6126e-01	3.4654e+02
2	702	27	9.5232e-01	2.9452e+02
3	1029	27	6.6332e-02	2.9904e+02
4	1357	36	1.3874e-01	3.0070e+02
5	1690	37	1.5379e-01	3.0200e+02
6	2014	50	1.7828e-01	3.0252e+02
7	2214	59	1.8536e-01	3.0320e+02
8	2344	60	1.9435e-01	3.0361e+02
9	2464	60	2.1055e-01	3.0388e+02

Pareto set found that satisfies the constraints.

Optimization completed because the relative change in the volume of the Pareto set is less than 'options.ParetoSetChangeTolerance' and constraints are satisfied to within 'options.ConstraintTolerance'.



Examine the additional outputs.

```
fprintf('Exit flag %d.\n',exitflag)
```

```
Exit flag 1.
```

```
disp(output)
```

```

iterations: 10
funccount: 2464
volume: 303.6076
averagedistance: 0.0250

```

```

        spread: 0.2105
    maxconstraint: 0
    message: 'Pareto set found that satisfies the constraints. ...'
    rngstate: [1x1 struct]

```

Obtain Pareto Front Residuals

Obtain and examine the Pareto front constraint residuals. Create a problem with the linear inequality constraint $\text{sum}(x) \leq -1/2$ and the nonlinear inequality constraint $\text{norm}(x)^2 \leq 1.2$. For improved accuracy, use 200 points on the Pareto front, and a `ParetoSetChangeTolerance` of $1e-7$, and give the natural bounds $-1.2 \leq x(i) \leq 1.2$.

The nonlinear constraint function appears at the end of this example, and works if you run this example as a live script. To run this example otherwise, include the nonlinear constraint function as a file on your MATLAB® path.

```

fun = @(x)[norm(x-[1,2])^2;norm(x+[2,1])^2];
A = [1,1];
b = -1/2;
lb = [-1.2,-1.2];
ub = -lb;
nonlcon = @circlecons;
rng default % For reproducibility
options = optimoptions('paretosearch','ParetoSetChangeTolerance',1e-7,...
    'PlotFcn',{'psplotparetof' 'psplotparetox'},'ParetoSetSize',200);

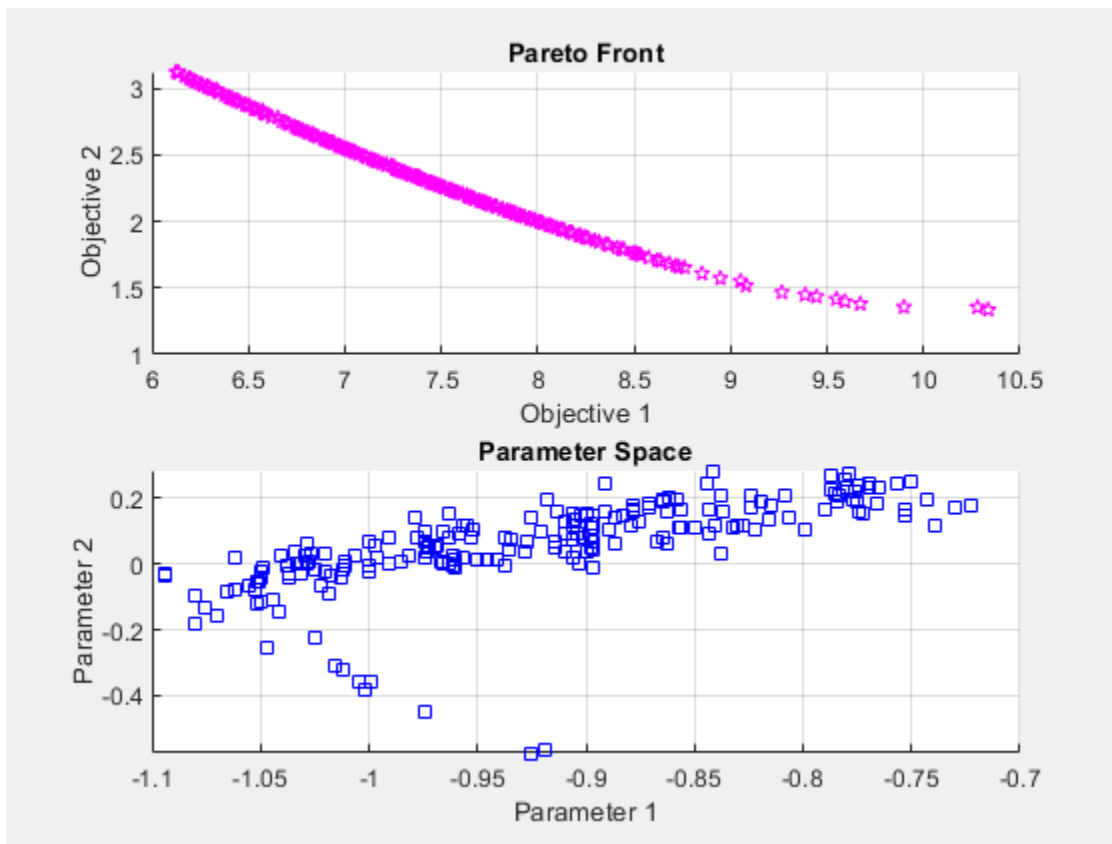
```

Call `paretosearch` using all outputs.

```
[x,fval,exitflag,output,residuals] = paretosearch(fun,2,A,b,[],[],lb,ub,nonlcon,options);
```

```
Pareto set found that satisfies the constraints.
```

```
Optimization completed because the relative change in the volume of the Pareto set
is less than 'options.ParetoSetChangeTolerance' and constraints are satisfied to within
'options.ConstraintTolerance'.
```



The inequality constraints reduce the size of the Pareto set compared to an unconstrained set. Examine the returned residuals.

```
fprintf('The maximum linear inequality constraint residual is %f.\n',max(residuals.ineqlin))
```

```
The maximum linear inequality constraint residual is 0.000000.
```

```
fprintf('The maximum nonlinear inequality constraint residual is %f.\n',max(residuals.ineqnonlin))
```

```
The maximum nonlinear inequality constraint residual is -0.000695.
```

The maximum returned residuals are negative, meaning that all the returned points are feasible. The maximum returned residuals are close to zero, meaning that each constraint is active for some points.

```
function [c,ceq] = circlecons(x)
ceq = [];
c = norm(x)^2 - 1.2;
end
```

Input Arguments

fun — Objective functions to optimize

function handle | function name

Objective functions to optimize, specified as a function handle or function name.

`fun` is a function that accepts a real row vector of doubles `x` of length `nvars` and returns a real vector `F(x)` of objective function values. For details on writing `fun`, see “Compute Objective Functions” on page 2-2.

If you set the `UseVectorized` option to `true`, then `fun` accepts a matrix of size `n-by-nvars`, where the matrix represents `n` individuals. `fun` returns a matrix of size `n-by-m`, where `m` is the number of objective functions. See “Vectorize the Fitness Function” on page 8-103.

Example: `@(x)[sin(x),cos(x)]`

Data Types: `char` | `function_handle` | `string`

nvars — Number of variables

positive integer

Number of variables, specified as a positive integer. The solver passes row vectors of length `nvars` to `fun`.

Example: 4

Data Types: `double`

A — Linear inequality constraints

real matrix

Linear inequality constraints, specified as a real matrix. `A` is an `M-by-nvars` matrix, where `M` is the number of inequalities.

`A` encodes the `M` linear inequalities

$$A*x \leq b,$$

where `x` is the column vector of `nvars` variables `x(:)`, and `b` is a column vector with `M` elements.

For example, to specify

$$\begin{aligned} x_1 + 2x_2 &\leq 10 \\ 3x_1 + 4x_2 &\leq 20 \\ 5x_1 + 6x_2 &\leq 30, \end{aligned}$$

give these constraints:

$$\begin{aligned} A &= [1,2;3,4;5,6]; \\ b &= [10;20;30]; \end{aligned}$$

Example: To specify that the control variables sum to 1 or less, give the constraints `A = ones(1,N)` and `b = 1`.

Data Types: `double`

b — Linear inequality constraints

real vector

Linear inequality constraints, specified as a real vector. `b` is an `M`-element vector related to the `A` matrix. If you pass `b` as a row vector, solvers internally convert `b` to the column vector `b(:)`.

`b` encodes the `M` linear inequalities

$$A*x \leq b,$$

where x is the column vector of N variables $x(:)$, and A is a matrix of size M -by- N .

For example, to specify

$$\begin{aligned}x_1 + 2x_2 &\leq 10 \\3x_1 + 4x_2 &\leq 20 \\5x_1 + 6x_2 &\leq 30,\end{aligned}$$

give these constraints:

$$\begin{aligned}A &= [1,2;3,4;5,6]; \\b &= [10;20;30];\end{aligned}$$

Example: To specify that the control variables sum to 1 or less, give the constraints $A = \text{ones}(1,N)$ and $b = 1$.

Data Types: double

Aeq — Linear equality constraints

real matrix

Linear equality constraints, specified as a real matrix. Aeq is an Me -by- $nvars$ matrix, where Me is the number of equalities.

Aeq encodes the Me linear equalities

$$Aeq*x = beq,$$

where x is the column vector of N variables $x(:)$, and beq is a column vector with Me elements.

For example, to specify

$$\begin{aligned}x_1 + 2x_2 + 3x_3 &= 10 \\2x_1 + 4x_2 + x_3 &= 20,\end{aligned}$$

give these constraints:

$$\begin{aligned}Aeq &= [1,2,3;2,4,1]; \\beq &= [10;20];\end{aligned}$$

Example: To specify that the control variables sum to 1, give the constraints $Aeq = \text{ones}(1,N)$ and $beq = 1$.

Data Types: double

beq — Linear equality constraints

real vector

Linear equality constraints, specified as a real vector. beq is an Me -element vector related to the Aeq matrix. If you pass beq as a row vector, solvers internally convert beq to the column vector $beq(:)$.

beq encodes the Me linear equalities

$$Aeq*x = beq,$$

where x is the column vector of N variables $x(:)$, and Aeq is a matrix of size Meq -by- N .

For example, to specify

$$\begin{aligned}x_1 + 2x_2 + 3x_3 &= 10 \\ 2x_1 + 4x_2 + x_3 &= 20,\end{aligned}$$

give these constraints:

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```

Example: To specify that the control variables sum to 1, give the constraints `Aeq = ones(1,N)` and `beq = 1`.

Data Types: `double`

lb – Lower bounds

`[]` (default) | real vector or array

Lower bounds, specified as a real vector or array of doubles. `lb` represents the lower bounds element-wise in $lb \leq x \leq ub$.

Internally, `paretosearch` converts an array `lb` to the vector `lb(:)`.

Example: `lb = [0;-Inf;4]` means $x(1) \geq 0$, $x(3) \geq 4$.

Data Types: `double`

ub – Upper bounds

`[]` (default) | real vector or array

Upper bounds, specified as a real vector or array of doubles. `ub` represents the upper bounds element-wise in $lb \leq x \leq ub$.

Internally, `paretosearch` converts an array `ub` to the vector `ub(:)`.

Example: `ub = [Inf;4;10]` means $x(2) \leq 4$, $x(3) \leq 10$.

Data Types: `double`

nonlcon – Nonlinear constraints

function handle | function name

Nonlinear constraints, specified as a function handle or function name. `nonlcon` is a function that accepts a row vector `x` and returns two row vectors, `c(x)` and `ceq(x)`.

- `c(x)` is the row vector of nonlinear inequality constraints at `x`. The `paretosearch` function attempts to satisfy $c(x) \leq 0$ for all entries of `c`.
- `ceq(x)` must return `[]`, because currently `paretosearch` does not support nonlinear equality constraints.

If you set the `UseVectorized` option to `true`, then `nonlcon` accepts a matrix of size `n-by-nvars`, where the matrix represents `n` individuals. `nonlcon` returns a matrix of size `n-by-mc` in the first argument, where `mc` is the number of nonlinear inequality constraints. See “Vectorize the Fitness Function” on page 8-103.

For example, `x = paretosearch(@myfun,nvars,A,b,Aeq,beq,lb,ub,@mycon)`, where `mycon` is a MATLAB function such as the following:

```
function [c,ceq] = mycon(x)
c = ...      % Compute nonlinear inequalities at x.
ceq = []     % No nonlinear equalities at x.
```

For more information, see “Nonlinear Constraints”.

Data Types: char | function_handle | string

options — Optimization options

output of `optimoptions` | structure

Optimization options, specified as the output of `optimoptions` or as a structure.

{ } denotes the default value. See option details in “Pattern Search Options” on page 17-7.

Options for paretosearch

Option	Description	Values
ConstraintTolerance	Tolerance on constraints. For an options structure, use TolCon.	Positive scalar $\{1e-6\}$
Display	Level of display.	'off' 'iter' 'diagnose' {'final'}
InitialPoints	Initial points for paretosearch. Use one of these data types: <ul style="list-style-type: none"> Matrix with <code>nvars</code> columns, where each row represents one initial point. Structure containing the following fields (all fields are optional except <code>X0</code>): <ul style="list-style-type: none"> <code>X0</code> — Matrix with <code>nvars</code> columns, where each row represents one initial point. <code>Fvals</code> — Matrix with <code>numObjectives</code> columns, where each row represents the objective function values at the corresponding point in <code>X0</code>. <code>Cineq</code> — Matrix with <code>numIneq</code> columns, where each row represents the nonlinear inequality constraint values at the corresponding point in <code>X0</code>. <p>paretosearch computes any missing values in the <code>Fvals</code> and <code>Cineq</code> fields.</p>	Matrix with <code>nvars</code> columns structure $\{\{\}\}$
MaxFunctionEvaluations	Maximum number of objective function evaluations. For an options structure, use MaxFunEvals.	Positive integer $\{ '2000*\text{numberOfVariables}' \}$ for patternsearch, $\{ '3000*(\text{numberOfVariables} + \text{numberOfObjectives})' \}$ for paretosearch, where <code>numberOfVariables</code> is the number of problem variables, and <code>numberOfObjectives</code> is the number of objective functions

Option	Description	Values
MaxIterations	<p>Maximum number of iterations.</p> <p>For an options structure, use MaxIter.</p>	<p>Positive integer {'100*numberOfVariables'} for patternsearch, {'100*(numberOfVariables+numberOfObjectives)'} for paretosearch, where numberOfVariables is the number of problem variables, and numberOfObjectives is the number of objective functions</p>
MaxTime	<p>Total time (in seconds) allowed for optimization.</p> <p>For an options structure, use TimeLimit.</p>	<p>Positive scalar {Inf}</p>
MeshTolerance	<p>Tolerance on the mesh size.</p> <p>For an options structure, use TolMesh.</p>	<p>Positive scalar {1e-6}</p>
MinPollFraction	<p>Minimum fraction of the pattern to poll.</p>	<p>Scalar from 0 through 1 {0}</p>
OutputFcn	<p>Function that an optimization function calls at each iteration. Specify as a function handle or a cell array of function handles.</p> <p>For an options structure, use OutputFcns.</p>	<p>Function handle or cell array of function handles on page 17-17 {}</p>

Option	Description	Values
ParetoSetChangeTolerance	<p>The solver stops when the relative change in a stopping measure over a window of iterations is less than or equal to ParetoSetChangeTolerance.</p> <ul style="list-style-type: none"> For three or fewer objectives, paretosearch uses the volume and spread measures. For four or more objectives, paretosearch uses the spread and distance measures. <p>See “Definitions for paretosearch Algorithm” on page 14-10.</p> <p>The solver stops when the relative change in any applicable measure is less than ParetoSetChangeTolerance, or the maximum of the squared Fourier transforms of the time series of these measures is relatively small. See “paretosearch Algorithm” on page 14-10.</p> <hr/> <p>Note Setting ParetoSetChangeTolerance < $\sqrt{\text{eps}}$ $\sim 1.5\text{e-}8$ is not recommended.</p>	Positive scalar {1e-4}
ParetoSetSize	Number of points in the Pareto set.	Positive integer {'max(numberOfObjectives, 60)'} , where numberOfObjectives is the number of objective functions
PlotFcn	<p>Plots of output from the pattern search. Specify as the name of a built-in plot function, a function handle, or a cell array of names of built-in plot functions or function handles.</p> <p>For an options structure, use PlotFcns.</p>	<p>{[]} 'psplotfunccount' 'psplotmaxconstr' custom plot function on page 17-8</p> <p>With multiple objectives: 'psplotdistance' 'psplotparetof' 'psplotparetox' 'psplotspread' 'psplotvolume'</p> <p>With a single objective: 'psplotbestf' 'psplotmeshsize' 'psplotbestx'</p>

Option	Description	Values
PollMethod	<p>Polling strategy used in the pattern search.</p> <hr/> <p>Note You cannot use MADS polling when the problem has linear equality constraints.</p>	{'GPSPositiveBasis2np2'} 'GPSPositiveBasis2N' 'GPSPositiveBasisNp1' 'GSSPositiveBasis2N' 'GSSPositiveBasisNp1' 'MADSPositiveBasis2N' 'MADSPositiveBasisNp1' 'GSSPositiveBasis2np2'
UseParallel	<p>Compute objective and nonlinear constraint functions in parallel. See “Vectorized and Parallel Options” on page 17-19 and “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11.</p> <hr/> <p>Note You must set UseCompletePoll to true for patternsearch to use vectorized or parallel polling. Similarly, set UseCompleteSearch to true for vectorized or parallel searching.</p> <p>Beginning in R2019a, when you set the UseParallel option to true, patternsearch internally overrides the UseCompletePoll setting to true so that the function polls in parallel.</p>	true {false}
UseVectorized	<p>Specifies whether functions are vectorized. See “Vectorized and Parallel Options” on page 17-19 and “Vectorize the Objective and Constraint Functions” on page 6-83.</p> <hr/> <p>Note You must set UseCompletePoll to true for patternsearch to use vectorized or parallel polling. Similarly, set UseCompleteSearch to true for vectorized or parallel searching.</p> <hr/> <p>For an options structure, use Vectorized = 'on' or 'off'.</p>	true {false}

```
Example: options =
optimoptions('paretosearch','Display','none','UseParallel',true)
```

problem — Problem structure
structure

Problem structure, specified as a structure with the following fields:

- `objective` — Objective function
- `nvars` — Number of variables
- `Aineq` — Matrix for linear inequality constraints
- `bineq` — Vector for linear inequality constraints
- `Aeq` — Matrix for linear equality constraints
- `beq` — Vector for linear equality constraints
- `lb` — Lower bound for x
- `ub` — Upper bound for x
- `nonlcon` — Nonlinear constraint function
- `solver` — 'paretosearch'
- `options` — Options created with `optimoptions`
- `rngstate` — Optional field to reset the state of the random number generator

Note All fields in `problem` are required, except for `rngstate`, which is optional.

Data Types: `struct`

Output Arguments

`x` — Pareto points

`m`-by-`nvars` array

Pareto points, returned as an `m`-by-`nvars` array, where `m` is the number of points on the Pareto front. Each row of `x` represents one point on the Pareto front.

`fval` — Function values on Pareto front

`m`-by-`nf` array

Function values on the Pareto front, returned as an `m`-by-`nf` array. `m` is the number of points on the Pareto front, and `nf` is the number of objective functions. Each row of `fval` represents the function values at one Pareto point in `x`.

`exitflag` — Reason paretosearch stopped

integer

Reason paretosearch stopped, returned as one of the integer values in this table.

Exit Flag	Stopping Condition
1	<p data-bbox="443 331 623 457">One of the following conditions is met.</p> <ul data-bbox="443 485 623 1902" style="list-style-type: none"><li data-bbox="443 485 623 926">• Mesh size of all incumbents is less than <code>options.MeshTolerance</code> and constraints (if any) are satisfied to within <code>options.ConstraintTolerance</code>.<li data-bbox="443 974 623 1499">• Relative change in the spread of the Pareto set is less than <code>options.ParetoSetChangeTolerance</code> and constraints (if any) are satisfied to within <code>options.ConstraintTolerance</code>.<li data-bbox="443 1556 623 1902">• Relative change in the volume of the Pareto set is less than <code>options.ParetoSetChangeTolerance</code> and constraints

Exit Flag	Stopping Condition
	(if any) are satisfied to within <code>options.ConstraintTolerance</code> .
0	Number of iterations exceeds <code>options.MaxIterations</code> , or the number of function evaluations exceeds <code>options.MaxFunctionEvaluations</code> .
-1	Optimization is stopped by an output function or plot function.
-2	Solver cannot find a point satisfying all the constraints.
-5	Optimization time exceeds <code>options.MaxTime</code> .

output — Information about the optimization process

structure

Information about the optimization process, returned as a structure with these fields:

- `iterations` — Total number of iterations.
- `funccount` — Total number of function evaluations.
- `volume` — Hyper-volume of the set formed from the Pareto points in function space. See “Definitions for paretosearch Algorithm” on page 14-10.
- `averagedistance` — Average distance measure of the Pareto points in function space. See “Definitions for paretosearch Algorithm” on page 14-10.
- `spread` — Average spread measure of the Pareto points. See “Definitions for paretosearch Algorithm” on page 14-10.

- `maxconstraint` — Maximum constraint violation, if any.
- `message` — Reason why the algorithm terminated.
- `rngstate` — State of the MATLAB random number generator just before the algorithm starts. You can use the values in `rngstate` to reproduce the output when you use a random poll method such as 'MADSPositiveBasis2N' or when you use the default quasirandom method of creating the initial population. See “Reproduce Results” on page 8-67, which discusses the identical technique for `ga`.

residuals — Constraint residuals at `x`

structure

Constraint residuals at `x`, returned as a structure with these fields (a glossary of the field size terms and entries follows the table).

Field Name	Field Size	Entries
<code>lower</code>	<code>m-by-nvars</code>	$\lfloor b - x$
<code>upper</code>	<code>m-by-nvars</code>	$x - ub$
<code>ineqlin</code>	<code>m-by-ncon</code>	$A*x - b$
<code>eqlin</code>	<code>m-by-ncon</code>	$ Aeq*x - b $
<code>ineqnonlin</code>	<code>m-by-ncon</code>	$c(x)$

- `m` — Number of returned points `x` on the Pareto front
- `nvars` — Number of control variables
- `ncon` — Number of constraints of the relevant type (such as number of rows of `A` or number of returned nonlinear equalities)
- `c(x)` — Numeric values of the nonlinear constraint functions

More About

Nondominated

Nondominated points, also called noninferior points, are points for which no other point has lower values of all objective functions. In other words, for nondominated points, none of the objective function values can be improved (lowered) without raising other objective function values. See “What Is Multiobjective Optimization?” on page 14-2.

Algorithms

`paretosearch` uses a pattern search to search for points on the Pareto front. For details, see “`paretosearch` Algorithm” on page 14-10.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for `paretosearch`.

Version History

Introduced in R2018b

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set the 'UseParallel' option to true.

```
options = optimoptions('solvername','UseParallel',true)
```

For more information, see “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11.

See Also

gamultiobj | patternsearch | **Optimize**

Topics

“Multiobjective Optimization”

particleswarm

Particle swarm optimization

Syntax

```
x = particleswarm(fun,nvars)
x = particleswarm(fun,nvars,lb,ub)
x = particleswarm(fun,nvars,lb,ub,options)
x = particleswarm(problem)
[x,fval,exitflag,output] = particleswarm( ___ )
```

Description

`x = particleswarm(fun,nvars)` attempts to find a vector `x` that achieves a local minimum of `fun`. `nvars` is the dimension (number of design variables) of `fun`.

Note “Passing Extra Parameters” explains how to pass extra parameters to the objective function, if necessary.

`x = particleswarm(fun,nvars,lb,ub)` defines a set of lower and upper bounds on the design variables, `x`, so that a solution is found in the range $lb \leq x \leq ub$.

`x = particleswarm(fun,nvars,lb,ub,options)` minimizes with the default optimization parameters replaced by values in `options`. Set `lb = []` and `ub = []` if no bounds exist.

`x = particleswarm(problem)` finds the minimum for `problem`, a structure described in `problem`.

`[x,fval,exitflag,output] = particleswarm(___)`, for any input arguments described above, returns:

- A scalar `fval`, which is the objective function value `fun(x)`
- A value `exitflag` describing the exit condition
- A structure `output` containing information about the optimization process

Examples

Minimize a Simple Function

Minimize a simple function of two variables.

Define the objective function.

```
fun = @(x)x(1)*exp(-norm(x)^2);
```

Call `particleswarm` to minimize the function.

```
rng default % For reproducibility
nvars = 2;
x = particleswarm(fun,nvars)
```

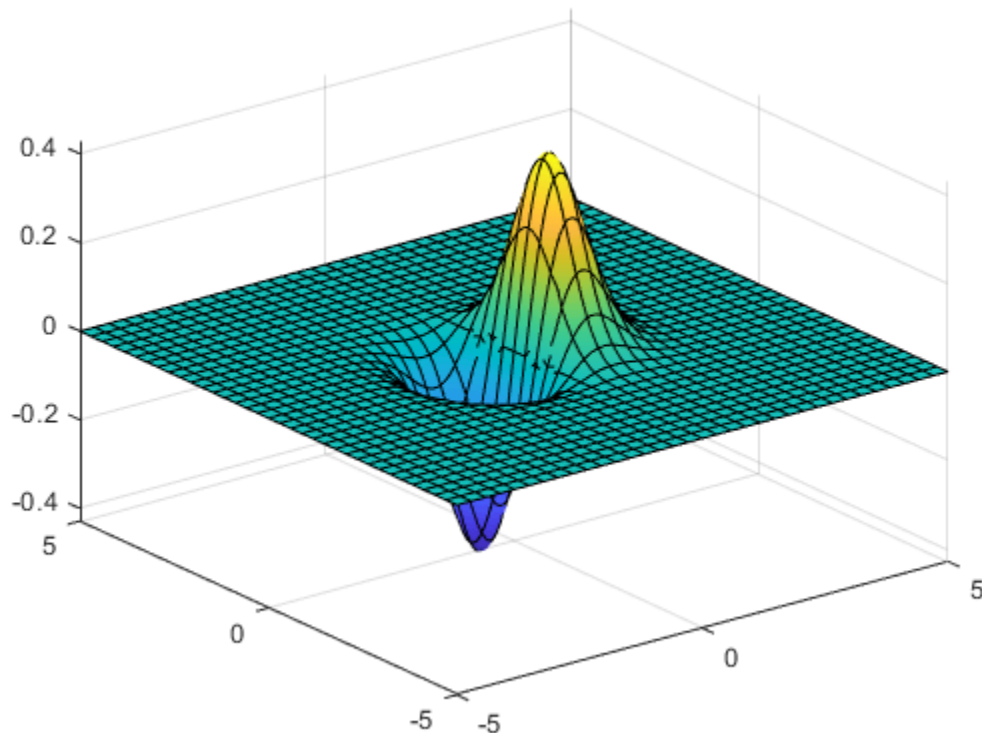
Optimization ended: relative change in the objective value over the last OPTIONS.MaxStallIterations iterations is less than OPTIONS.FunctionTolerance.

x =

```
629.4474 311.4814
```

This solution is far from the true minimum, as you see in a function plot.

```
fsurf(@(x,y)x.*exp(-(x.^2+y.^2)))
```



Usually, it is best to set bounds. See “Minimize a Simple Function with Bounds” on page 18-125.

Minimize a Simple Function with Bounds

Minimize a simple function of two variables with bound constraints.

Define the objective function.

```
fun = @(x)x(1)*exp(-norm(x)^2);
```

Set bounds on the variables.

```
lb = [-10, -15];  
ub = [15, 20];
```

Call `particleswarm` to minimize the function.

```
rng default % For reproducibility  
nvars = 2;  
x = particleswarm(fun, nvars, lb, ub)
```

Optimization ended: relative change in the objective value over the last `OPTIONS.MaxStallIterations` iterations is less than `OPTIONS.FunctionTolerance`.

```
x = 1×2  
    -0.7071    -0.0000
```

Minimize Using Nondefault Options

Use a larger population and a hybrid function to try to get a better solution.

Specify the objective function and bounds.

```
fun = @(x)x(1)*exp(-norm(x)^2);  
lb = [-10, -15];  
ub = [15, 20];
```

Specify the options.

```
options = optimoptions('particleswarm', 'SwarmSize', 100, 'HybridFcn', @fmincon);
```

Call `particleswarm` to minimize the function.

```
rng default % For reproducibility  
nvars = 2;  
x = particleswarm(fun, nvars, lb, ub, options)
```

Optimization ended: relative change in the objective value over the last `OPTIONS.MaxStallIterations` iterations is less than `OPTIONS.FunctionTolerance`.

```
x = 1×2  
    -0.7071    -0.0000
```

Examine the Solution Process

Return the optional output arguments to examine the solution process in more detail.

Define the problem.

```
fun = @(x)x(1)*exp(-norm(x)^2);  
lb = [-10, -15];
```

```
ub = [15,20];
options = optimoptions('particleswarm','SwarmSize',50,'HybridFcn',@fmincon);
```

Call `particleswarm` with all outputs to minimize the function and get information about the solution process.

```
rng default % For reproducibility
nvars = 2;
[x,fval,exitflag,output] = particleswarm(fun,nvars,lb,ub,options)
```

Optimization ended: relative change in the objective value over the last `OPTIONS.MaxStallIterations` iterations is less than `OPTIONS.FunctionTolerance`.

```
x = 1x2
```

```
    -0.7071    -0.0000
```

```
fval = -0.4289
```

```
exitflag = 1
```

```
output = struct with fields:
```

```
    rngstate: [1x1 struct]
```

```
    iterations: 43
```

```
    funccount: 2203
```

```
    message: 'Optimization ended: relative change in the objective value ...'
```

```
    hybridflag: 1
```

Input Arguments

fun — Objective function

function handle | function name

Objective function, specified as a function handle or function name. Write the objective function to accept a row vector of length `nvars` and return a scalar value.

When the `'UseVectorized'` option is `true`, write `fun` to accept a `pop-by-nvars` matrix, where `pop` is the current population size. In this case, `fun` returns a vector the same length as `pop` containing the fitness function values. Ensure that `fun` does not assume any particular size for `pop`, since `particleswarm` can pass a single member of a population even in a vectorized calculation.

Example: `fun = @(x)(x-[4,2]).^2`

Data Types: `char` | `function_handle` | `string`

nvars — Number of variables

positive integer

Number of variables, specified as a positive integer. The solver passes row vectors of length `nvars` to `fun`.

Example: 4

Data Types: `double`

lb – Lower bounds

[] (default) | real vector or array

Lower bounds, specified as a real vector or array of doubles. `lb` represents the lower bounds element-wise in $lb \leq x \leq ub$.

Internally, `particleswarm` converts an array `lb` to the vector `lb(:)`.

Example: `lb = [0; -Inf; 4]` means $x(1) \geq 0$, $x(3) \geq 4$.

Data Types: double

ub – Upper bounds

[] (default) | real vector or array

Upper bounds, specified as a real vector or array of doubles. `ub` represents the upper bounds element-wise in $lb \leq x \leq ub$.

Internally, `particleswarm` converts an array `ub` to the vector `ub(:)`.

Example: `ub = [Inf; 4; 10]` means $x(2) \leq 4$, $x(3) \leq 10$.

Data Types: double

options – Options for particleswarmoptions created using `optimoptions`

Options for `particleswarm`, specified as the output of the `optimoptions` function.

Some options are absent from the `optimoptions` display. These options are listed in italics. For details, see “View Optimization Options”.

<code>CreationFcn</code>	Function that creates the initial swarm. Specify as 'pswcreationuniform' or a function handle. Default is 'pswcreationuniform'. See “Swarm Creation” on page 17-45.
<code>Display</code>	Level of display returned to the command line. <ul style="list-style-type: none"> 'off' or 'none' displays no output. 'final' displays just the final output (default). 'iter' gives iterative display.
<i>DisplayInterval</i>	Interval for iterative display. The iterative display prints one line for every <code>DisplayInterval</code> iterations. Default is 1.
<code>FunctionTolerance</code>	Nonnegative scalar with default <code>1e-6</code> . Iterations end when the relative change in best objective function value over the last <code>MaxStallIterations</code> iterations is less than <code>options.FunctionTolerance</code> .
<i>FunValCheck</i>	Check whether objective function and constraints values are valid. 'on' displays an error when the objective function or constraints return a value that is complex, <code>Inf</code> , or <code>NaN</code> . The default, 'off', displays no error.

HybridFcn	<p>Function that continues the optimization after <code>particleswarm</code> terminates. Specify as a name or a function handle. Possible values:</p> <ul style="list-style-type: none"> 'fmincon' 'fminsearch' 'fminunc' 'patternsearch' <p>Can also be a cell array specifying the hybrid function and its options, such as <code>{@fmincon, fminconopts}</code>. Default is <code>[]</code>. See "Hybrid Function" on page 17-47.</p> <p>See "When to Use a Hybrid Function" on page 8-116.</p>
InertiaRange	<p>Two-element real vector with same sign values in increasing order. Gives the lower and upper bound of the adaptive inertia. To obtain a constant (nonadaptive) inertia, set both elements of <code>InertiaRange</code> to the same value. Default is <code>[0.1, 1.1]</code>. See "Particle Swarm Optimization Algorithm" on page 10-11.</p>
InitialSwarmMatrix	<p>Initial population or partial population of particles. <code>M</code>-by-<code>nvars</code> matrix, where each row represents one particle. If <code>M < SwarmSize</code>, then <code>particleswarm</code> creates more particles so that the total number is <code>SwarmSize</code>. If <code>M > SwarmSize</code>, then <code>particleswarm</code> uses the first <code>SwarmSize</code> rows.</p>
InitialSwarmSpan	<p>Initial range of particle positions that <code>@pswcreationuniform</code> creates. Can be a positive scalar or a vector with <code>nvars</code> elements, where <code>nvars</code> is the number of variables. The range for any particle component is <code>-InitialSwarmSpan/2, InitialSwarmSpan/2</code>, shifted and scaled if necessary to match any bounds. Default is <code>2000</code>.</p> <p><code>InitialSwarmSpan</code> also affects the range of initial particle velocities. See "Initialization" on page 10-11.</p>
MaxIterations	<p>Maximum number of iterations <code>particleswarm</code> takes. Default is <code>200*nvars</code>, where <code>nvars</code> is the number of variables.</p>
MaxStallIterations	<p>Positive integer with default <code>20</code>. Iterations end when the relative change in best objective function value over the last <code>MaxStallIterations</code> iterations is less than <code>options.FunctionTolerance</code>.</p>
MaxStallTime	<p>Maximum number of seconds without an improvement in the best known objective function value. Positive scalar with default <code>Inf</code>.</p>
MaxTime	<p>Maximum time in seconds that <code>particleswarm</code> runs. Default is <code>Inf</code>.</p>
MinNeighborsFraction	<p>Minimum adaptive neighborhood size, a scalar from <code>0</code> to <code>1</code>. Default is <code>0.25</code>. See "Particle Swarm Optimization Algorithm" on page 10-11.</p>
ObjectiveLimit	<p>Minimum objective value, a stopping criterion. Scalar, with default <code>-Inf</code>.</p>
OutputFcn	<p>Function handle or cell array of function handles. Output functions can read iterative data, and stop the solver. Default is <code>[]</code>. See "Output Function and Plot Function" on page 17-48.</p>

<code>PlotFcn</code>	Function name, function handle, or cell array of function handles. For custom plot functions, pass function handles. Plot functions can read iterative data, plot each iteration, and stop the solver. Default is <code>[]</code> . Available built-in plot function: <code>'pswplotbestf'</code> . See “Output Function and Plot Function” on page 17-48.
<code>SelfAdjustmentWeight</code>	Weighting of each particle’s best position when adjusting velocity. Finite scalar with default 1.49. See “Particle Swarm Optimization Algorithm” on page 10-11.
<code>SocialAdjustmentWeight</code>	Weighting of the neighborhood’s best position when adjusting velocity. Finite scalar with default 1.49. See “Particle Swarm Optimization Algorithm” on page 10-11.
<code>SwarmSize</code>	Number of particles in the swarm, an integer greater than 1. Default is <code>min(100, 10*nvars)</code> , where <code>nvars</code> is the number of variables.
<code>UseParallel</code>	Compute objective function in parallel when <code>true</code> . Default is <code>false</code> . See “Parallel or Vectorized Function Evaluation” on page 17-49.
<code>UseVectorized</code>	Compute objective function in vectorized fashion when <code>true</code> . Default is <code>false</code> . See “Parallel or Vectorized Function Evaluation” on page 17-49.

problem — Optimization problem

structure

Optimization problem, specified as a structure with the following fields.

<code>solver</code>	<code>'particleswarm'</code>
<code>objective</code>	Function handle to the objective function, or name of the objective function.
<code>nvars</code>	Number of variables in problem.
<code>lb</code>	Vector or array of lower bounds.
<code>ub</code>	Vector or array of upper bounds.
<code>options</code>	Options created by <code>optimoptions</code> .
<code>rngstate</code>	Optional state of the random number generator at the beginning of the solution process.

Data Types: `struct`**Output Arguments****x — Solution**

real vector

Solution, returned as a real vector that minimizes the objective function subject to any bound constraints.

fval — Objective value

real scalar

Objective value, returned as the real scalar `fun(x)`.

exitflag — Algorithm stopping condition

integer

Algorithm stopping condition, returned as an integer identifying the reason the algorithm stopped. The following lists the values of `exitflag` and the corresponding reasons `particleswarm` stopped.

1	Relative change in the objective value over the last <code>options.MaxStallIterations</code> iterations is less than <code>options.FunctionTolerance</code> .
0	Number of iterations exceeded <code>options.MaxIterations</code> .
-1	Iterations stopped by output function or plot function.
-2	Bounds are inconsistent: for some <code>i</code> , <code>lb(i) > ub(i)</code> .
-3	Best objective function value is below <code>options.ObjectiveLimit</code> .
-4	Best objective function value did not change within <code>options.MaxStallTime</code> seconds.
-5	Run time exceeded <code>options.MaxTime</code> seconds.

output — Solution process summary

structure

Solution process summary, returned as a structure containing information about the optimization process.

<code>iterations</code>	Number of solver iterations
<code>funccount</code>	Number of objective function evaluations.
<code>message</code>	Reason the algorithm stopped.
<code>hybridflag</code>	Exit flag from the hybrid function. Relates to the <code>HybridFcn</code> options.
<code>rngstate</code>	State of the default random number generator just before the algorithm started.

Algorithms

For a description of the particle swarm optimization algorithm, see “Particle Swarm Optimization Algorithm” on page 10-11.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for `particleswarm`.

Version History

Introduced in R2014b

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set the `'UseParallel'` option to `true`.

```
options = optimoptions('solvername','UseParallel',true)
```

For more information, see “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11.

See Also

ga | patternsearch | **Optimize**

Topics

“Optimize Using Particle Swarm” on page 10-5

“Particle Swarm Output Function” on page 10-8

“What Is Particle Swarm Optimization?” on page 10-2

“Solver-Based Optimization Problem Setup”

patternsearch

Find minimum of function using pattern search

Syntax

```
x = patternsearch(fun,x0)
x = patternsearch(fun,x0,A,b)
x = patternsearch(fun,x0,A,b,Aeq,beq)
x = patternsearch(fun,x0,A,b,Aeq,beq,lb,ub)
x = patternsearch(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
x = patternsearch(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
x = patternsearch(problem)
[x,fval] = patternsearch(____)
[x,fval,exitflag,output] = patternsearch(____)
```

Description

`x = patternsearch(fun,x0)` finds a local minimum, `x`, to the function handle `fun` that computes the values of the objective function. `x0` is a real vector specifying an initial point for the pattern search algorithm.

Note “Passing Extra Parameters” explains how to pass extra parameters to the objective function and nonlinear constraint functions, if necessary.

`x = patternsearch(fun,x0,A,b)` minimizes `fun` subject to the linear inequalities $A*x \leq b$. See “Linear Inequality Constraints”.

`x = patternsearch(fun,x0,A,b,Aeq,beq)` minimizes `fun` subject to the linear equalities $Aeq*x = beq$ and $A*x \leq b$. If no linear inequalities exist, set `A = []` and `b = []`.

`x = patternsearch(fun,x0,A,b,Aeq,beq,lb,ub)` defines a set of lower and upper bounds on the design variables in `x`, so that the solution is always in the range $lb \leq x \leq ub$. If no linear equalities exist, set `Aeq = []` and `beq = []`. If `x(i)` has no lower bound, set `lb(i) = -Inf`. If `x(i)` has no upper bound, set `ub(i) = Inf`.

`x = patternsearch(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)` subjects the minimization to the nonlinear inequalities $c(x)$ or equalities $ceq(x)$ defined in `nonlcon`. `patternsearch` optimizes `fun` such that $c(x) \leq 0$ and $ceq(x) = 0$. If no bounds exist, set `lb = []`, `ub = []`, or both.

`x = patternsearch(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)` minimizes `fun` with the optimization options specified in `options`. Use `optimoptions` to set these options. If there are no nonlinear inequality or equality constraints, set `nonlcon = []`.

`x = patternsearch(problem)` finds the minimum for `problem`, a structure described in `problem`.

`[x,fval] = patternsearch(____)`, for any syntax, returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag,output] = patternsearch(___)` additionally returns `exitflag`, a value that describes the exit condition of `patternsearch`, and a structure `output` with information about the optimization process.

Examples

Unconstrained Pattern Search Minimization

Minimize an unconstrained problem using the `patternsearch` solver.

Create the following two-variable objective function. On your MATLAB® path, save the following code to a file named `psobj.m`.

```
function y = psobj(x)
y = exp(-x(1)^2-x(2)^2)*(1+5*x(1) + 6*x(2) + 12*x(1)*cos(x(2)));
```

Set the objective function to `@psobj`.

```
fun = @psobj;
```

Find the minimum, starting at the point `[0,0]`.

```
x0 = [0,0];
x = patternsearch(fun,x0)
```

```
Optimization terminated: mesh size less than options.MeshTolerance.
```

```
x =
```

```
    -0.7037    -0.1860
```

Pattern Search with a Linear Inequality Constraint

Minimize a function subject to some linear inequality constraints.

Create the following two-variable objective function. On your MATLAB® path, save the following code to a file named `psobj.m`.

```
function y = psobj(x)
y = exp(-x(1)^2-x(2)^2)*(1+5*x(1) + 6*x(2) + 12*x(1)*cos(x(2)));
```

Set the objective function to `@psobj`.

```
fun = @psobj;
```

Set the two linear inequality constraints.

```
A = [-3, -2;
     -4, -7];
b = [-1; -8];
```

Find the minimum, starting at the point [0.5, -0.5].

```
x0 = [0.5, -0.5];
x = patternsearch(fun, x0, A, b)
```

Optimization terminated: mesh size less than options.MeshTolerance.

x =

```
5.2827 -1.8758
```

Pattern Search with Bounds

Find the minimum of a function that has only bound constraints.

Create the following two-variable objective function. On your MATLAB® path, save the following code to a file named `psobj.m`.

```
function y = psobj(x)
y = exp(-x(1)^2-x(2)^2)*(1+5*x(1) + 6*x(2) + 12*x(1)*cos(x(2)));
```

Set the objective function to `@psobj`.

```
fun = @psobj;
```

Find the minimum when $0 \leq x(1) \leq \infty$ and $-\infty \leq x(2) \leq -3$.

```
lb = [0, -Inf];
ub = [Inf, -3];
A = [];
b = [];
Aeq = [];
beq = [];
```

Find the minimum, starting at the point [1, -5].

```
x0 = [1, -5];
x = patternsearch(fun, x0, A, b, Aeq, beq, lb, ub)
```

Optimization terminated: mesh size less than options.MeshTolerance.

x =

```
0.1880 -3.0000
```

Pattern Search with Nonlinear Constraints

Find the minimum of a function subject to a nonlinear inequality constraint.

Create the following two-variable objective function. On your MATLAB® path, save the following code to a file named `psobj.m`.

```
function y = psobj(x)
y = exp(-x(1)^2-x(2)^2)*(1+5*x(1) + 6*x(2) + 12*x(1)*cos(x(2)));
```

Set the objective function to `@psobj`.

```
fun = @psobj;
```

Create the nonlinear constraint

$$\frac{xy}{2} + (x+2)^2 + \frac{(y-2)^2}{2} \leq 2.$$

To do so, on your MATLAB path, save the following code to a file named `ellipsetilt.m`.

```
function [c,ceq] = ellipsetilt(x)
ceq = [];
c = x(1)*x(2)/2 + (x(1)+2)^2 + (x(2)-2)^2/2 - 2;
```

Start `patternsearch` from the initial point `[-2, -2]`.

```
x0 = [-2, -2];
A = [];
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];
nonlcon = @ellipsetilt;
x = patternsearch(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
```

Optimization finished: mesh size less than options.MeshTolerance and constraint violation is less than options.ConstraintTolerance.

```
x =
```

```
    -1.5144    0.0875
```

Try Different `patternsearch` Algorithms

Sometimes the different `patternsearch` algorithms have noticeably different behavior. While it can be difficult to predict which algorithm works best for a problem, you can easily try different algorithms. For this example, use the `sawtoothxy` objective function, which is available when you

run this example, and which is described and plotted in "Find Global or Multiple Local Minima" on page 4-57.

type `sawtoothxy`

```
function f = sawtoothxy(x,y)
[t r] = cart2pol(x,y); % change to polar coordinates
h = cos(2*t - 1/2)/2 + cos(t) + 2;
g = (sin(r) - sin(2*r)/2 + sin(3*r)/3 - sin(4*r)/4 + 4) ...
    .*r.^2./(r+1);
f = g.*h;
end
```

To see the behavior of the different algorithms when minimizing this objective function, set some asymmetric bounds. Also set an initial point x_0 that is far from the true solution $sol = [0 \ 0]$, where $sawtoothxy(0,0) = 0$.

```
rng default
x0 = 12*randn(1,2);
lb = [-15,-26];
ub = [26,15];
fun = @(x)sawtoothxy(x(1),x(2));
```

Minimize the `sawtoothxy` function using the "classic" `patternsearch` algorithm.

```
optsc = optimoptions("patternsearch",Algorithm="classic");
[sol,fval,eflag,output] = patternsearch(fun,...
    x0,[],[],[],[],lb,ub,[],optsc)
```

Optimization terminated: mesh size less than options.MeshTolerance.

```
sol = 1x2
10-5 x
```

```
    0.9825    0
```

```
fval = 1.3278e-09
```

```
eflag = 1
```

```
output = struct with fields:
```

```
    function: @(x)sawtoothxy(x(1),x(2))
    problemtype: 'boundconstraints'
    pollmethod: 'gpspositivebasis2n'
    maxconstraint: 0
    searchmethod: []
    iterations: 52
    funccount: 168
    meshsize: 9.5367e-07
    rngstate: [1x1 struct]
    message: 'Optimization terminated: mesh size less than options.MeshTolerance.'
```

The "classic" algorithm reaches the global solution in 52 iterations and 168 function evaluations.

Try the "nups" algorithm.

```
rng default % For reproducibility
optsn = optimoptions("patternsearch",Algorithm="nups");
```

```
[sol,fval,eflag,output] = patternsearch(fun,...
    x0,[],[],[],[],lb,ub,[],optsn)

Optimization terminated: mesh size less than options.MeshTolerance.

sol = 1×2

    6.3204    15.0000

fval = 85.9256

eflag = 1

output = struct with fields:
    function: @(x)sawtoothxy(x(1),x(2))
    problemtype: 'boundconstraints'
    pollmethod: 'nups'
    maxconstraint: 0
    searchmethod: []
    iterations: 29
    funccount: 88
    meshsize: 7.1526e-07
    rngstate: [1×1 struct]
    message: 'Optimization terminated: mesh size less than options.MeshTolerance.'
```

This time the solver reaches a local solution in just 29 iterations and 88 function evaluations, but the solution is not the global solution.

Try using the "nups-mads" algorithm, which takes no steps in the coordinate directions.

```
rng default % For reproducibility
optsm = optimoptions("patternsearch",Algorithm="nups-mads");
[sol,fval,eflag,output] = patternsearch(fun,...
    x0,[],[],[],[],lb,ub,[],optsm)

Optimization terminated: mesh size less than options.MeshTolerance.

sol = 1×2
10-4 ×

    -0.5275    0.0806

fval = 1.5477e-08

eflag = 1

output = struct with fields:
    function: @(x)sawtoothxy(x(1),x(2))
    problemtype: 'boundconstraints'
    pollmethod: 'nups-mads'
    maxconstraint: 0
    searchmethod: []
    iterations: 55
    funccount: 189
    meshsize: 9.5367e-07
    rngstate: [1×1 struct]
```

```
message: 'Optimization terminated: mesh size less than options.MeshTolerance.'
```

This time, the solver reaches the global solution in 55 iterations and 189 function evaluations, which is similar to the 'classic' algorithm.

Pattern Search with Nondefault Options

Set options to observe the progress of the `patternsearch` solution process.

Create the following two-variable objective function. On your MATLAB path, save the following code to a file named `psobj.m`.

```
function y = psobj(x)
y = exp(-x(1)^2-x(2)^2)*(1+5*x(1) + 6*x(2) + 12*x(1)*cos(x(2)));
```

Set the objective function to `@psobj`.

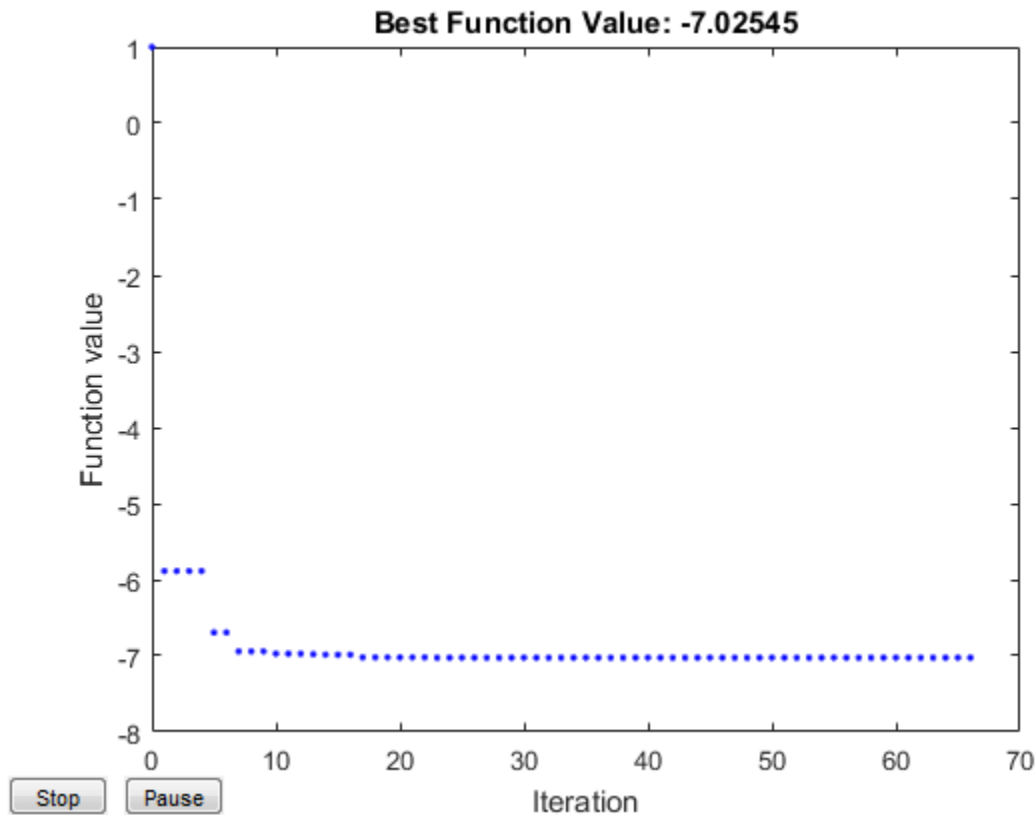
```
fun = @psobj;
```

Set `options` to give iterative display and to plot the objective function at each iteration.

```
options = optimoptions('patternsearch','Display','iter','PlotFcn',@psplotbestf);
```

Find the unconstrained minimum of the objective starting from the point `[0,0]`.

```
x0 = [0,0];
A = [];
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];
nonlcon = [];
x = patternsearch(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
```



Iter	f-count	f(x)	MeshSize	Method
0	1	1	1	
1	4	-5.88607	2	Successful Poll
2	8	-5.88607	1	Refine Mesh
3	12	-5.88607	0.5	Refine Mesh
4	16	-5.88607	0.25	Refine Mesh

(output trimmed)

63	218	-7.02545	1.907e-06	Refine Mesh
64	221	-7.02545	3.815e-06	Successful Poll
65	225	-7.02545	1.907e-06	Refine Mesh
66	229	-7.02545	9.537e-07	Refine Mesh

Optimization terminated: mesh size less than options.MeshTolerance.

x =

-0.7037 -0.1860

Obtain Function Value And Minimizing Point

Find a minimum value of a function and report both the location and value of the minimum.

Create the following two-variable objective function. On your MATLAB® path, save the following code to a file named `psobj.m`.

```
function y = psobj(x)
```

```
y = exp(-x(1)^2-x(2)^2)*(1+5*x(1) + 6*x(2) + 12*x(1)*cos(x(2)));
```

Set the objective function to @psobj.

```
fun = @psobj;
```

Find the unconstrained minimum of the objective, starting from the point [0,0]. Return both the location of the minimum, x, and the value of fun(x).

```
x0 = [0,0];
[x,fval] = patternsearch(fun,x0)
```

```
Optimization terminated: mesh size less than options.MeshTolerance.
```

```
x =
```

```
    -0.7037    -0.1860
```

```
fval =
```

```
    -7.0254
```

Obtain All Outputs

To examine the patternsearch solution process, obtain all outputs.

Create the following two-variable objective function. On your MATLAB® path, save the following code to a file named psobj.m.

```
function y = psobj(x)
```

```
y = exp(-x(1)^2-x(2)^2)*(1+5*x(1) + 6*x(2) + 12*x(1)*cos(x(2)));
```

Set the objective function to @psobj.

```
fun = @psobj;
```

Find the unconstrained minimum of the objective, starting from the point [0,0]. Return the solution, x, the objective function value at the solution, fun(x), the exit flag, and the output structure.

```
x0 = [0,0];
[x,fval,exitflag,output] = patternsearch(fun,x0)
```

```
Optimization terminated: mesh size less than options.MeshTolerance.
```

```
x =
```

```
    -0.7037    -0.1860
```

```
fval =  
    -7.0254  
  
exitflag =  
    1  
  
output =  
    struct with fields:  
        function: @psobj  
        problemtype: 'unconstrained'  
        pollmethod: 'gpspositivebasis2n'  
        maxconstraint: []  
        searchmethod: []  
        iterations: 66  
        funccount: 229  
        meshsize: 9.5367e-07  
        rngstate: [1x1 struct]  
        message: 'Optimization terminated: mesh size less than options.MeshTolerance.'
```

The `exitflag` is 1, indicating convergence to a local minimum.

The `output` structure includes information such as how many iterations `patternsearch` took, and how many function evaluations. Compare this output structure with the results from “Pattern Search with Nondefault Options” on page 18-139. In that example, you obtain some of this information, but did not obtain, for example, the number of function evaluations.

Input Arguments

fun — Function to be minimized

function handle | function name

Function to be minimized, specified as a function handle or function name. The `fun` function accepts a vector `x` and returns a real scalar `f`, which is the objective function evaluated at `x`.

You can specify `fun` as a function handle for a file

```
x = patternsearch(@myfun,x0)
```

Here, `myfun` is a MATLAB function such as

```
function f = myfun(x)  
f = ...           % Compute function value at x
```

`fun` can also be a function handle for an anonymous function

```
x = patternsearch(@(x)norm(x)^2,x0,A,b);
```

```
Example: fun = @(x)sin(x(1))*cos(x(2))
```

Data Types: char | function_handle | string

x0 — Initial point

real vector

Initial point, specified as a real vector. `patternsearch` uses the number of elements in `x0` to determine the number of variables that `fun` accepts.

Example: `x0 = [1,2,3,4]`

Data Types: `double`

A — Linear inequality constraints

real matrix

Linear inequality constraints, specified as a real matrix. `A` is an `M`-by-`nvars` matrix, where `M` is the number of inequalities.

`A` encodes the `M` linear inequalities

$$A*x \leq b,$$

where `x` is the column vector of `nvars` variables `x(:)`, and `b` is a column vector with `M` elements.

For example, to specify

$$\begin{aligned} x_1 + 2x_2 &\leq 10 \\ 3x_1 + 4x_2 &\leq 20 \\ 5x_1 + 6x_2 &\leq 30, \end{aligned}$$

give these constraints:

$$\begin{aligned} A &= [1,2;3,4;5,6]; \\ b &= [10;20;30]; \end{aligned}$$

Example: To specify that the control variables sum to 1 or less, give the constraints `A = ones(1,N)` and `b = 1`.

Data Types: `double`

b — Linear inequality constraints

real vector

Linear inequality constraints, specified as a real vector. `b` is an `M`-element vector related to the `A` matrix. If you pass `b` as a row vector, solvers internally convert `b` to the column vector `b(:)`.

`b` encodes the `M` linear inequalities

$$A*x \leq b,$$

where `x` is the column vector of `N` variables `x(:)`, and `A` is a matrix of size `M`-by-`N`.

For example, to specify

$$\begin{aligned} x_1 + 2x_2 &\leq 10 \\ 3x_1 + 4x_2 &\leq 20 \\ 5x_1 + 6x_2 &\leq 30, \end{aligned}$$

give these constraints:

```
A = [1,2;3,4;5,6];
b = [10;20;30];
```

Example: To specify that the control variables sum to 1 or less, give the constraints $A = \text{ones}(1, N)$ and $b = 1$.

Data Types: double

Aeq – Linear equality constraints

real matrix

Linear equality constraints, specified as a real matrix. **Aeq** is an M_e -by- $nvars$ matrix, where M_e is the number of equalities.

Aeq encodes the M_e linear equalities

$$\mathbf{Aeq} \cdot \mathbf{x} = \mathbf{beq},$$

where \mathbf{x} is the column vector of N variables $\mathbf{x}(:)$, and \mathbf{beq} is a column vector with M_e elements.

For example, to specify

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 10 \\ 2x_1 + 4x_2 + x_3 &= 20, \end{aligned}$$

give these constraints:

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```

Example: To specify that the control variables sum to 1, give the constraints $\mathbf{Aeq} = \text{ones}(1, N)$ and $\mathbf{beq} = 1$.

Data Types: double

beq – Linear equality constraints

real vector

Linear equality constraints, specified as a real vector. **beq** is an M_e -element vector related to the **Aeq** matrix. If you pass **beq** as a row vector, solvers internally convert **beq** to the column vector $\mathbf{beq}(:)$.

beq encodes the M_e linear equalities

$$\mathbf{Aeq} \cdot \mathbf{x} = \mathbf{beq},$$

where \mathbf{x} is the column vector of N variables $\mathbf{x}(:)$, and **Aeq** is a matrix of size M_e -by- N .

For example, to specify

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 10 \\ 2x_1 + 4x_2 + x_3 &= 20, \end{aligned}$$

give these constraints:

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```


Example: To specify that the control variables sum to 1, give the constraints $A_{eq} = \text{ones}(1, N)$ and $b_{eq} = 1$.

Data Types: double

lb — Lower bounds

real vector | real array

Lower bounds, specified as a real vector or real array. If the number of elements in x_0 is equal to that of lb , then lb specifies that

$$x(i) \geq lb(i)$$

for all i .

If $\text{numel}(lb) < \text{numel}(x_0)$, then lb specifies that

$$x(i) \geq lb(i)$$

for

$$1 \leq i \leq \text{numel}(lb)$$

In this case, solvers issue a warning.

Example: To specify that all control variables are positive, $lb = \text{zeros}(\text{size}(x_0))$

Data Types: double

ub — Upper bounds

real vector | real array

Upper bounds, specified as a real vector or real array. If the number of elements in x_0 is equal to that of ub , then ub specifies that

$$x(i) \leq ub(i)$$

for all i .

If $\text{numel}(ub) < \text{numel}(x_0)$, then ub specifies that

$$x(i) \leq ub(i)$$

for

$$1 \leq i \leq \text{numel}(ub)$$

In this case, solvers issue a warning.

Example: To specify that all control variables are less than one, $ub = \text{ones}(\text{size}(x_0))$

Data Types: double

nonlcon — Nonlinear constraints

function handle | function name

Nonlinear constraints, specified as a function handle or function name. `nonlcon` is a function that accepts a vector or array x and returns two arrays, $c(x)$ and $ceq(x)$.

- $c(x)$ is the array of nonlinear inequality constraints at x . `patternsearch` attempts to satisfy $c(x) \leq 0$ for all entries of c .
- $ceq(x)$ is the array of nonlinear equality constraints at x . `patternsearch` attempts to satisfy $ceq(x) = 0$ for all entries of ceq .

For example,

```
x = patternsearch(@myfun,x0,A,b,Aeq,beq,lb,ub,@mycon)
```

where `mycon` is a MATLAB function such as

```
function [c,ceq] = mycon(x)
c = ...      % Compute nonlinear inequalities at x.
ceq = ...    % Compute nonlinear equalities at x.
```

For more information, see “Nonlinear Constraints”.

Data Types: `char` | `function_handle` | `string`

options – Optimization options

object returned by `optimoptions` | structure

Optimization options, specified as an object returned by `optimoptions` (recommended), or a structure.

The following table describes the optimization options. `optimoptions` hides the options shown in *italics*; see “Options that `optimoptions` Hides” on page 17-65. {} denotes the default value. See option details in “Pattern Search Options” on page 17-7.

Options for patternsearch

Option	Description	Values
Algorithm	<p>Algorithm used by patternsearch. The Algorithm setting affects the available options. For algorithm details, see “How Pattern Search Polling Works” on page 6-27 and “Nonuniform Pattern Search (NUPS) Algorithm” on page 6-35.</p> <p>For examples of algorithm effects, see “Explore patternsearch Algorithms” on page 6-103 and “Explore patternsearch Algorithms in Optimize Live Editor Task” on page 6-107.</p>	{"classic"} "nups" "nups-gps" "nups-mads"
Cache	<p>With Cache set to "on", patternsearch keeps a history of the mesh points it polls. At subsequent iterations, patternsearch does not poll points close to those already polled. Use this option if patternsearch runs slowly while computing the objective function. If the objective function is stochastic, do not use this option.</p> <hr/> <p>Note Cache does not work when you run the solver in parallel.</p>	"on" {"off"}
CacheSize	Size of the history.	Positive scalar {1e4}
CacheTol	Largest distance from the current mesh point to any point in the history in order for patternsearch to avoid polling the current point. Use if the Cache option is set to "on".	Positive scalar {eps}
ConstraintTolerance	<p>Tolerance on constraints.</p> <p>For an options structure, use TolCon.</p>	Positive scalar {1e-6}
Display	Level of display, meaning how much information patternsearch returns to the Command Line during the solution process.	"off" "iter" "diagnose" {"final"}
FunctionTolerance	<p>Tolerance on the function. Iterations stop if the change in function value is less than FunctionTolerance and the mesh size is less than StepTolerance. This option does not apply to MADS (mesh adaptive direct search) polling.</p> <p>For an options structure, use TolFun.</p>	Positive scalar {1e-6}
InitialMeshSize	Initial mesh size for the algorithm. See “How Pattern Search Polling Works” on page 6-27.	Positive scalar {1.0}
InitialPenalty	Initial value of the penalty parameter. See “Nonlinear Constraint Solver Algorithm for Pattern Search” on page 6-46.	Positive scalar {10}

Option	Description	Values
MaxFunctionEvaluations	Maximum number of objective function evaluations. For an options structure, use MaxFunEvals.	Positive integer {"2000*numberOfVariables"}, where numberOfVariables is the number of problem variables
MaxIterations	Maximum number of iterations. For an options structure, use MaxIter.	Positive integer {"100*numberOfVariables"}, where numberOfVariables is the number of problem variables
MaxMeshSize	Maximum mesh size used in a poll or search step. See "How Pattern Search Polling Works" on page 6-27.	Positive scalar {Inf}
MaxTime	Total time (in seconds) allowed for optimization. For an options structure, use TimeLimit.	Positive scalar {Inf}
MeshContractionFactor	Mesh contraction factor for an unsuccessful iteration. This option applies only when Algorithm is "classic". For an options structure, use MeshContraction.	Positive scalar {0.5}
MeshExpansionFactor	Mesh expansion factor for a successful iteration. This option applies only when Algorithm is "classic". For an options structure, use MeshExpansion.	Positive scalar {2.0}
MeshRotate	Flag to rotate the pattern before declaring a point to be optimum. See "Mesh Options" on page 17-15. This option applies only when Algorithm is "classic".	"off" {"on"}
MeshTolerance	Tolerance on the mesh size. For an options structure, use TolMesh.	Positive scalar {1e-6}
OutputFcn	Function called by an optimization function at each iteration. Specify as a function handle or a cell array of function handles. For an options structure, use OutputFcns.	Function handle or cell array of function handles on page 17-17 {[]}
PenaltyFactor	Penalty update parameter. See "Nonlinear Constraint Solver Algorithm for Pattern Search" on page 6-46.	Positive scalar {100}
PlotFcn	Plots of output from the pattern search. Specify as the name of a built-in plot function, a function handle, or a cell array of names of built-in plot functions or function handles. For an options structure, use PlotFcns.	{[]} "psplotbestf" "psplotfunccount" "psplotmeshsize" "psplotbestx" "psplotmaxconstr" custom plot function on page 17-8

Option	Description	Values
<i>PlotInterval</i>	Number of iterations for plots. 1 means plot every iteration, 2 means plot every other iteration, and so on.	positive integer {1}
<i>PollMethod</i>	<p>Polling strategy used in the pattern search.</p> <p>This option applies only when <code>Algorithm</code> is "classic".</p> <hr/> <p>Note You cannot use MADS polling when the problem has linear equality constraints.</p>	{"GPSPositiveBasis2N" } "GPSPositiveBasisNp1" "GSSPositiveBasis2N" "GSSPositiveBasisNp1" "MADSPositiveBasis2N" "MADSPositiveBasisNp1"
<i>PollOrderAlgorithm</i>	<p>Order of the poll directions in the pattern search.</p> <p>This option applies only when <code>Algorithm</code> is "classic".</p> <p>For an options structure, use <code>PollingOrder</code>.</p>	"Random" "Success" {"Consecutive"}
<i>ScaleMesh</i>	<p>Automatic scaling of variables.</p> <p>For an options structure, use <code>ScaleMesh</code> = "on" or "off".</p>	{true} false
<i>SearchFcn</i>	<p>Type of search used in the pattern search. Specify as a name or a function handle.</p> <p>For an options structure, use <code>SearchMethod</code>.</p>	"GPSPositiveBasis2N" "GPSPositiveBasisNp1" "GSSPositiveBasis2N" "GSSPositiveBasisNp1" "MADSPositiveBasis2N" "MADSPositiveBasisNp1" "searchga" "searchlhs" "searchneldermead" "rbfsurrogate" {} custom search function on page 17-12
<i>StepTolerance</i>	<p>Tolerance on the variable. Iterations stop if both the change in position and the mesh size are less than <code>StepTolerance</code>. This option does not apply to MADS polling.</p> <p>For an options structure, use <code>TolX</code>.</p>	Positive scalar {1e-6}
<i>TolBind</i>	Binding tolerance. See "Constraint Parameters" on page 17-16.	Positive scalar {1e-3}

Option	Description	Values
UseCompletePoll	<p>Flag to complete the poll around the current point. See “How Pattern Search Polling Works” on page 6-27.</p> <p>This option applies only when Algorithm is "classic".</p> <hr/> <p>Note For the "classic" algorithm, you must set UseCompletePoll to true for vectorized or parallel polling. Similarly, set UseCompleteSearch to true for vectorized or parallel searching.</p> <p>Beginning in R2019a, when you set the UseParallel option to true, patternsearch internally overrides the UseCompletePoll setting to true so that the function polls in parallel.</p> <hr/> <p>For an options structure, use CompletePoll = "on" or "off".</p>	true {false}
UseCompleteSearch	<p>Flag to complete the search around the current point when the search method is a poll method. See “Searching and Polling” on page 6-37.</p> <p>This option applies only when Algorithm is "classic".</p> <hr/> <p>Note For the "classic" algorithm, you must set UseCompleteSearch to true for vectorized or parallel searching.</p> <hr/> <p>For an options structure, use CompleteSearch = "on" or "off".</p>	true {false}

Option	Description	Values
UseParallel	<p>Flag to compute objective and nonlinear constraint functions in parallel. See “Vectorized and Parallel Options” on page 17-19 and “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11.</p> <hr/> <p>Note For the "classic" algorithm, you must set <code>UseCompletePoll</code> to <code>true</code> for vectorized or parallel polling. Similarly, set <code>UseCompleteSearch</code> to <code>true</code> for vectorized or parallel searching.</p> <p>Beginning in R2019a, when you set the <code>UseParallel</code> option to <code>true</code>, <code>patternsearch</code> internally overrides the <code>UseCompletePoll</code> setting to <code>true</code> so that the function polls in parallel.</p> <hr/> <p>Note <code>Cache</code> does not work when you run the solver in parallel.</p>	<code>true</code> <code>{false}</code>
UseVectorized	<p>Specifies whether functions are vectorized. See “Vectorized and Parallel Options” on page 17-19 and “Vectorize the Objective and Constraint Functions” on page 6-83.</p> <hr/> <p>Note For the "classic" algorithm, you must set <code>UseCompletePoll</code> to <code>true</code> for vectorized or parallel polling. Similarly, set <code>UseCompleteSearch</code> to <code>true</code> for vectorized or parallel searching.</p> <hr/> <p>For an options structure, use <code>Vectorized = "on"</code> or <code>"off"</code>.</p>	<code>true</code> <code>{false}</code>

```
Example: options =
optimoptions("patternsearch",MaxIterations=150,MeshTolerance=1e-4)
```

problem — Problem structure

structure

Problem structure, specified as a structure with the following fields:

- `objective` — Objective function
- `x0` — Starting point
- `Aineq` — Matrix for linear inequality constraints
- `bineq` — Vector for linear inequality constraints
- `Aeq` — Matrix for linear equality constraints
- `beq` — Vector for linear equality constraints
- `lb` — Lower bound for `x`

- `ub` — Upper bound for `x`
- `nonlcon` — Nonlinear constraint function
- `solver` — 'patternsearch'
- `options` — Options created with `optimoptions` or a structure
- `rngstate` — Optional field to reset the state of the random number generator

Note All fields in `problem` are required except for `rngstate`.

Data Types: `struct`

Output Arguments

x — Solution

real vector

Solution, returned as a real vector. The size of `x` is the same as the size of `x0`. When `exitflag` is positive, `x` is typically a local solution to the problem.

fval — Objective function value at the solution

real number

Objective function value at the solution, returned as a real number. Generally, `fval = fun(x)`.

exitflag — Reason patternsearch stopped

integer

Reason patternsearch stopped, returned as an integer.

Exit Flag	Meaning
1	Without nonlinear constraints — The magnitude of the mesh size is less than the specified tolerance, and the constraint violation is less than <code>ConstraintTolerance</code> . With nonlinear constraints — The magnitude of the complementarity measure (defined after this table) is less than <code>sqrt(ConstraintTolerance)</code> , the subproblem is solved using a mesh finer than <code>MeshTolerance</code> , and the constraint violation is less than <code>ConstraintTolerance</code> .
2	The change in <code>x</code> and the mesh size are both less than the specified tolerance, and the constraint violation is less than <code>ConstraintTolerance</code> .
3	The change in <code>fval</code> and the mesh size are both less than the specified tolerance, and the constraint violation is less than <code>ConstraintTolerance</code> .
4	The magnitude of the step is smaller than machine precision, and the constraint violation is less than <code>ConstraintTolerance</code> .
0	The maximum number of function evaluations or iterations is reached.
-1	Optimization terminated by an output function or plot function.
-2	No feasible point found.

In the nonlinear constraint solver, the complementarity measure is the norm of the vector whose elements are $c_i\lambda_i$, where c_i is the nonlinear inequality constraint violation, and λ_i is the corresponding Lagrange multiplier.

output — Information about the optimization process

structure

Information about the optimization process, returned as a structure with these fields:

- `function` — Objective function.
- `problemtype` — Problem type, one of:
 - `'unconstrained'`
 - `'boundconstraints'`
 - `'linearconstraints'`
 - `'nonlinearconstr'`
- `pollmethod` — Polling technique.
- `searchmethod` — Search technique used, if any.
- `iterations` — Total number of iterations.
- `funccount` — Total number of function evaluations.
- `meshsize` — Mesh size at x .
- `maxconstraint` — Maximum constraint violation, if any.
- `rngstate` — State of the MATLAB random number generator, just before the algorithm started. You can use the values in `rngstate` to reproduce the output when you use a random search method or random poll method. See “Reproduce Results” on page 8-67, which discusses the identical technique for `ga`.
- `message` — Reason why the algorithm terminated.

Algorithms

By default and in the absence of linear constraints, `patternsearch` looks for a minimum based on an adaptive mesh that is aligned with the coordinate directions. See “What Is Direct Search?” on page 6-2 and “How Pattern Search Polling Works” on page 6-27.

When you set the `Algorithm` option to `"nups"` or one of its variants, `patternsearch` uses the algorithm described in “Nonuniform Pattern Search (NUPS) Algorithm” on page 6-35. This algorithm is different from the default algorithm in several ways; for example, it has fewer options to set.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for `patternsearch`.

Version History

Introduced before R2006a

References

- [1] Audet, Charles, and J. E. Dennis Jr. "Analysis of Generalized Pattern Searches." *SIAM Journal on Optimization*. Volume 13, Number 3, 2003, pp. 889-903.
- [2] Conn, A. R., N. I. M. Gould, and Ph. L. Toint. "A Globally Convergent Augmented Lagrangian Barrier Algorithm for Optimization with General Inequality Constraints and Simple Bounds." *Mathematics of Computation*. Volume 66, Number 217, 1997, pp. 261-288.
- [3] Abramson, Mark A. *Pattern Search Filter Algorithms for Mixed Variable General Constrained Optimization Problems*. Ph.D. Thesis, Department of Computational and Applied Mathematics, Rice University, August 2002.
- [4] Abramson, Mark A., Charles Audet, J. E. Dennis, Jr., and Sebastien Le Digabel. "ORTHOMADS: A deterministic MADS instance with orthogonal directions." *SIAM Journal on Optimization*. Volume 20, Number 2, 2009, pp. 948-966.
- [5] Kolda, Tamara G., Robert Michael Lewis, and Virginia Torczon. "Optimization by direct search: new perspectives on some classical and modern methods." *SIAM Review*. Volume 45, Issue 3, 2003, pp. 385-482.
- [6] Kolda, Tamara G., Robert Michael Lewis, and Virginia Torczon. "A generating set direct search augmented Lagrangian algorithm for optimization with a combination of general and linear constraints." Technical Report SAND2006-5315, Sandia National Laboratories, August 2006.
- [7] Lewis, Robert Michael, Anne Shepherd, and Virginia Torczon. "Implementing generating set search methods for linearly constrained minimization." *SIAM Journal on Scientific Computing*. Volume 29, Issue 6, 2007, pp. 2507-2530.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set the 'UseParallel' option to true.

```
options = optimoptions('solvername','UseParallel',true)
```

For more information, see "How to Use Parallel Processing in Global Optimization Toolbox" on page 16-11.

See Also

ga | optimoptions | paretosearch | **Optimize**

Topics

- "Optimize Using the GPS Algorithm" on page 6-3
- "Coding and Minimizing an Objective Function Using Pattern Search" on page 6-10
- "Constrained Minimization Using Pattern Search, Solver-Based" on page 6-14
- "Effects of Pattern Search Options" on page 6-18
- "Optimize ODEs in Parallel" on page 6-87
- "Pattern Search Climbs Mount Washington" on page 6-52
- "Optimization Workflow" on page 1-28
- "What Is Direct Search?" on page 6-2

"Pattern Search Terminology" on page 6-24
"How Pattern Search Polling Works" on page 6-27
"Polling Types" on page 6-59
"Search and Poll" on page 6-42

psoptimget

(Not recommended) Obtain values of pattern search options structure

Note `psoptimget` is not recommended. Instead, query options using dot notation. For more information, see “Compatibility Considerations”.

Syntax

```
val = psoptimget(options,'name')
val = psoptimget(options,'name',default)
```

Description

`val = psoptimget(options,'name')` returns the value of the parameter `name` from the pattern search options structure `options`. `psoptimget(options,'name')` returns an empty matrix `[]` if the value of `name` is not specified in `options`. It is only necessary to type enough leading characters of `name` to uniquely identify it. `psoptimget` ignores case in parameter names.

`val = psoptimget(options,'name',default)` returns the value of the parameter `name` from the pattern search options structure `options`, but returns `default` if the parameter is not specified (as in `[]`) in `options`.

Examples

```
opts = psoptimset('TolX',1e-4);
val = psoptimget(opts,'TolX')
```

returns `val = 1e-4`.

Version History

Introduced before R2006a

R2018b: psoptimget is not recommended

Not recommended starting in R2018b

To query options, the `gaoptimget`, `psoptimget`, and `saoptimget` functions are not recommended. Instead, use dot notation. For example, to see the setting of the `Display` option in `opts`,

```
displayopt = opts.Display
% instead of
displayopt = gaoptimget(opts,'Display')
```

Using automatic code completions, dot notation takes fewer keystrokes: `displayopt = opts.D`
Tab.

There are no plans to remove `gaoptimget`, `psoptimget`, and `saoptimget` at this time.

See Also

patternsearch

Topics

“Pattern Search Options” on page 17-7

psoptimset

(Not recommended) Create pattern search options structure

Note `psoptimset` is not recommended. Use `optimoptions` instead. For more information, see “Compatibility Considerations”.

Syntax

```
psoptimset
options = psoptimset
options = psoptimset(@patternsearch)
options = psoptimset('param1',value1,'param2',value2,...)
options = psoptimset(olddopts,'param1',value1,...)
options = psoptimset(olddopts,newopts)
```

Description

`psoptimset` with no input or output arguments displays a complete list of parameters with their valid values.

`options = psoptimset` (with no input arguments) creates a structure called `options` that contains the options, or *parameters*, for `patternsearch`, and sets parameters to `[]`, indicating `patternsearch` uses the default values.

`options = psoptimset(@patternsearch)` creates a structure called `options` that contains the default values for `patternsearch`.

`options = psoptimset('param1',value1,'param2',value2,...)` creates a structure `options` and sets the value of 'param1' to `value1`, 'param2' to `value2`, and so on. Any unspecified parameters are set to their default values. It is sufficient to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`options = psoptimset(olddopts,'param1',value1,...)` creates a copy of `olddopts`, modifying the specified parameters with the specified values.

`options = psoptimset(olddopts,newopts)` combines an existing options structure, `olddopts`, with a new options structure, `newopts`. Any parameters in `newopts` with nonempty values overwrite the corresponding old parameters in `olddopts`.

Options

The following table lists the options you can set with `psoptimset`. See “Pattern Search Options” on page 17-7 for a complete description of the options and their values. Values in `{}` denote the default value. You can also view the optimization parameters and defaults by typing `psoptimset` at the command line.

`optimoptions` hides the options listed in *italics*, but `psoptimset` does not. See “Options that `optimoptions` Hides” on page 17-65.

Options for patternsearch

Option	Description	Values
Algorithm	<p>Algorithm used by patternsearch. The Algorithm setting affects the available options. For algorithm details, see “How Pattern Search Polling Works” on page 6-27 and “Nonuniform Pattern Search (NUPS) Algorithm” on page 6-35.</p> <p>For examples of algorithm effects, see “Explore patternsearch Algorithms” on page 6-103 and “Explore patternsearch Algorithms in Optimize Live Editor Task” on page 6-107.</p>	{"classic"} "nups" "nups-gps" "nups-mads"
Cache	<p>With Cache set to "on", patternsearch keeps a history of the mesh points it polls. At subsequent iterations, patternsearch does not poll points close to those already polled. Use this option if patternsearch runs slowly while computing the objective function. If the objective function is stochastic, do not use this option.</p> <hr/> <p>Note Cache does not work when you run the solver in parallel.</p>	"on" {"off"}
CacheSize	Size of the history.	Positive scalar {1e4}
CacheTol	Largest distance from the current mesh point to any point in the history in order for patternsearch to avoid polling the current point. Use if the Cache option is set to "on".	Positive scalar {eps}
ConstraintTolerance	<p>Tolerance on constraints.</p> <p>For an options structure, use TolCon.</p>	Positive scalar {1e-6}
Display	Level of display, meaning how much information psoptimset returns to the Command Line during the solution process.	"off" "iter" "diagnose" {"final"}
FunctionTolerance	<p>Tolerance on the function. Iterations stop if the change in function value is less than FunctionTolerance and the mesh size is less than StepTolerance. This option does not apply to MADS (mesh adaptive direct search) polling.</p> <p>For an options structure, use TolFun.</p>	Positive scalar {1e-6}
InitialMeshSize	Initial mesh size for the algorithm. See “How Pattern Search Polling Works” on page 6-27.	Positive scalar {1.0}
InitialPenalty	Initial value of the penalty parameter. See “Nonlinear Constraint Solver Algorithm for Pattern Search” on page 6-46.	Positive scalar {10}

Option	Description	Values
MaxFunctionEvaluations	Maximum number of objective function evaluations. For an options structure, use MaxFunEvals.	Positive integer {"2000*numberOfVariables"}, where numberOfVariables is the number of problem variables
MaxIterations	Maximum number of iterations. For an options structure, use MaxIter.	Positive integer {"100*numberOfVariables"}, where numberOfVariables is the number of problem variables
MaxMeshSize	Maximum mesh size used in a poll or search step. See "How Pattern Search Polling Works" on page 6-27.	Positive scalar {Inf}
MaxTime	Total time (in seconds) allowed for optimization. For an options structure, use TimeLimit.	Positive scalar {Inf}
MeshContractionFactor	Mesh contraction factor for an unsuccessful iteration. This option applies only when Algorithm is "classic". For an options structure, use MeshContraction.	Positive scalar {0.5}
MeshExpansionFactor	Mesh expansion factor for a successful iteration. This option applies only when Algorithm is "classic". For an options structure, use MeshExpansion.	Positive scalar {2.0}
MeshRotate	Flag to rotate the pattern before declaring a point to be optimum. See "Mesh Options" on page 17-15. This option applies only when Algorithm is "classic".	"off" {"on"}
MeshTolerance	Tolerance on the mesh size. For an options structure, use TolMesh.	Positive scalar {1e-6}
OutputFcn	Function called by an optimization function at each iteration. Specify as a function handle or a cell array of function handles. For an options structure, use OutputFcns.	Function handle or cell array of function handles on page 17-17 {[]}
PenaltyFactor	Penalty update parameter. See "Nonlinear Constraint Solver Algorithm for Pattern Search" on page 6-46.	Positive scalar {100}
PlotFcn	Plots of output from the pattern search. Specify as the name of a built-in plot function, a function handle, or a cell array of names of built-in plot functions or function handles. For an options structure, use PlotFcns.	{[]} "psplotbestf" "psplotfunccount" "psplotmeshsize" "psplotbestx" "psplotmaxconstr" custom plot function on page 17-8

Option	Description	Values
<i>PlotInterval</i>	Number of iterations for plots. 1 means plot every iteration, 2 means plot every other iteration, and so on.	positive integer {1}
<i>PollMethod</i>	<p>Polling strategy used in the pattern search.</p> <p>This option applies only when <code>Algorithm</code> is "classic".</p> <hr/> <p>Note You cannot use MADS polling when the problem has linear equality constraints.</p>	{"GPSPositiveBasis2N" } "GPSPositiveBasisNp1" "GSSPositiveBasis2N" "GSSPositiveBasisNp1" "MADSPositiveBasis2N" "MADSPositiveBasisNp1"
<i>PollOrderAlgorithm</i>	<p>Order of the poll directions in the pattern search.</p> <p>This option applies only when <code>Algorithm</code> is "classic".</p> <p>For an options structure, use <code>PollingOrder</code>.</p>	"Random" "Success" {"Consecutive"}
<i>ScaleMesh</i>	<p>Automatic scaling of variables.</p> <p>For an options structure, use <code>ScaleMesh</code> = "on" or "off".</p>	{true} false
<i>SearchFcn</i>	<p>Type of search used in the pattern search. Specify as a name or a function handle.</p> <p>For an options structure, use <code>SearchMethod</code>.</p>	"GPSPositiveBasis2N" "GPSPositiveBasisNp1" "GSSPositiveBasis2N" "GSSPositiveBasisNp1" "MADSPositiveBasis2N" "MADSPositiveBasisNp1" "searchga" "searchlhs" "searchneldermead" "rbfsurrogate" {} custom search function on page 17-12
<i>StepTolerance</i>	<p>Tolerance on the variable. Iterations stop if both the change in position and the mesh size are less than <code>StepTolerance</code>. This option does not apply to MADS polling.</p> <p>For an options structure, use <code>TolX</code>.</p>	Positive scalar {1e-6}
<i>TolBind</i>	Binding tolerance. See "Constraint Parameters" on page 17-16.	Positive scalar {1e-3}

Option	Description	Values
UseCompletePoll	<p>Flag to complete the poll around the current point. See “How Pattern Search Polling Works” on page 6-27.</p> <p>This option applies only when Algorithm is "classic".</p> <hr/> <p>Note For the "classic" algorithm, you must set UseCompletePoll to true for vectorized or parallel polling. Similarly, set UseCompleteSearch to true for vectorized or parallel searching.</p> <p>Beginning in R2019a, when you set the UseParallel option to true, patternsearch internally overrides the UseCompletePoll setting to true so that the function polls in parallel.</p> <hr/> <p>For an options structure, use CompletePoll = "on" or "off".</p>	true {false}
UseCompleteSearch	<p>Flag to complete the search around the current point when the search method is a poll method. See “Searching and Polling” on page 6-37.</p> <p>This option applies only when Algorithm is "classic".</p> <hr/> <p>Note For the "classic" algorithm, you must set UseCompleteSearch to true for vectorized or parallel searching.</p> <hr/> <p>For an options structure, use CompleteSearch = "on" or "off".</p>	true {false}

Option	Description	Values
UseParallel	<p>Flag to compute objective and nonlinear constraint functions in parallel. See “Vectorized and Parallel Options” on page 17-19 and “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11.</p> <hr/> <p>Note For the "classic" algorithm, you must set UseCompletePoll to true for vectorized or parallel polling. Similarly, set UseCompleteSearch to true for vectorized or parallel searching.</p> <p>Beginning in R2019a, when you set the UseParallel option to true, patternsearch internally overrides the UseCompletePoll setting to true so that the function polls in parallel.</p> <hr/> <p>Note Cache does not work when you run the solver in parallel.</p>	true {false}
UseVectorized	<p>Specifies whether functions are vectorized. See “Vectorized and Parallel Options” on page 17-19 and “Vectorize the Objective and Constraint Functions” on page 6-83.</p> <hr/> <p>Note For the "classic" algorithm, you must set UseCompletePoll to true for vectorized or parallel polling. Similarly, set UseCompleteSearch to true for vectorized or parallel searching.</p> <hr/> <p>For an options structure, use Vectorized = "on" or "off".</p>	true {false}

Version History

Introduced before R2006a

R2018b: psoptimset is not recommended

Not recommended starting in R2018b

To set options, the gaoptimset, psoptimset, and saoptimset functions are not recommended. Instead, use optimoptions.

The main difference between using optimoptions and the other functions is that you include the solver name as the first argument in optimoptions. For example, to set an iterative display in ga:

```
options = optimoptions('ga', 'Display', 'iter');
% instead of
options = gaoptimset('Display', 'iter');
```

The other difference is that some option names have changed. You can continue to use the old names in `optimoptions`. For details, see “Options Changes in R2016a” on page 17-65.

`optimoptions` offers these advantages over the other functions:

- `optimoptions` provides better automatic code suggestions and completions, especially in the Live Editor.
- You can use a single option-setting function instead of a variety of functions.

There are no plans to remove `gaoptimset`, `psoptimset`, and `saoptimset` at this time.

See Also

`optimoptions` | `patternsearch`

RandomStartPointSet

Random start points

Description

A `RandomStartPointSet` object describes how to generate a set of pseudorandom points for use with `MultiStart`. A `RandomStartPointSet` object does not contain points. It contains parameters for generating the points when `MultiStart` runs or when you use the `list` function.

Creation

Syntax

```
rs = RandomStartPointSet
rs = RandomStartPointSet(Name, Value)
rs = RandomStartPointSet(oldrs, Name, Value)
```

Description

`rs = RandomStartPointSet` creates a default `RandomStartPointSet` object.

`rs = RandomStartPointSet(Name, Value)` sets properties using name-value pairs.

`rs = RandomStartPointSet(oldrs, Name, Value)` creates a copy of the `oldrs` `RandomStartPointSet` object, and sets properties using name-value pairs.

Properties

ArtificialBound — Absolute value of default bounds for unbounded components

1000 (default) | positive scalar

Absolute value of the default bounds for unbounded components, specified as a positive scalar.

Example: `1e2`

Data Types: `double`

NumStartPoints — Number of start points

10 (default) | positive integer

Number of start points, specified as a positive integer.

Example: `40`

Data Types: `double`

Object Functions

`list` List start points

Examples

Create Default RandomStartPointSet

Create a default RandomStartPointSet object.

```
rs = RandomStartPointSet
rs =
  RandomStartPointSet with properties:
    NumStartPoints: 10
    ArtificialBound: 1000
```

Create RandomStartPointSet

Create a RandomStartPointSet object for 40 points.

```
rs = RandomStartPointSet('NumStartPoints',40);
```

Create a problem with 3-D variables, lower bounds of 0, and upper bounds of [10,20,30].

```
problem = createOptimProblem('fmincon','x0',rand(3,1),'lb',zeros(3,1),'ub',[10,20,30]);
```

Generate a random set of 40 points consistent with the problem.

```
points = list(rs,problem);
```

Examine the maximum and minimum generated components.

```
largest = max(max(points))
```

```
largest = 29.8840
```

```
smallest = min(min(points))
```

```
smallest = 0.1390
```

Update RandomStartPointSet

Create a RandomStartPointSet object that generates 50 points.

```
rs = RandomStartPointSet('NumStartPoints',50)
```

```
rs =
  RandomStartPointSet with properties:
```

```
    NumStartPoints: 50
    ArtificialBound: 1000
```

Update rs to use 100 points and an artificial bound of 1e4.

```
rs = RandomStartPointSet(rs, 'NumStartPoints', 100, 'ArtificialBound', 1e4)
```

```
rs =  
RandomStartPointSet with properties:  
  
    NumStartPoints: 100  
    ArtificialBound: 10000
```

You can also update properties using dot notation.

```
rs.ArtificialBound = 500
```

```
rs =  
RandomStartPointSet with properties:  
  
    NumStartPoints: 100  
    ArtificialBound: 500
```

Version History

Introduced in R2010a

See Also

[MultiStart](#) | [CustomStartPointSet](#) | [list](#)

Topics

“Workflow for GlobalSearch and MultiStart” on page 4-3

run

Run multiple-start solver

Syntax

```
x = run(gs,problem)
x = run(ms,problem,k)
x = run(ms,problem,startpts)
[x,fval] = run(____)
[x,fval,exitflag,output] = run(____)
[x,fval,exitflag,output,solutions] = run(____)
```

Description

`x = run(gs,problem)` runs `GlobalSearch` to find a solution or multiple local solutions to `problem`.

`x = run(ms,problem,k)` runs `MultiStart` on `k` start points to find a solution or multiple local solutions to `problem`.

`x = run(ms,problem,startpts)` runs `MultiStart` on `problem` from the start points described in `startpts`.

`[x,fval] = run(____)` returns the objective function value at `x`, the best point found, using any of the arguments in the previous syntaxes. For the `lsqcurvefit` and `lsqnonlin` local solvers, `fval` contains the squared norm of the residual.

`[x,fval,exitflag,output] = run(____)` also returns an exit flag describing the return condition, and an output structure describing the iterations of the run.

`[x,fval,exitflag,output,solutions] = run(____)` also returns a vector of solutions containing the distinct local minima found during the run.

Examples

Run GlobalSearch on Multidimensional Problem

Create an optimization problem that has several local minima, and try to find the global minimum using `GlobalSearch`. The objective is the six-hump camel back problem (see “Run the Solver” on page 4-13).

```
rng default % For reproducibility
gs = GlobalSearch;
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3]);
x = run(gs,problem)
```


GlobalSearch stopped because it analyzed all the trial points.

All 8 local solver runs converged with a positive local solver exit flag.

```
x = 1x2
    -0.0898    0.7127
```

You can request the objective function value at `x` when you call `run` by using the following syntax:

```
[x,fval] = run(gs,problem)
```

However, if you neglected to request `fval`, you can still compute the objective function value at `x`.

```
fval = sixmin(x)
fval = -1.0316
```

Run a Multiple Start Solver

Use a default `MultiStart` object to solve the six-hump camel back problem (see “Run the Solver” on page 4-13).

```
rng default % For reproducibility
ms = MultiStart;
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3]);
[x,fval,exitflag,oupt,solutions] = run(ms,problem,30);
```

`MultiStart` completed the runs from all start points.

All 30 local solver runs converged with a positive local solver exit flag.

Examine the best function value and the location where the best function value is attained.

```
fprintf('The best function value is %f.\n',fval)
```

The best function value is -1.031628.

```
fprintf('The location where this value is attained is [%f,%f].',x)
```

The location where this value is attained is [-0.089842,0.712656].

Run MultiStart from a Regular Array

Create a set of initial 2-D points for `MultiStart` in the range `[-3,3]` for each component.

```
v = -3:0.5:3;
[X,Y] = meshgrid(v);
ptmatrix = [X(:),Y(:)];
tpoints = CustomStartPointSet(ptmatrix);
```

Find the point that minimizes the six-hump camel back problem (see “Run the Solver” on page 4-13) by starting MultiStart at the points in tpoints.

```
rng default % For reproducibility
ms = MultiStart;
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3]);
x = run(ms,problem,tpoints)
```

MultiStart completed the runs from all start points.

All 169 local solver runs converged with a positive local solver exit flag.

```
x = 1x2
    -0.0898    0.7127
```

Examine GlobalSearch Process

Create an optimization problem that has several local minima, and try to find the global minimum using GlobalSearch. The objective is the six-hump camel back problem (see “Run the Solver” on page 4-13).

```
rng default % For reproducibility
gs = GlobalSearch;
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3]);
[x,fval,exitflag,output,solutions] = run(gs,problem);
```

GlobalSearch stopped because it analyzed all the trial points.

All 8 local solver runs converged with a positive local solver exit flag.

To understand what GlobalSearch did to solve this problem, examine the output structure and solutions object.

```
disp(output)

          funcCount: 2245
      localSolverTotal: 8
    localSolverSuccess: 8
 localSolverIncomplete: 0
 localSolverNoSolution: 0
          message: 'GlobalSearch stopped because it analyzed all the trial points....'
```

- GlobalSearch evaluated the objective function 2261 times.
- GlobalSearch ran fmincon starting from eight different points.
- All of the fmincon runs converged successfully to a local solution.

```
disp(solutions)
```

```

1x4 GlobalOptimSolution array with properties:
    X
    Fval
    Exitflag
    Output
    X0

arrayfun(@(x)x.Output.funcCount,solutions)

ans = 1x4

    31    34    40     3

```

The eight local solver runs found four solutions. The `funcCount` output shows that `fmincon` took no more than 40 function evaluations to reach each of the four solutions. The output does not show how many function evaluations four of the `fmincon` runs took. Most of the 2261 function evaluations seem to be for `GlobalSearch` to evaluate trial points, not for `fmincon` to run starting from those points.

Input Arguments

gs — GlobalSearch solver

GlobalSearch object

GlobalSearch solver, specified as a GlobalSearch object. Create `gs` using the GlobalSearch command.

ms — MultiStart solver

MultiStart object

MultiStart solver, specified as a MultiStart object. Create `ms` using the MultiStart command.

problem — Optimization problem

problem structure

Optimization problem, specified as a problem structure. Create `problem` using `createOptimProblem`. For further details, see “Create Problem Structure” on page 4-4.

Example: `problem = createOptimProblem('fmincon','objective',fun,'x0',x0,'lb',lb)`

Data Types: struct

k — Number of start points

positive integer

Number of start points, specified as a positive integer. MultiStart generates `k - 1` start points using the same algorithm as for a RandomStartPointSet object. MultiStart also uses the `x0` point from the problem structure.

Example: 50

Data Types: double

startpts — Start points for MultiStart

CustomStartPointSet object | RandomStartPointSet object | cell array of such objects

Start points for `MultiStart`, specified as a `CustomStartPointSet` object, as a `RandomStartPointSet` object, or as a cell array of such objects.

Example: `{custompts, randompts}`

Output Arguments

x — Best point found

real array

Best point found, returned as a real array. The best point is the one with lowest objective function value.

fval — Lowest objective function value encountered

real scalar

Lowest objective function value encountered, returned as a real scalar. For `lsqcurvefit` and `lsqnonlin`, the objective function is the sum of squares, also known as the squared norm of the residual.

exitflag — Exit condition summary

integer

Exit condition summary, returned as an integer.

Global Solver Exit Flags

2	At least one feasible local minimum found. Some runs of the local solver did not converge.
1	At least one feasible local minimum found. All runs of the local solver converged (had positive exit flag).
0	No local minimum found. Local solver called at least once, and at least one local solver exceeded the <code>MaxIterations</code> or <code>MaxFunctionEvaluations</code> tolerances.
-1	One or more local solver runs stopped by the local solver output or plot function.
-2	No feasible local minimum found.
-5	<code>MaxTime</code> limit exceeded.
-8	No solution found. All runs had local solver exit flag -2 or lower, not all equal -2.
-10	Failures encountered in user-provided functions.

output — Solution process details

structure

Solution process details, returned as a structure with the following fields.

Field	Meaning
<code>funcCount</code>	Number of function evaluations.
<code>localSolverIncomplete</code>	Number of local solver runs with 0 exit flag.

Field	Meaning
localSolverNoSolution	Number of local solver runs with negative exit flag.
localSolverSuccess	Number of local solver runs with positive exit flag.
localSolverTotal	Total number of local solver runs.
message	Exit message.

solutions – Distinct local solutions

vector of `GlobalOptimSolution` objects

Distinct local solutions, returned as a vector of `GlobalOptimSolution` objects.

Version History

Introduced in R2010a

See Also

`MultiStart` | `GlobalSearch` | `GlobalOptimSolution`

Topics

“Run the Solver” on page 4-13

“Workflow for `GlobalSearch` and `MultiStart`” on page 4-3

saoptimget

(Not recommended) Values of simulated annealing options structure

Note `saoptimget` is not recommended. Instead, query options using dot notation. For more information, see “Compatibility Considerations”.

Syntax

```
val = saoptimget(options, 'name')
val = saoptimget(options, 'name', default)
```

Description

`val = saoptimget(options, 'name')` returns the value of the parameter `name` from the simulated annealing options structure `options`. `saoptimget(options, 'name')` returns an empty matrix `[]` if the value of `name` is not specified in `options`. It is only necessary to type enough leading characters of `name` to uniquely identify the parameter. `saoptimget` ignores case in parameter names.

`val = saoptimget(options, 'name', default)` returns the `'name'` parameter, but returns the default value if the `'name'` parameter is not specified (or is `[]`) in `options`.

Examples

```
opts = saoptimset('TolFun',1e-4);
val = saoptimget(opts,'TolFun');
```

returns `val = 1e-4` for `TolFun`.

Version History

Introduced in R2007a

R2018b: saoptimget is not recommended

Not recommended starting in R2018b

To query options, the `gaoptimget`, `psoptimget`, and `saoptimget` functions are not recommended. Instead, use dot notation. For example, to see the setting of the `Display` option in `opts`,

```
displayopt = opts.Display
% instead of
displayopt = gaoptimget(opts,'Display')
```

Using automatic code completions, dot notation takes fewer keystrokes: `displayopt = opts.D`
Tab.

There are no plans to remove `gaoptimget`, `psoptimget`, and `saoptimget` at this time.

See Also

simulannealbnd

Topics

“Simulated Annealing Options” on page 17-58

saoptimset

(Not recommended) Create simulated annealing options structure

Note `saoptimset` is not recommended. Use `optimoptions` instead. For more information, see “Compatibility Considerations”.

Syntax

```
saoptimset
options = saoptimset
options = saoptimset('param1',value1,'param2',value2,...)
options = saoptimset(oldopts,'param1',value1,...)
options = saoptimset(oldopts,newopts)
options = saoptimset('simulannealbnd')
```

Description

`saoptimset` with no input or output arguments displays a complete list of parameters with their valid values.

`options = saoptimset` (with no input arguments) creates a structure called `options` that contains the options, or *parameters*, for the simulated annealing algorithm, with all parameters set to `[]`.

`options = saoptimset('param1',value1,'param2',value2,...)` creates a structure `options` and sets the value of 'param1' to `value1`, 'param2' to `value2`, and so on. Any unspecified parameters are set to `[]`. It is sufficient to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names. Note that for character values, correct case and the complete value are required.

`options = saoptimset(oldopts,'param1',value1,...)` creates a copy of `oldopts`, modifying the specified parameters with the specified values.

`options = saoptimset(oldopts,newopts)` combines an existing options structure, `oldopts`, with a new options structure, `newopts`. Any parameters in `newopts` with nonempty values overwrite the corresponding old parameters in `oldopts`.

`options = saoptimset('simulannealbnd')` creates an options structure with all the parameter names and default values relevant to 'simulannealbnd'. For example,

```
saoptimset('simulannealbnd')

ans =
    AnnealingFcn: @annealingfast
    TemperatureFcn: @temperatureexp
    AcceptanceFcn: @acceptancesa
                TolFun: 1.0000e-006
    StallIterLimit: '500*numberofvariables'
    MaxFunctionEvaluations: '3000*numberofvariables'
    TimeLimit: Inf
```



```

MaxIterations: Inf
ObjectiveLimit: -Inf
    Display: 'final'
DisplayInterval: 10
    HybridFcn: []
HybridInterval: 'end'
    PlotFcns: []
PlotInterval: 1
    OutputFcns: []
InitialTemperature: 100
ReannealInterval: 100
    DataType: 'double'

```

Options

The following table lists the options you can set with `saoptimset`. See “Simulated Annealing Options” on page 17-58 for a complete description of these options and their values. Values in {} denote the default value. You can also view the options parameters by typing `saoptimset` at the command line.

`optimoptions` hides the options listed in *italics*, but `saoptimset` does not. See “Options that `optimoptions` Hides” on page 17-65.

Option	Description	Values
AcceptanceFcn	Function the algorithm uses to determine if a new point is accepted. Specify as 'acceptancesa' or a function handle.	Function handle {'acceptancesa'}
AnnealingFcn	Function the algorithm uses to generate new points. Specify as a name of a built-in annealing function or a function handle.	Function handle function name 'annealingboltz' {'annealingfast'}
DataType	Type of decision variable	'custom' {'double'}
Display	Level of display	'off' 'iter' 'diagnose' {'final'}
<i>DisplayInterval</i>	Interval for iterative display	Positive integer {10}
FunctionTolerance	Termination tolerance on function value For an options structure, use TolFun.	Positive scalar {1e-6}
HybridFcn	Automatically run HybridFcn (another optimization function) during or at the end of iterations of the solver. Specify as a name or a function handle. See “When to Use a Hybrid Function” on page 8-116.	'fminsearch' 'patternsearch' 'fminunc' 'fmincon' {} or 1-by-2 cell array {@solver, hybridoptions}, where solver = fminsearch, patternsearch, fminunc, or fmincon {}

Option	Description	Values
<i>HybridInterval</i>	Interval (if not 'end' or 'never') at which HybridFcn is called	Positive integer 'never' {'end'}
InitialTemperature	Initial value of temperature	Positive scalar positive vector {100}
MaxFunctionEvaluations	Maximum number of objective function evaluations allowed For an options structure, use MaxFunEvals.	Positive integer {3000*numberOfVariables}
MaxIterations	Maximum number of iterations allowed For an options structure, use MaxIter.	Positive integer {Inf}
MaxStallIterations	Number of iterations over which average change in fitness function value at current point is less than options.FunctionTolerance For an options structure, use StallIterLimit.	Positive integer {500*numberOfVariables}
MaxTime	The algorithm stops after running for MaxTime seconds For an options structure, use TimeLimit.	Positive scalar {Inf}
ObjectiveLimit	Minimum objective function value desired	Scalar {-Inf}
OutputFcn	Function(s) get(s) iterative data and can change options at run time For an options structure, use OutputFcns.	Function handle cell array of function handles {}
PlotFcn	Plot function(s) called during iterations For an options structure, use PlotFcns.	Function handle built-in plot function name cell array of function handles cell array of built-in plot function names 'splotbestf' 'splotbestx' 'splotf' 'splotstopping' 'splottemperature' {}
<i>PlotInterval</i>	Plot functions are called at every interval	Positive integer {1}
ReannealInterval	Reannealing interval	Positive integer {100}
TemperatureFcn	Function used to update temperature schedule	Function handle built-in temperature function name 'temperatureboltz' 'temperaturefast' {'temperatureexp'}

Version History

Introduced in R2007a

R2018b: saoptimset is not recommended

Not recommended starting in R2018b

To set options, the `gaoptimset`, `psoptimset`, and `saoptimset` functions are not recommended. Instead, use `optimoptions`.

The main difference between using `optimoptions` and the other functions is that you include the solver name as the first argument in `optimoptions`. For example, to set an iterative display in `ga`:

```
options = optimoptions('ga','Display','iter');  
% instead of  
options = gaoptimset('Display','iter');
```

The other difference is that some option names have changed. You can continue to use the old names in `optimoptions`. For details, see “Options Changes in R2016a” on page 17-65.

`optimoptions` offers these advantages over the other functions:

- `optimoptions` provides better automatic code suggestions and completions, especially in the Live Editor.
- You can use a single option-setting function instead of a variety of functions.

There are no plans to remove `gaoptimset`, `psoptimset`, and `saoptimset` at this time.

See Also

`optimoptions` | `simulannealbnd`

Topics

“Simulated Annealing Options” on page 17-58

simulannealbnd

Find minimum of function using simulated annealing algorithm

Syntax

```
x = simulannealbnd(fun,x0)
x = simulannealbnd(fun,x0,lb,ub)
x = simulannealbnd(fun,x0,lb,ub,options)
x = simulannealbnd(problem)
[x,fval] = simulannealbnd(____)
[x,fval,exitflag,output] = simulannealbnd(____)
```

Description

`x = simulannealbnd(fun,x0)` finds a local minimum, `x`, to the function handle `fun` that computes the values of the objective function. `x0` is an initial point for the simulated annealing algorithm, a real vector.

Note “Passing Extra Parameters” explains how to pass extra parameters to the objective function, if necessary.

`x = simulannealbnd(fun,x0,lb,ub)` defines a set of lower and upper bounds on the design variables in `x`, so that the solution is always in the range $lb \leq x \leq ub$. If `x(i)` is unbounded below, set `lb(i) = -Inf`, and if `x(i)` is unbounded above, set `ub(i) = Inf`.

`x = simulannealbnd(fun,x0,lb,ub,options)` minimizes with the optimization options specified in `options`. Create options using `optimoptions`. If no bounds exist, set `lb = []` and/or `ub = []`.

`x = simulannealbnd(problem)` finds the minimum for `problem`, a structure described in `problem`.

`[x,fval] = simulannealbnd(____)`, for any syntax, returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag,output] = simulannealbnd(____)` additionally returns a value `exitflag` that describes the exit condition of `simulannealbnd`, and a structure `output` with information about the optimization process.

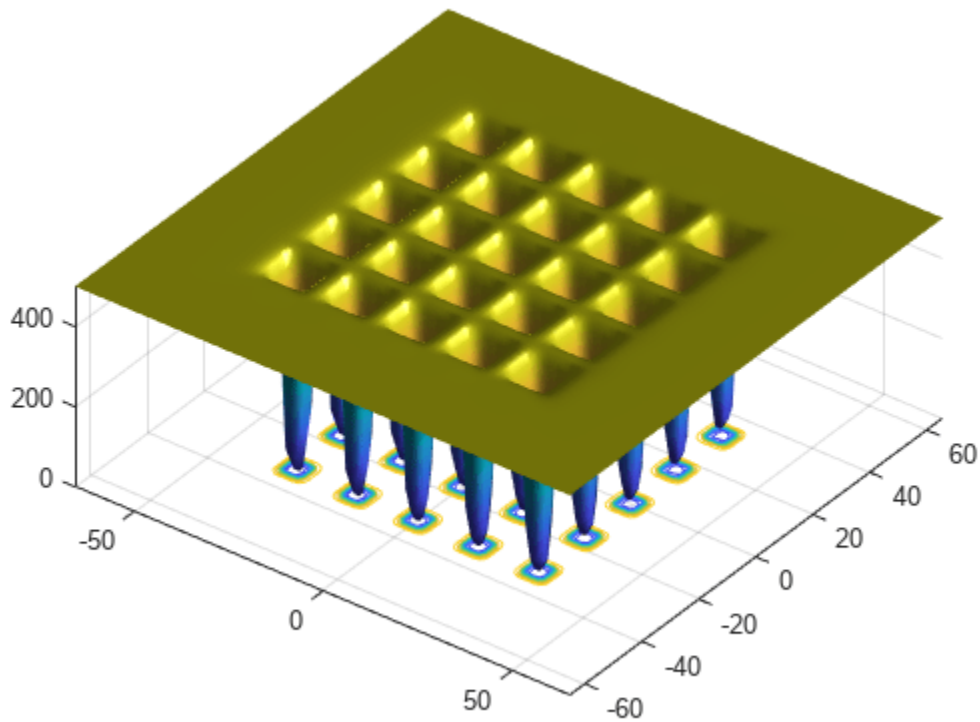
Examples

Minimize a Function with Many Local Minima

Minimize De Jong's fifth function, a two-dimensional function with many local minima. This function is available when you run this example.

Plot De Jong's fifth function.

dejong5fcn



Minimize De Jong's fifth function using `simulannealbnd` starting from the point $[0, 0]$.

```
fun = @dejong5fcn;
x0 = [0 0];
x = simulannealbnd(fun,x0)
```

Optimization terminated: change in best function value less than options.FunctionTolerance.

```
x = 1×2
```

```
   -32.0285   -0.1280
```

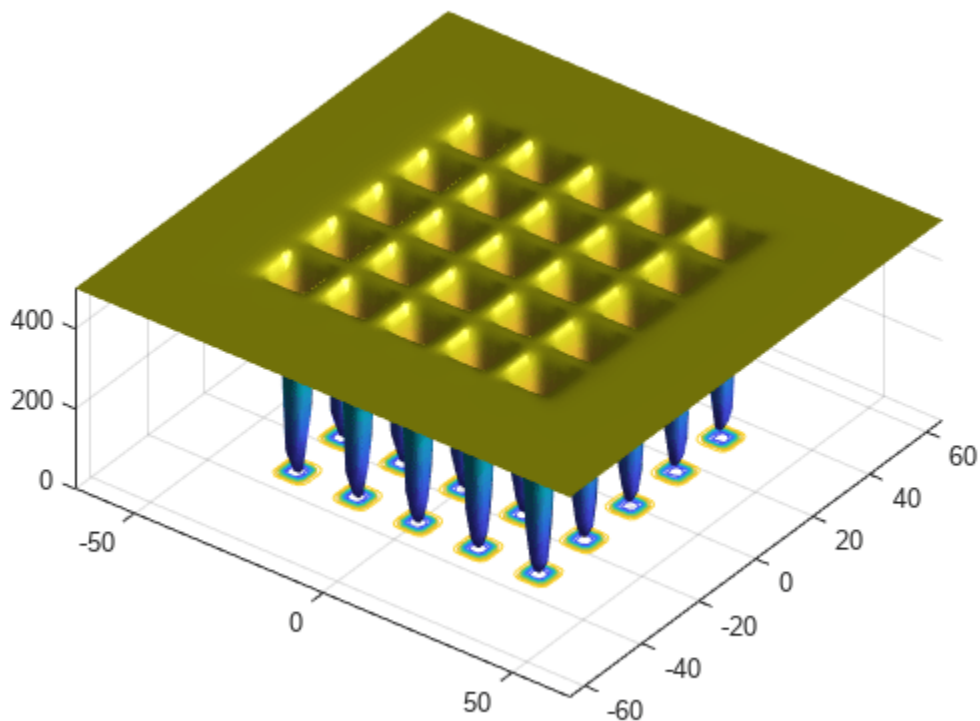
The `simulannealbnd` algorithm uses the MATLAB® random number stream, so you might obtain a different result.

Minimize Subject to Bounds

Minimize De Jong's fifth function within a bounded region. This function is available when you run this example.

Plot De Jong's fifth function.

```
dejong5fcn
```



Start `simulannealbnd` starting at the point $[0, 0]$, and set lower bounds of -64 and upper bounds of 64 on each component.

```
fun = @dejong5fcn;
x0 = [0 0];
lb = [-64 -64];
ub = [64 64];
x = simulannealbnd(fun,x0,lb,ub)
```

Optimization terminated: change in best function value less than options.FunctionTolerance.

```
x = 1×2
    -15.9790    -31.9593
```

The `simulannealbnd` algorithm uses the MATLAB® random number stream, so you might obtain a different result.

Minimize Using Nondefault Options

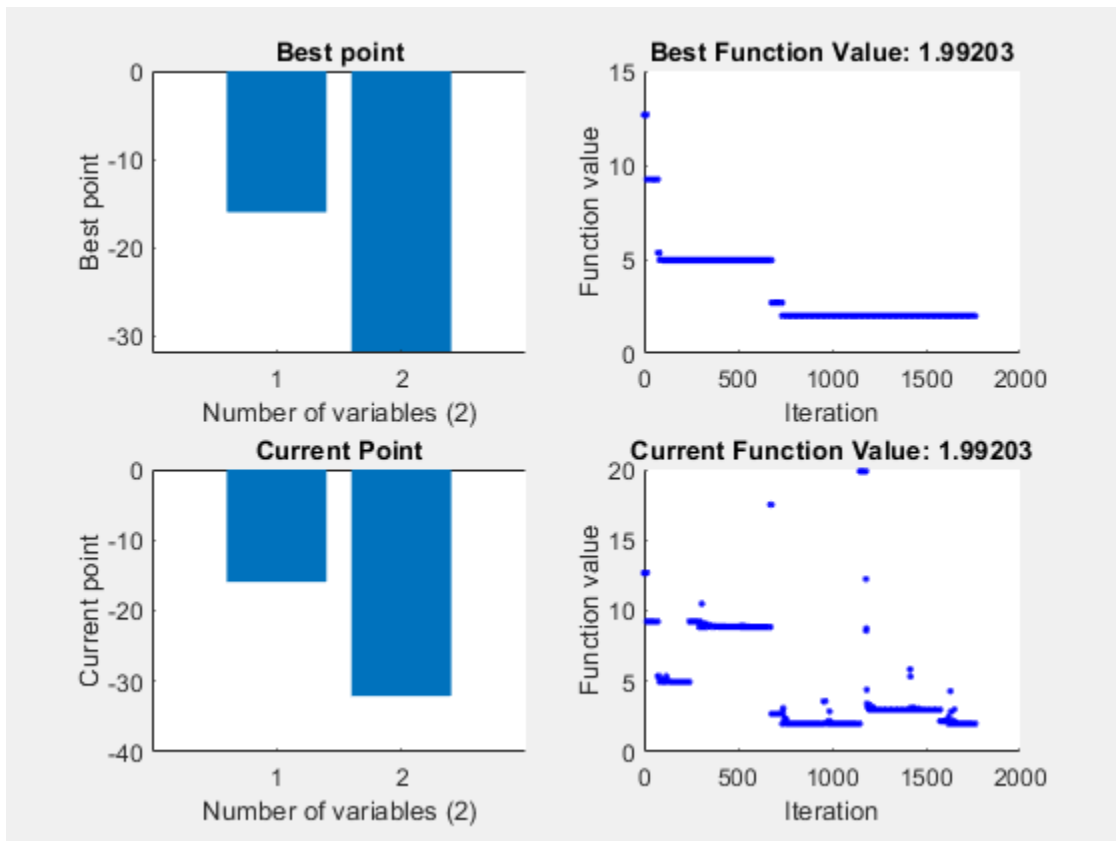
Observe the progress of `simulannealbnd` by setting options to use some plot functions.

Set simulated annealing options to use several plot functions.

```
options = optimoptions('simulannealbnd','PlotFcns',...
    {@splotbestx,@splotbestf,@splotx,@splotf});
```

Start `simulannealbnd` starting at the point $[0, 0]$, and set lower bounds of -64 and upper bounds of 64 on each component. Minimize the `dejong5fcn`, which is available when you run this example.

```
rng default % For reproducibility
fun = @dejong5fcn;
x0 = [0,0];
lb = [-64,-64];
ub = [64,64];
x = simulannealbnd(fun,x0,lb,ub,options)
```



Optimization terminated: change in best function value less than options.FunctionTolerance.

```
x = 1x2
```

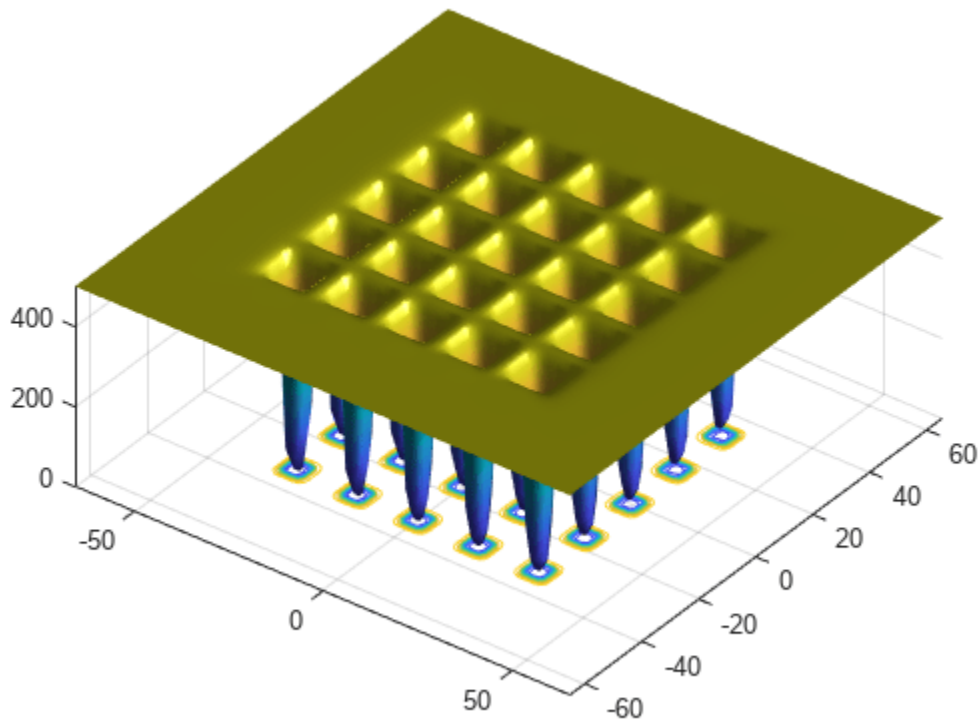
```
    -15.9790    -31.9593
```

Obtain All Outputs

Obtain all the outputs of a simulated annealing minimization.

Plot De Jong's fifth function, which is available when you run this example.

dejong5fcn



Start `simulannealbnd` starting at the point $[0, 0]$, and set lower bounds of -64 and upper bounds of 64 on each component.

```
fun = @dejong5fcn;
x0 = [0,0];
lb = [-64,-64];
ub = [64,64];
[x,fval,exitflag,output] = simulannealbnd(fun,x0,lb,ub)
```

Optimization terminated: change in best function value less than options.FunctionTolerance.

```
x = 1x2
```

```
    -15.9790    -31.9593
```

```
fval = 1.9920
```

```
exitflag = 1
```

```
output = struct with fields:
```

```
    iterations: 1762
```

```
    funccount: 1779
```

```
    message: 'Optimization terminated: change in best function value less than options.Funct
```

```
    rngstate: [1x1 struct]
```

```
    problemtype: 'boundconstraints'
```



```
temperature: [2x1 double]
totaltime: 0.5518
```

The `simulannealbnd` algorithm uses the MATLAB® random number stream, so you might obtain a different result.

Input Arguments

fun — Function to be minimized

function handle | function name

Function to be minimized, specified as a function handle or function name. `fun` is a function that accepts a vector `x` and returns a real scalar `f`, the objective function evaluated at `x`.

`fun` can be specified as a function handle for a file:

```
x = simulannealbnd(@myfun,x0)
```

where `myfun` is a MATLAB function such as

```
function f = myfun(x)
f = ...           % Compute function value at x
```

`fun` can also be a function handle for an anonymous function:

```
x = simulannealbnd(@(x)norm(x)^2,x0,lb,ub);
```

Example: `fun = @(x)sin(x(1))*cos(x(2))`

Data Types: char | function_handle | string

x0 — Initial point

real vector

Initial point, specified as a real vector. `simulannealbnd` uses the number of elements in `x0` to determine the number of variables that `fun` accepts.

Example: `x0 = [1,2,3,4]`

Data Types: double

lb — Lower bounds

real vector | real array

Lower bounds, specified as a real vector or real array. If the number of elements in `x0` is equal to that of `lb`, then `lb` specifies that

$x(i) \geq lb(i)$ for all i .

If `numel(lb) < numel(x0)`, then `lb` specifies that

$x(i) \geq lb(i)$ for $1 \leq i \leq \text{numel}(lb)$.

In this case, solvers issue a warning.

Example: To specify that all control variables are positive, `lb = zeros(size(x0))`

Data Types: `double`

ub — Upper bounds

`real vector` | `real array`

Upper bounds, specified as a real vector or real array. If the number of elements in `x0` is equal to that of `ub`, then `ub` specifies that

$x(i) \leq ub(i)$ for all i .

If `numel(ub) < numel(x0)`, then `ub` specifies that

$x(i) \leq ub(i)$ for $1 \leq i \leq \text{numel}(ub)$.

In this case, solvers issue a warning.

Example: To specify that all control variables are less than one, `ub = ones(size(x0))`

Data Types: `double`

options — Optimization options

object returned by `optimoptions` | structure

Optimization options, specified as an object returned by `optimoptions` or a structure. For details, see “Simulated Annealing Options” on page 17-58.

`optimoptions` hides the options listed in *italics*; see “Options that `optimoptions` Hides” on page 17-65.

`{}` denotes the default value. See option details in “Simulated Annealing Options” on page 17-58.

Option	Description	Values
AcceptanceFcn	Function the algorithm uses to determine if a new point is accepted. Specify as 'acceptancesa' or a function handle.	Function handle {'acceptancesa'}
AnnealingFcn	Function the algorithm uses to generate new points. Specify as a name of a built-in annealing function or a function handle.	Function handle function name 'annealingboltz' {'annealingfast'}
DataType	Type of decision variable	'custom' {'double'}
Display	Level of display	'off' 'iter' 'diagnose' {'final'}
<i>DisplayInterval</i>	Interval for iterative display	Positive integer {10}
FunctionTolerance	Termination tolerance on function value For an options structure, use TolFun.	Positive scalar {1e-6}

Option	Description	Values
HybridFcn	Automatically run HybridFcn (another optimization function) during or at the end of iterations of the solver. Specify as a name or a function handle. See “When to Use a Hybrid Function” on page 8-116.	'fminsearch' 'patternsearch' 'fminunc' 'fmincon' {} or 1-by-2 cell array {@solver, hybridoptions}, where solver = fminsearch, patternsearch, fminunc, or fmincon {}
HybridInterval	Interval (if not 'end' or 'never') at which HybridFcn is called	Positive integer 'never' {'end'}
InitialTemperature	Initial value of temperature	Positive scalar positive vector {100}
MaxFunctionEvaluations	Maximum number of objective function evaluations allowed For an options structure, use MaxFunEvals.	Positive integer {3000*numberOfVariables}
MaxIterations	Maximum number of iterations allowed For an options structure, use MaxIter.	Positive integer {Inf}
MaxStallIterations	Number of iterations over which average change in fitness function value at current point is less than options.FunctionTolerance For an options structure, use StallIterLimit.	Positive integer {500*numberOfVariables}
MaxTime	The algorithm stops after running for MaxTime seconds For an options structure, use TimeLimit.	Positive scalar {Inf}
ObjectiveLimit	Minimum objective function value desired	Scalar {-Inf}
OutputFcn	Function(s) get(s) iterative data and can change options at run time For an options structure, use OutputFcns.	Function handle cell array of function handles {}
PlotFcn	Plot function(s) called during iterations For an options structure, use PlotFcns.	Function handle built-in plot function name cell array of function handles cell array of built-in plot function names 'splotbestf' 'splotbestx' 'splotf' 'splotstopping' 'splottemperature' {}

Option	Description	Values
<i>PlotInterval</i>	Plot functions are called at every interval	Positive integer {1}
<i>ReannealInterval</i>	Reannealing interval	Positive integer {100}
<i>TemperatureFcn</i>	Function used to update temperature schedule	Function handle built-in temperature function name 'temperatureboltz' 'temperaturefast' {'temperatureexp'}

Example: `options = optimoptions(@simulannealbnd,'MaxIterations',150)`

Data Types: `struct`

problem — Problem structure

structure

Problem structure, specified as a structure with the following fields:

- `objective` — Objective function
- `x0` — Starting point
- `lb` — Lower bound for `x`
- `ub` — Upper bound for `x`
- `solver` — 'simulannealbnd'
- `options` — Options created with `optimoptions` or an options structure
- `rngstate` — Optional field to reset the state of the random number generator

Note problem must have all the fields as specified above.

Data Types: `struct`

Output Arguments

`x` — Solution

real vector

Solution, returned as a real vector. The size of `x` is the same as the size of `x0`. Typically, `x` is a local solution to the problem when `exitflag` is positive.

`fval` — Objective function value at the solution

real number

Objective function value at the solution, returned as a real number. Generally, `fval = fun(x)`.

`exitflag` — Reason `simulannealbnd` stopped

integer

Reason `simulannealbnd` stopped, returned as an integer.

Exit Flag	Meaning
1	Average change in the value of the objective function over <code>options.MaxStallIterations</code> iterations is less than <code>options.FunctionTolerance</code> .
5	Objective function value is less than <code>options.ObjectiveLimit</code> .
0	Maximum number of function evaluations or iterations reached.
-1	Optimization terminated by an output function or plot function.
-2	No feasible point found.
-5	Time limit exceeded.

output — Information about the optimization process

structure

Information about the optimization process, returned as a structure with fields:

- `problemtype` — Type of problem: unconstrained or bound constrained.
- `iterations` — The number of iterations computed.
- `funccount` — The number of evaluations of the objective function.
- `message` — The reason the algorithm terminated.
- `temperature` — Temperature when the solver terminated.
- `totaltime` — Total time for the solver to run.
- `rngstate` — State of the MATLAB random number generator, just before the algorithm started. You can use the values in `rngstate` to reproduce the output of `simulannealbnd`. See “Reproduce Your Results” on page 13-17.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for `simulannealbnd`.

Version History

Introduced in R2007a

See Also

`ga` | `optimoptions` | `patternsearch` | **Optimize**

Topics

“Minimization Using Simulated Annealing Algorithm” on page 13-19

“Simulated Annealing Options” on page 13-22

“Multiprocessor Scheduling Using Simulated Annealing with a Custom Data Type” on page 13-28

“Optimization Workflow” on page 1-28

“What Is Simulated Annealing?” on page 13-2

“Simulated Annealing Terminology” on page 13-12

“How Simulated Annealing Works” on page 13-14

surrogateopt

Surrogate optimization for global minimization of time-consuming objective functions

Syntax

```
x = surrogateopt(objconstr, lb, ub)
x = surrogateopt(objconstr, lb, ub, intcon)
x = surrogateopt(objconstr, lb, ub, intcon, A, b, Aeq, beq)
x = surrogateopt( ____, options)
x = surrogateopt(problem)

x = surrogateopt(checkpointFile)
x = surrogateopt(checkpointFile, opts)

[x, fval] = surrogateopt( ____)
[x, fval, exitflag, output] = surrogateopt( ____)
[x, fval, exitflag, output, trials] = surrogateopt( ____)
```

Description

`surrogateopt` is a global solver for time-consuming objective functions.

`surrogateopt` attempts to solve problems of the form

$$\min_x f(x) \text{ such that } \begin{cases} lb \leq x \leq ub \\ A \cdot x \leq b \\ Aeq \cdot x = beq \\ c(x) \leq 0 \\ x_i \text{ integer, } i \in \text{intcon.} \end{cases}$$

The solver searches for the global minimum of a real-valued objective function in multiple dimensions, subject to bounds, optional linear constraints, optional integer constraints, and optional nonlinear inequality constraints. `surrogateopt` is best suited to objective functions that take a long time to evaluate. The objective function can be nonsmooth. The solver requires finite bounds on all variables. The solver can optionally maintain a checkpoint file to enable recovery from crashes or partial execution, or optimization continuation after meeting a stopping condition. The objective function $f(x)$ can be empty (`[]`), in which case `surrogateopt` attempts to find a point satisfying all the constraints.

`x = surrogateopt(objconstr, lb, ub)` searches for a global minimum of `objconstr(x)` in the region `lb <= x <= ub`. If `objconstr(x)` returns a structure, then `surrogateopt` searches for a minimum of `objconstr(x).Fval`, subject to `objconstr(x).Ineq <= 0`.

Note “Passing Extra Parameters” explains how to pass extra parameters to the objective function, if necessary.

`x = surrogateopt(objconstr, lb, ub, intcon)` requires that the variables listed in `intcon` take integer values.

`x = surrogateopt(objconstr,lb,ub,intcon,A,b,Aeq,beq)` requires that the solution `x` satisfy the inequalities $A*x \leq b$ and the equalities $Aeq*x = beq$. If no inequalities exist, set `A = []` and `b = []`. Similarly, if no equalities exist, set `Aeq = []` and `beq = []`.

`x = surrogateopt(___, options)` modifies the search procedure using the options in `options`. Specify options following any input argument combination in the previous syntaxes.

`x = surrogateopt(problem)` searches for a minimum for `problem`, a structure described in `problem`.

`x = surrogateopt(checkpointFile)` continues running the optimization from the state in a saved checkpoint file. See “Work with Checkpoint Files” on page 11-56.

`x = surrogateopt(checkpointFile,opts)` continues running the optimization from the state in a saved checkpoint file, and replaces options in `checkpointFile` with those in `opts`. See “Checkpoint File” on page 17-56.

`[x,fval] = surrogateopt(___,)` also returns the best (smallest) value of the objective function found by the solver, using any of the input argument combinations in the previous syntaxes.

`[x,fval,exitflag,output] = surrogateopt(___,)` also returns `exitflag`, an integer describing the reason the solver stopped, and `output`, a structure describing the optimization procedure.

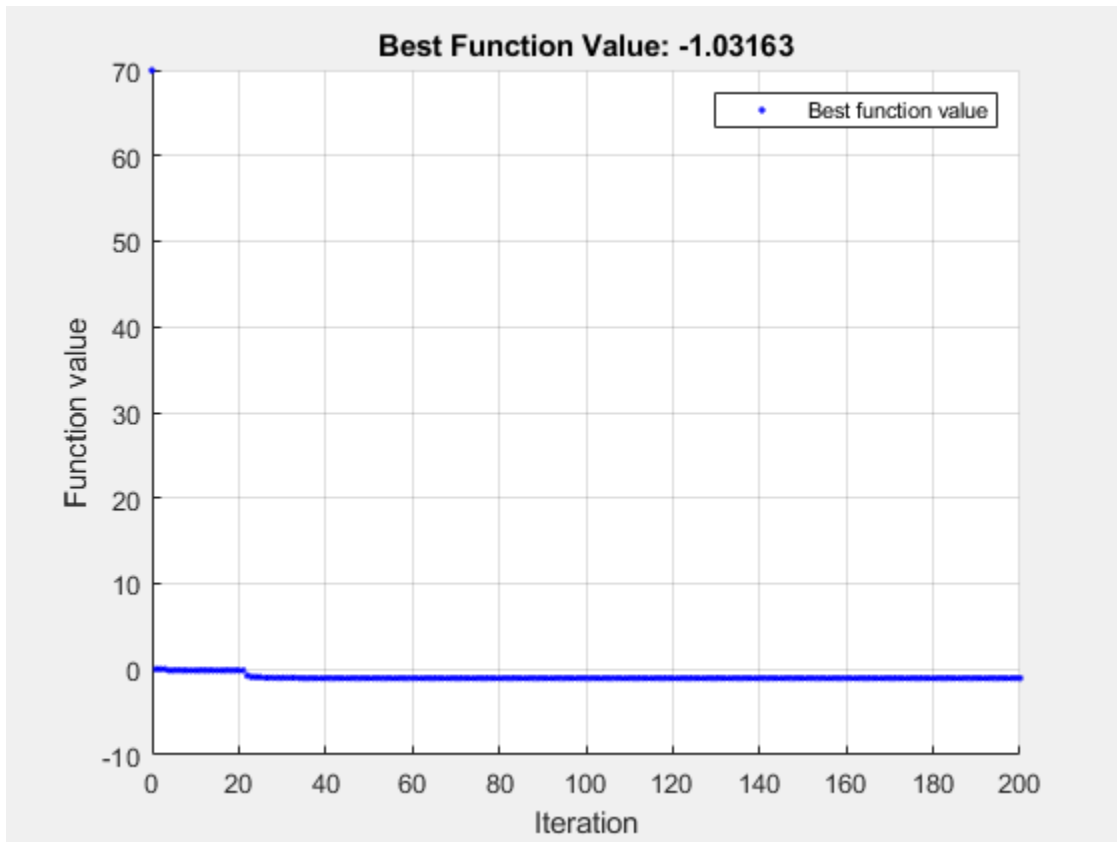
`[x,fval,exitflag,output,trials] = surrogateopt(___,)` also returns a structure containing all of the evaluated points and the objective function values at those points.

Examples

Search for Global Minimum

Search for a minimum of the six-hump camel back function in the region $-2.1 \leq x(i) \leq 2.1$. This function has two global minima with the objective function value $-1.0316284\dots$ and four local minima with higher objective function values.

```
rng default % For reproducibility
objconstr = @(x)(4*x(:,1).^2 - 2.1*x(:,1).^4 + x(:,1).^6/3 ...
    + x(:,1).*x(:,2) - 4*x(:,2).^2 + 4*x(:,2).^4);
lb = [-2.1,-2.1];
ub = -lb;
x = surrogateopt(objconstr,lb,ub)
```



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

`x = 1x2`

0.0893 -0.7130

Solve Problem with Nonlinear Constraints

Find the minimum of Rosenbrock's function

$$100(x(2) - x(1)^2)^2 + (1 - x(1))^2$$

subject to the nonlinear constraint that the solution lies in a disk of radius $1/3$ around the point $[1/3, 1/3]$:

$$(x(1) - 1/3)^2 + (x(2) - 1/3)^2 \leq (1/3)^2.$$

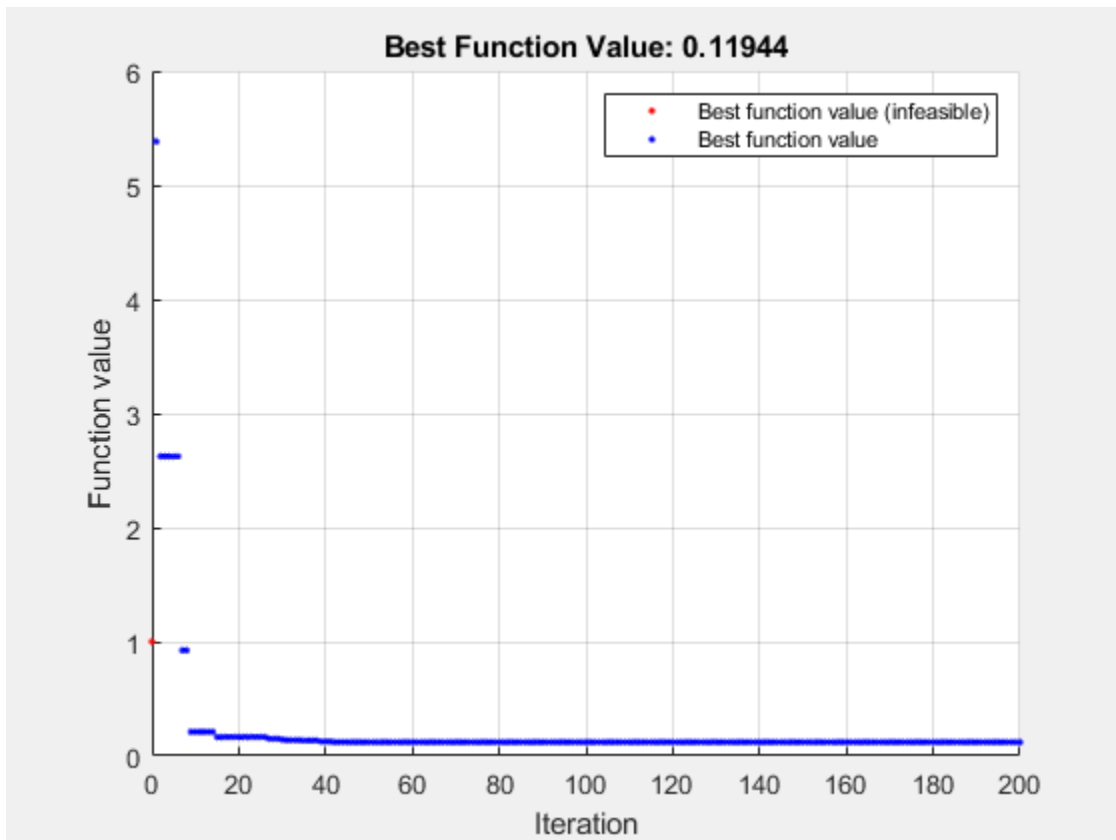
To do so, write a function `objconstr(x)` that returns the value of Rosenbrock's function in a structure field `Fval`, and returns the nonlinear constraint value in the form $c(x) \leq 0$ in the structure field `Ineq`.

type `objconstr`


```
function f = objconstr(x)
f.Fval = 100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;
f.Ineq = (x(1)-1/3)^2 + (x(2)-1/3)^2 - (1/3)^2;
```

Call `surrogateopt` using lower bounds of 0 and upper bounds of 2/3 on each component.

```
lb = [0,0];
ub = [2/3,2/3];
[x,fval,exitflag] = surrogateopt(@objconstr,lb,ub)
```



`surrogateopt` stopped because it exceeded the function evaluation limit set by `'options.MaxFunctionEvaluations'`.

```
x = 1x2
```

```
    0.6544    0.4280
```

```
fval = 0.1194
```

```
exitflag = 0
```

Check the value of the nonlinear constraint at the solution.

```
disp(objconstr(x).Ineq)
```

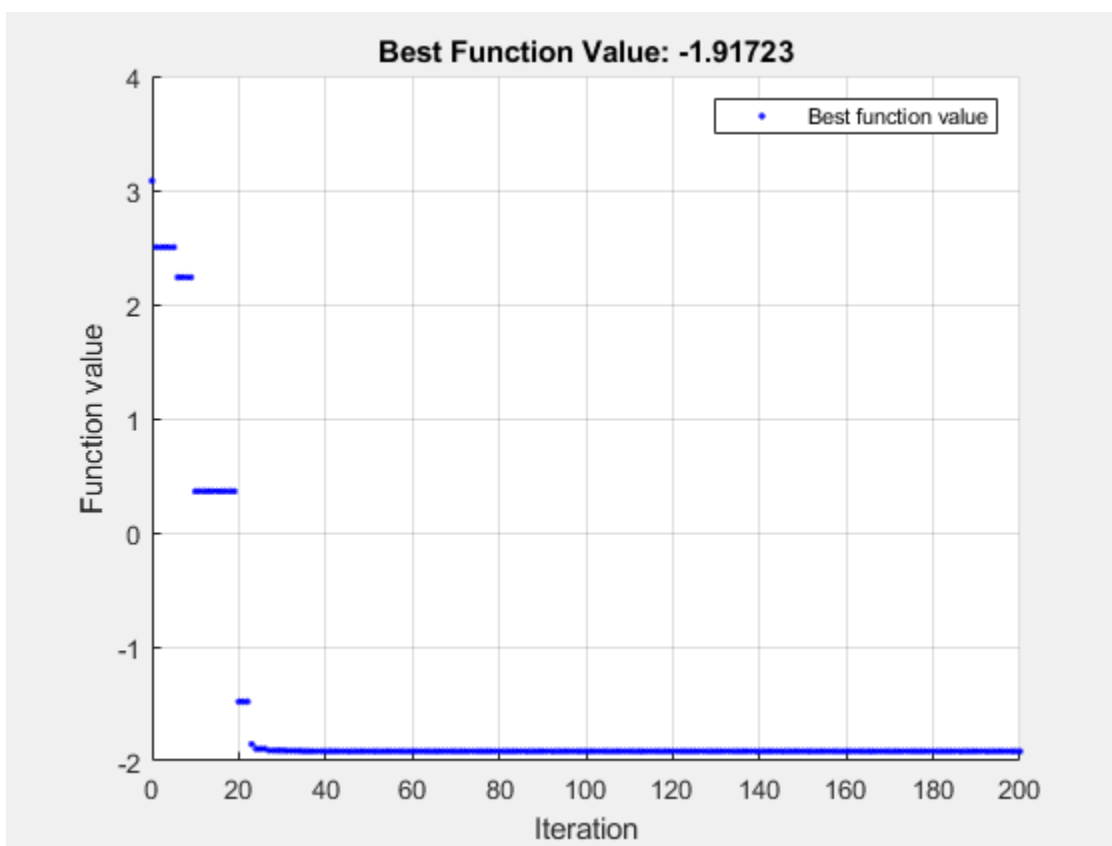
```
    9.3778e-04
```

The constraint function value is near zero, indicating that the constraint is active at the solution.

Solve Mixed-Integer Problem

Find the minimum of the `ps_example` function for a two-dimensional variable x whose first component is restricted to integer values, and all components are between -5 and 5 .

```
intcon = 1;
rng default % For reproducibility
objconstr = @ps_example;
lb = [-5,-5];
ub = [5,5];
x = surrogateopt(objconstr,lb,ub,intcon)
```



`surrogateopt` stopped because it exceeded the function evaluation limit set by `'options.MaxFunctionEvaluations'`.

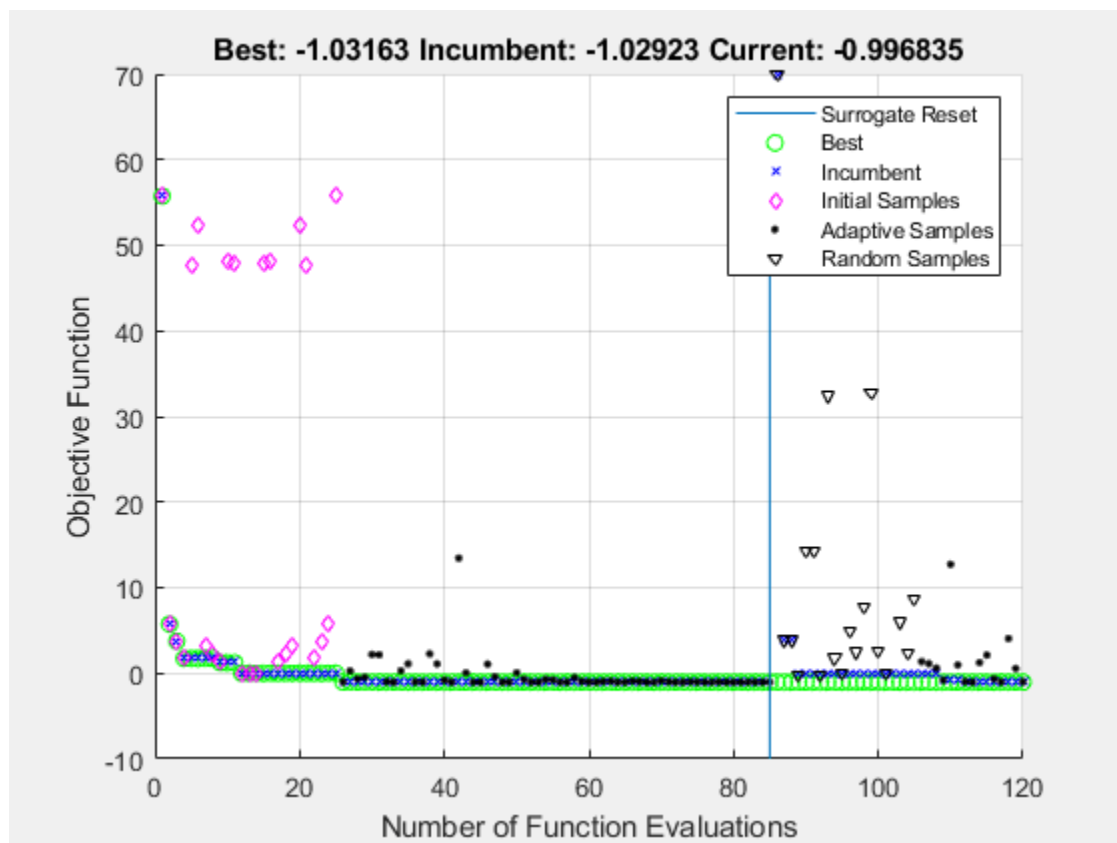
```
x = 1x2
    -5.0000    0.0006
```

Surrogate Optimization Using Nondefault Options

Minimize the six-hump camel back function in the region $-2.1 \leq x(i) \leq 2.1$. This function has two global minima with the objective function value $-1.0316284\dots$ and four local minima with higher objective function values.

To search the region systematically, use a regular grid of starting points. Set 120 as the maximum number of function evaluations. Use the 'surrogateoptplot' plot function. To understand the 'surrogateoptplot' plot, see "Interpret surrogateoptplot" on page 11-25.

```
rng default % For reproducibility
objconstr = @(x)(4*x(:,1).^2 - 2.1*x(:,1).^4 + x(:,1).^6/3 ...
    + x(:,1).*x(:,2) - 4*x(:,2).^2 + 4*x(:,2).^4);
lb = [-2.1,-2.1];
ub = -lb;
[Xpts,Ypts] = meshgrid(-3:3);
startpts = [Xpts(:),Ypts(:)];
options = optimoptions('surrogateopt','PlotFcn','surrogateoptplot',...
    'InitialPoints',startpts,'MaxFunctionEvaluations',120);
x = surrogateopt(objconstr,lb,ub,options)
```



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

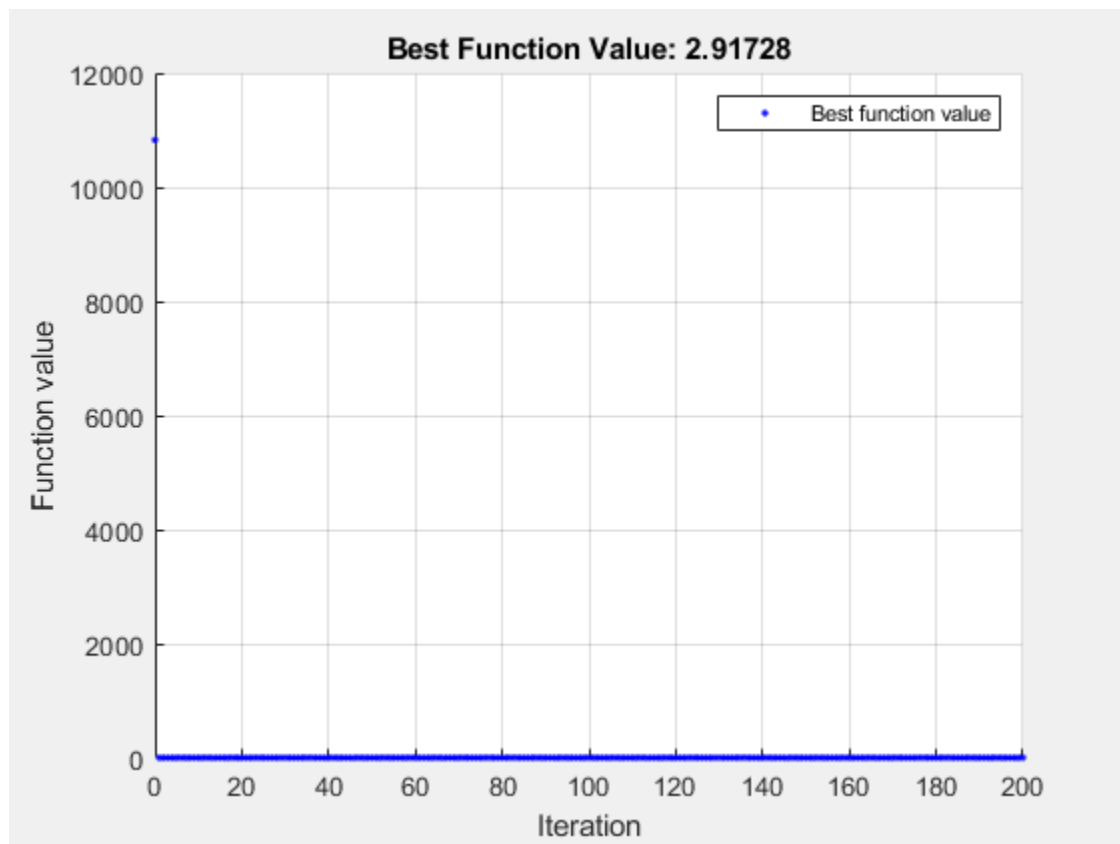
```
x = 1x2
```

```
0.0900 -0.7123
```

Linear Constraints in surrogateopt

Minimize a nonlinear objective function subject to linear inequality constraints. Minimize for 200 function evaluations.

```
objconstr = @multirosenbrock;
nvar = 6;
lb = -2*ones(nvar,1);
ub = -lb;
intcon = [];
A = ones(1,nvar);
b = 3;
Aeq = [];
beq = [];
options = optimoptions('surrogateopt','MaxFunctionEvaluations',200);
[sol,fval,exitflag,output] = ...
    surrogateopt(objconstr,lb,ub,intcon,A,b,Aeq,beq,options)
```



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
sol = 1x6
```

```
    0.0159    0.0061    0.0289    0.0046    0.0207    0.0209
```

```

fval = 2.9173
exitflag = 0
output = struct with fields:
    elapsedtime: 21.7633
    funccount: 200
    constrviolation: 0
        ineq: [1x0 double]
    rngstate: [1x1 struct]
    message: 'surrogateopt stopped because it exceeded the function evaluation limit set

```

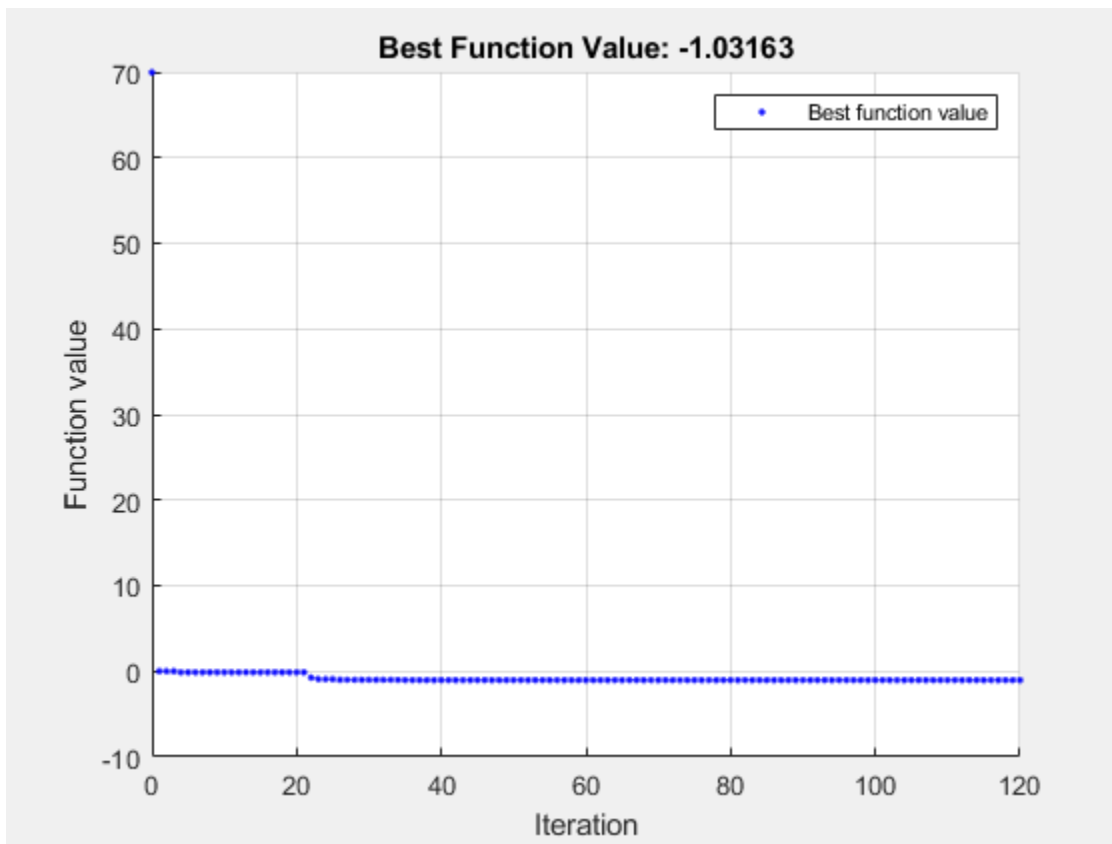
Surrogate Optimization of Problem Structure

Create a problem structure representing the six-hump camel back function in the region $-2.1 \leq x(i) \leq 2.1$. Set 120 as the maximum number of function evaluations.

```

rng default % For reproducibility
objconstr = @(x)(4*x(:,1).^2 - 2.1*x(:,1).^4 + x(:,1).^6/3 ...
    + x(:,1).*x(:,2) - 4*x(:,2).^2 + 4*x(:,2).^4);
options = optimoptions('surrogateopt','MaxFunctionEvaluations',120);
problem = struct('objective',objconstr,...
    'lb',[-2.1,-2.1],...
    'ub',[2.1,2.1],...
    'options',options,...
    'solver','surrogateopt');
x = surrogateopt(problem)

```



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
x = 1x2
    -0.0892    0.7129
```

Return Surrogate Optimization Objective Function Value

Minimize the six-hump camel back function and return both the minimizing point and the objective function value. Set options to suppress all other display.

```
rng default % For reproducibility
objconstr = @(x)(4*x(:,1).^2 - 2.1*x(:,1).^4 + x(:,1).^6/3 ...
    + x(:,1).*x(:,2) - 4*x(:,2).^2 + 4*x(:,2).^4);
lb = [-2.1,-2.1];
ub = -lb;
options = optimoptions('surrogateopt','Display','off','PlotFcn',[]);
[x,fval] = surrogateopt(objconstr,lb,ub,options)
```

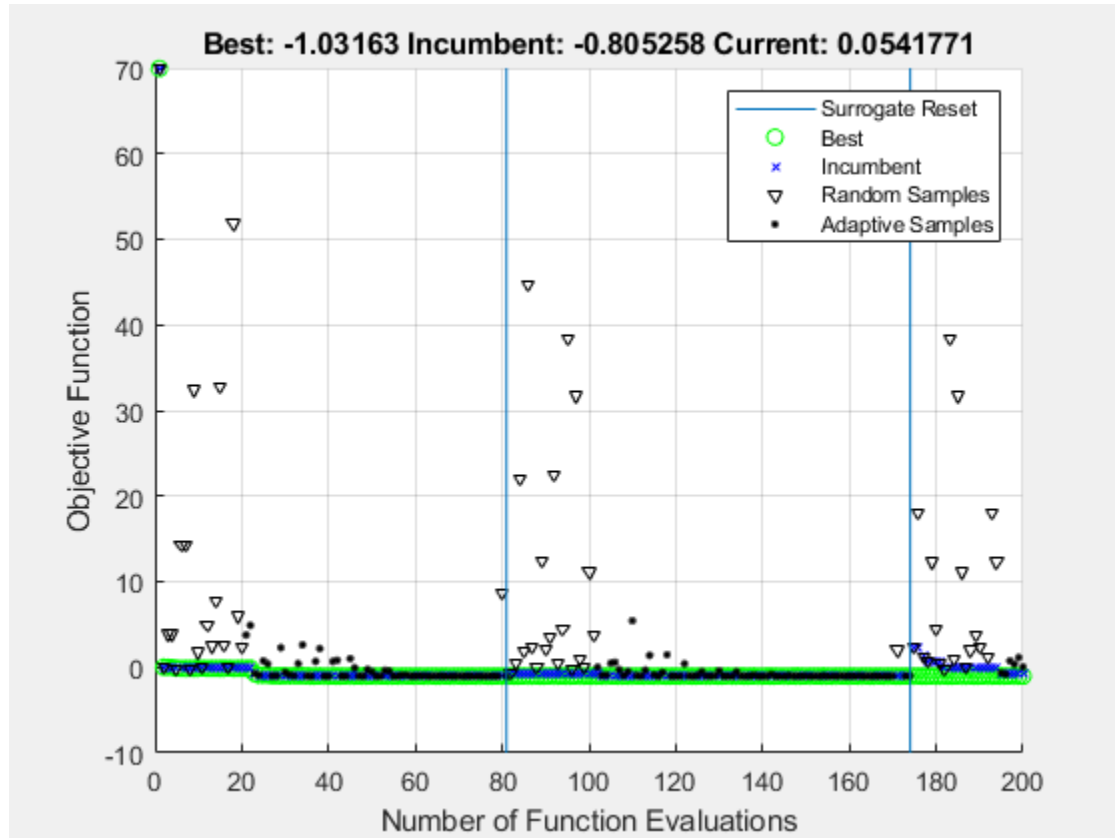
```
x = 1x2
    0.0893   -0.7130
```

```
fval = -1.0316
```

Monitor Surrogate Optimization Process

Monitor the surrogate optimization process by requesting that `surrogateopt` return more outputs. Use the `'surrogateoptplot'` plot function. To understand the `'surrogateoptplot'` plot, see “Interpret surrogateoptplot” on page 11-25.

```
rng default % For reproducibility
objconstr = @(x)(4*x(:,1).^2 - 2.1*x(:,1).^4 + x(:,1).^6/3 ...
    + x(:,1).*x(:,2) - 4*x(:,2).^2 + 4*x(:,2).^4);
lb = [-2.1,-2.1];
ub = -lb;
options = optimoptions('surrogateopt','PlotFcn','surrogateoptplot');
[x,fval,exitflag,output] = surrogateopt(objconstr,lb,ub,options)
```



`surrogateopt` stopped because it exceeded the function evaluation limit set by `'options.MaxFunctionEvaluations'`.

```
x = 1x2
```

```
    0.0893    -0.7130
```

```
fval = -1.0316
```

```

exitflag = 0
output = struct with fields:
    elapsedtime: 44.2029
    funccount: 200
    constrviolation: 0
        ineq: [1x0 double]
    rngstate: [1x1 struct]
    message: 'surrogateopt stopped because it exceeded the function evaluation limit set

```

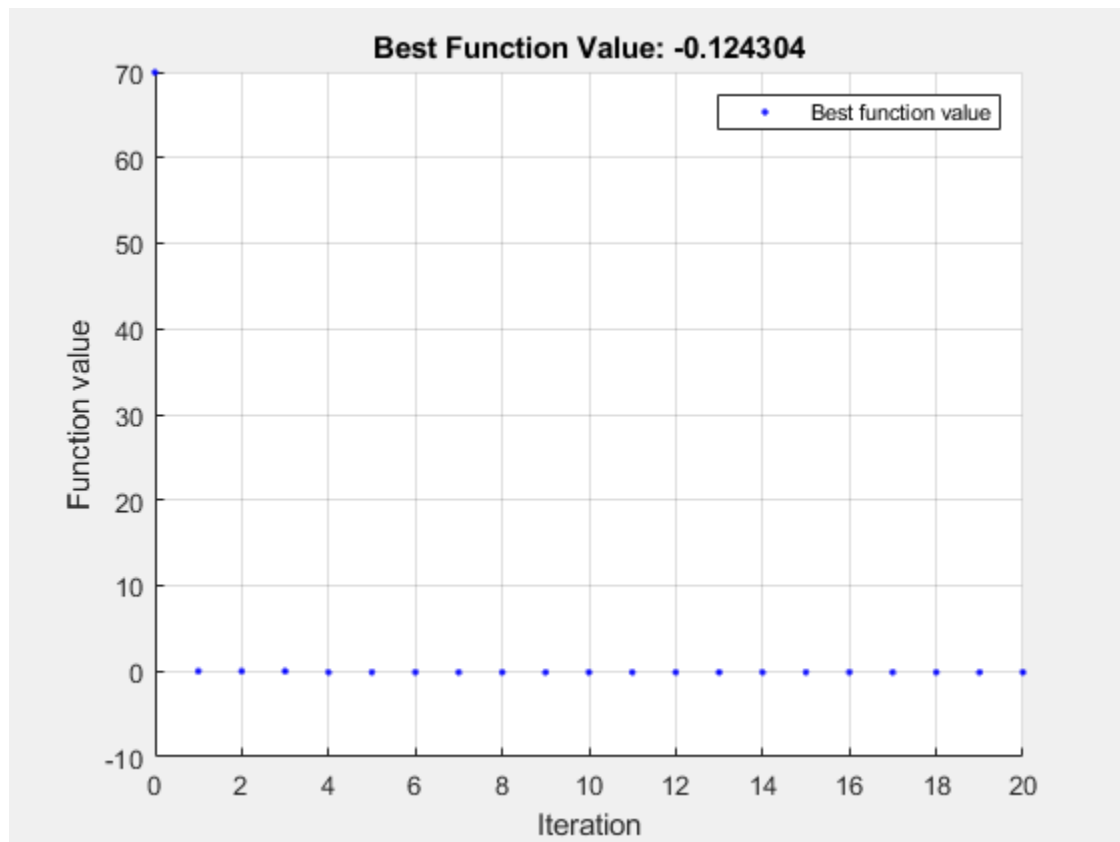
Restart Surrogate Optimization

Conclude a surrogate optimization quickly by setting a small maximum number of function evaluations. To prepare for the possibility of restarting the optimization, request all solver outputs.

```

rng default % For reproducibility
objconstr = @(x)(4*x(:,1).^2 - 2.1*x(:,1).^4 + x(:,1).^6/3 ...
    + x(:,1).*x(:,2) - 4*x(:,2).^2 + 4*x(:,2).^4);
lb = [-2.1,-2.1];
ub = -lb;
options = optimoptions('surrogateopt','MaxFunctionEvaluations',20);
[x,fval,exitflag,output,trials] = surrogateopt(objconstr,lb,ub,options);

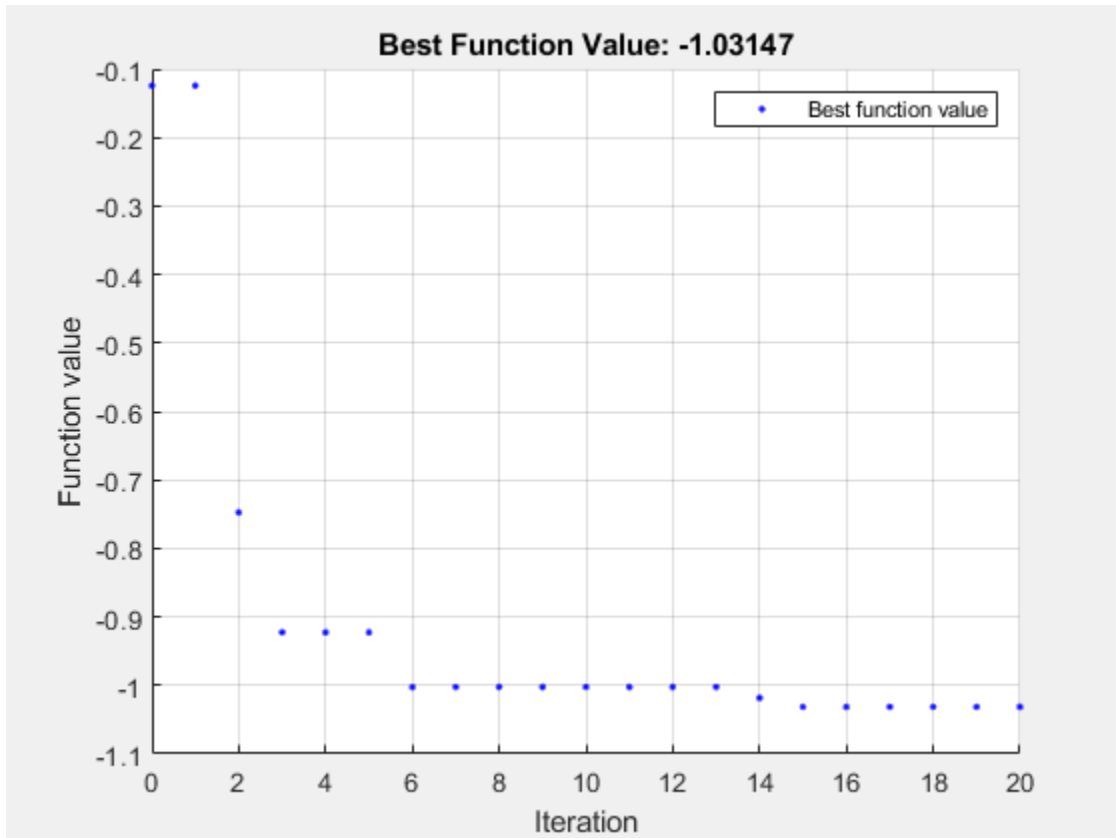
```



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

Optimize for another 20 function evaluations, starting from the previously evaluated points.

```
options.InitialPoints = trials;
[x,fval,exitflag,output,trials] = surrogateopt(objconstr,lb,ub,options);
```



surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

By comparing the plots of these 40 function evaluations to those in “Search for Global Minimum” on page 18-191, you see that restarting surrogate optimization is not the same as having the solver run continuously.

Restart Surrogate Optimization from Checkpoint File

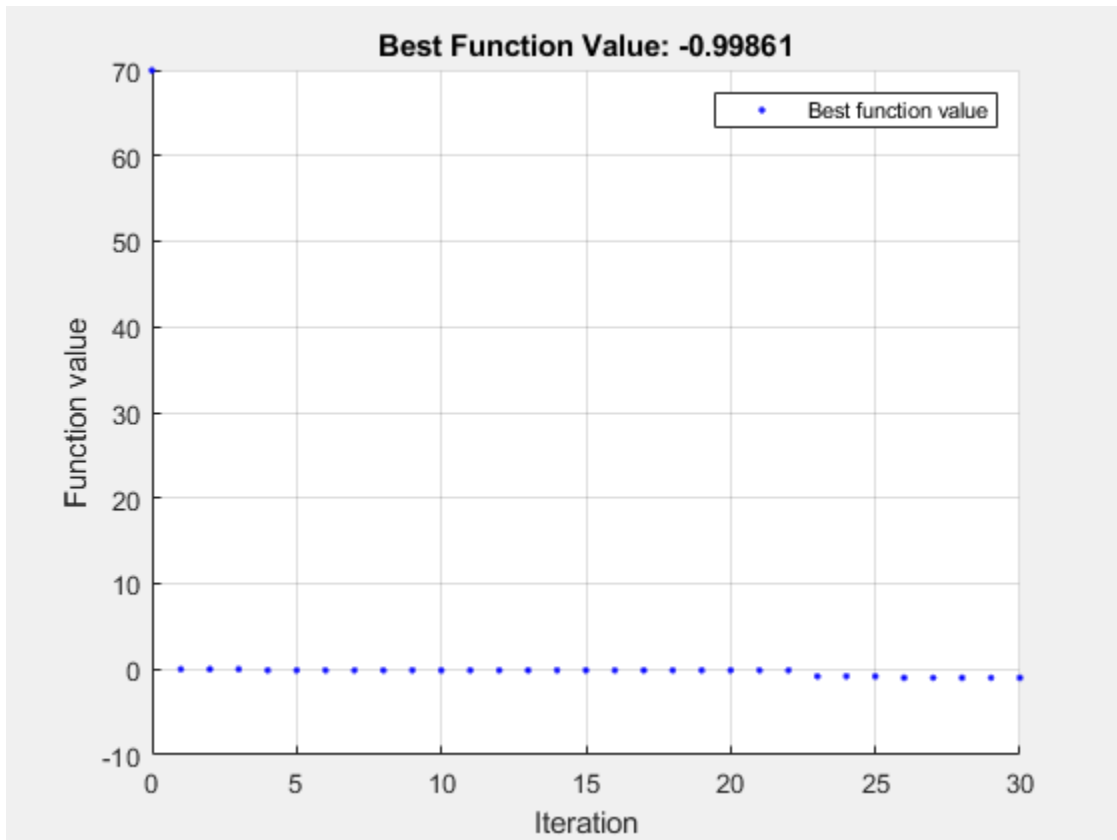
To enable restarting surrogate optimization due to a crash or any other reason, set a checkpoint file name.

```
opts = optimoptions('surrogateopt','CheckpointFile','checkfile.mat');
```

Create an optimization problem and set a small number of function evaluations.

```
rng default % For reproducibility
objconstr = @(x)(4*x(:,1).^2 - 2.1*x(:,1).^4 + x(:,1).^6/3 ...
    + x(:,1).*x(:,2) - 4*x(:,2).^2 + 4*x(:,2).^4);
lb = [-2.1,-2.1];
```

```
ub = -lb;
opts.MaxFunctionEvaluations = 30;
[x,fval,exitflag,output] = surrogateopt(objconstr,lb,ub,opts)
```



Surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
x = 1x2
```

```
0.0067 -0.7343
```

```
fval = -0.9986
```

```
exitflag = 0
```

```
output = struct with fields:
```

```
    elapsedtime: 28.7221
```

```
    funccount: 30
```

```
    constrviolation: 0
```

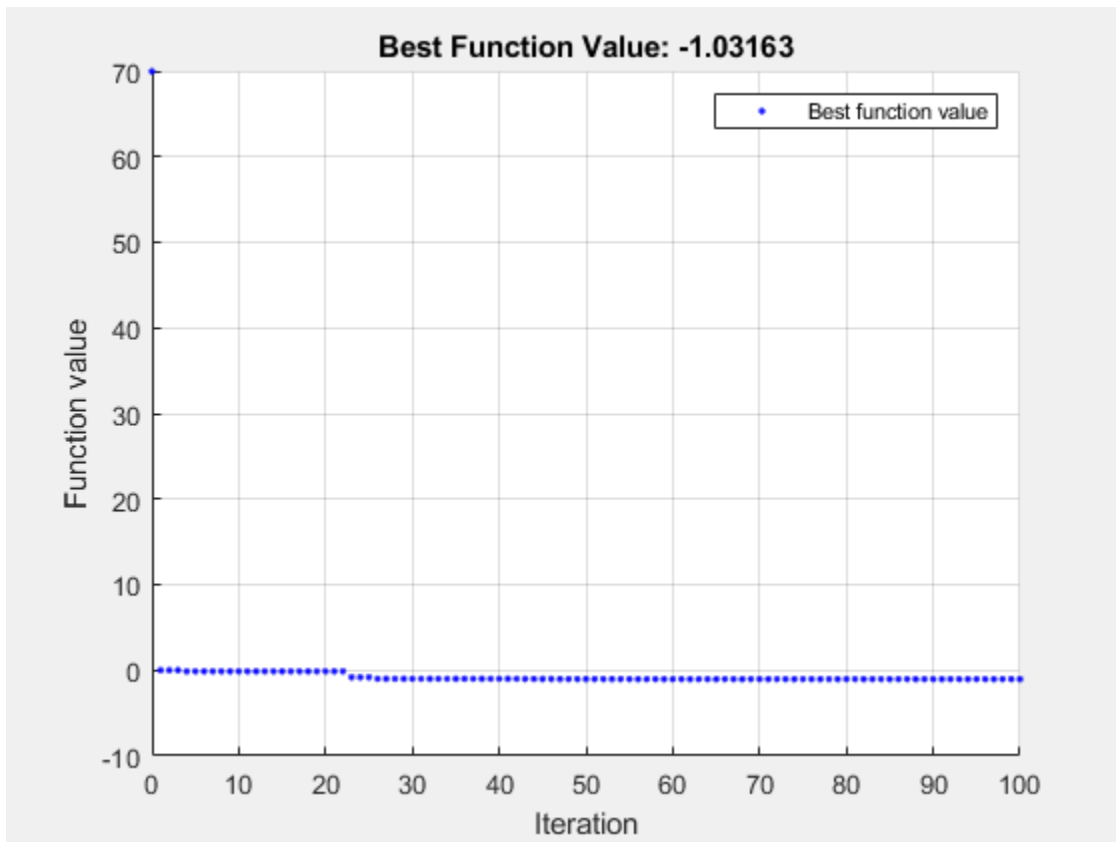
```
        ineq: [1x0 double]
```

```
    rngstate: [1x1 struct]
```

```
    message: 'Surrogateopt stopped because it exceeded the function evaluation limit set
```

Set options to use 100 function evaluations (which means 70 more than already done) and restart the optimization.

```
opts.MaxFunctionEvaluations = 100;
[x2,fval2,exitflag2,output2] = surrogateopt('checkfile.mat',opts)
```



Surrogateopt stopped because it exceeded the function evaluation limit set by 'options.MaxFunctionEvaluations'.

```
x2 = 1×2
```

```
0.0895 -0.7130
```

```
fval2 = -1.0316
```

```
exitflag2 = 0
```

```
output2 = struct with fields:
```

```
elapsedtime: 159.2411
```

```
funccount: 100
```

```
constrviolation: 0
```

```
ineq: [1×0 double]
```

```
rngstate: [1×1 struct]
```

```
message: 'Surrogateopt stopped because it exceeded the function evaluation limit set
```

Input Arguments

objconstr — Objective function and nonlinear constraint

function handle | function name

Objective function and nonlinear constraint, specified as a function handle or function name. `objconstr` accepts a single argument `x`, where `x` is typically a row vector. However, when the `Vectorized` option is `true`, `x` is a matrix containing `options.BatchUpdateInterval` rows; each row represents one point to evaluate. `objconstr` returns one of the following:

- Real scalar `fval = objconstr(x)`.
- Structure. If the structure contains the field `Fval`, then `surrogateopt` attempts to minimize `objconstr(x).Fval`. If the structure contains the field `Ineq`, then `surrogateopt` attempts to make all components of that field nonpositive: `objconstr(x).Ineq <= 0` for all entries. `objconstr(x)` must include either the `Fval` or `Ineq` fields, or both. `surrogateopt` ignores other fields.

When the `Vectorized` option is `true` and the `BatchUpdateInterval` is greater than one, `objconstr` operates on each row of `x` and returns one of the following:

- Real vector `fval = objconstr(x)`. `fval` is a column vector with `options.BatchUpdateInterval` entries (or fewer for the last function evaluation when `BatchUpdateInterval` does not evenly divide `MaxFunctionEvaluations`).
- Structure with vector entries. If the structure contains the field `Fval`, then `surrogateopt` attempts to minimize `objconstr(x).Fval`, and `objconstr(x).Fval` is a vector of length `BatchUpdateInterval` (or less). If the structure contains the field `Ineq`, then `surrogateopt` attempts to make all components of that field nonpositive: `objconstr(x).Ineq <= 0` for all entries, and `objconstr(x).Ineq` contains up to `BatchUpdateInterval` entries.

The objective function `objconstr.Fval` can be empty (`[]`), in which case `surrogateopt` attempts to find a point satisfying all the constraints. See “Solve Feasibility Problem” on page 11-78.

For examples using a nonlinear constraint, see “Solve Problem with Nonlinear Constraints” on page 18-192, “Surrogate Optimization with Nonlinear Constraint” on page 11-41, and “Solve Feasibility Problem” on page 11-78. For information on converting between the `surrogateopt` structure syntax and other solvers, see `packfcn` and “Convert Nonlinear Constraints Between `surrogateopt` Form and Other Solver Forms” on page 11-74. For an example using vectorized batch evaluations, see “Vectorized Surrogate Optimization for Custom Parallel Simulation” on page 11-92.

Data Types: `function_handle` | `char` | `string`

lb — Lower bounds

finite real vector

Lower bounds, specified as a finite real vector. `lb` represents the lower bounds element-wise in $lb \leq x \leq ub$. The lengths of `lb` and `ub` must be equal to the number of variables that `objconstr` accepts.

Caution Although `lb` is optional for most solvers, `lb` is a required input for `surrogateopt`.

Note `surrogateopt` allows equal entries in `lb` and `ub`. For each `i` in `intcon`, you must have `ceil(lb(i)) <= floor(ub(i))`. See “Construct Surrogate Details” on page 11-4.

Example: `lb = [0; -20; 4]` means $x(1) \geq 0, x(2) \geq -20, x(3) \geq 4$.

Data Types: `double`

ub — Upper bounds

finite real vector

Upper bounds, specified as a finite real vector. `ub` represents the upper bounds element-wise in $lb \leq x \leq ub$. The lengths of `lb` and `ub` must be equal to the number of variables that `objconstr` accepts.

Caution Although `ub` is optional for most solvers, `ub` is a required input for `surrogateopt`.

Note `surrogateopt` allows equal entries in `lb` and `ub`. For each `i` in `intcon`, you must have `ceil(lb(i)) <= floor(ub(i))`. See “Construct Surrogate Details” on page 11-4.

Example: `ub = [10; -20; 4]` means $x(1) \leq 10, x(2) \leq -20, x(3) \leq 4$.

Data Types: `double`

intcon — Integer variables

vector of positive integers

Integer variables, specified as a vector of positive integers with values from 1 to the number of problem variables. Each value in `intcon` represents an `x` component that is integer-valued.

Example: To specify that the even entries in `x` are integer-valued, set `intcon` to `2:2:nvars`.

Data Types: `double`

A — Linear inequality constraints

real matrix

Linear inequality constraints, specified as a real matrix. `A` is an `M`-by-`nvars` matrix, where `M` is the number of inequalities.

`A` encodes the `M` linear inequalities

$$A*x \leq b,$$

where `x` is the column vector of `nvars` variables `x(:)`, and `b` is a column vector with `M` elements.

For example, to specify

$$\begin{aligned} x_1 + 2x_2 &\leq 10 \\ 3x_1 + 4x_2 &\leq 20 \\ 5x_1 + 6x_2 &\leq 30, \end{aligned}$$

give these constraints:

```
A = [1,2;3,4;5,6];
b = [10;20;30];
```

Example: To specify that the control variables sum to 1 or less, give the constraints $A = \text{ones}(1, N)$ and $b = 1$.

Data Types: double

b – Linear inequality constraints

real vector

Linear inequality constraints, specified as a real vector. b is an M -element vector related to the A matrix. If you pass b as a row vector, solvers internally convert b to the column vector $b(:)$.

b encodes the M linear inequalities

$$A*x \leq b,$$

where x is the column vector of N variables $x(:)$, and A is a matrix of size M -by- N .

For example, to specify

$$\begin{aligned}x_1 + 2x_2 &\leq 10 \\3x_1 + 4x_2 &\leq 20 \\5x_1 + 6x_2 &\leq 30,\end{aligned}$$

give these constraints:

```
A = [1,2;3,4;5,6];
b = [10;20;30];
```

Example: To specify that the control variables sum to 1 or less, give the constraints $A = \text{ones}(1, N)$ and $b = 1$.

Data Types: double

Aeq – Linear equality constraints

real matrix

Linear equality constraints, specified as a real matrix. Aeq is an Me -by- $nvars$ matrix, where Me is the number of equalities.

Aeq encodes the Me linear equalities

$$Aeq*x = beq,$$

where x is the column vector of N variables $x(:)$, and beq is a column vector with Me elements.

For example, to specify

$$\begin{aligned}x_1 + 2x_2 + 3x_3 &= 10 \\2x_1 + 4x_2 + x_3 &= 20,\end{aligned}$$

give these constraints:

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```

Example: To specify that the control variables sum to 1, give the constraints $A_{eq} = \text{ones}(1, N)$ and $beq = 1$.

Data Types: double

beq — Linear equality constraints

real vector

Linear equality constraints, specified as a real vector. **beq** is an M_{eq} -element vector related to the **Aeq** matrix. If you pass **beq** as a row vector, solvers internally convert **beq** to the column vector **beq(:)**.

beq encodes the M_{eq} linear equalities

$$A_{eq} * x = beq,$$

where x is the column vector of N variables $x(:)$, and A_{eq} is a matrix of size M_{eq} -by- N .

For example, to specify

$$x_1 + 2x_2 + 3x_3 = 10$$

$$2x_1 + 4x_2 + x_3 = 20,$$

give these constraints:

$$A_{eq} = [1, 2, 3; 2, 4, 1];$$

$$beq = [10; 20];$$

Example: To specify that the control variables sum to 1, give the constraints $A_{eq} = \text{ones}(1, N)$ and $beq = 1$.

Data Types: double

options — Options

output of `optimoptions`

Options, specified as the output of `optimoptions`.

For more information, see “Surrogate Optimization Options” on page 17-51.

Option	Description	Values
<code>BatchUpdateInterval</code>	<ul style="list-style-type: none"> Number of function evaluations before the surrogate is updated. Number of points to pass in a vectorized evaluation. When <code>UseVectorized</code> is true, <code>surrogateopt</code> passes a matrix of size <code>BatchUpdateInterval</code>-by-<code>nvar</code>, where <code>nvar</code> is the number of problem variables. Each row of the matrix represents one evaluation point. Output functions and plot functions are updated after each batch is evaluated completely. 	Positive integer. Default is 1.

Option	Description	Values
CheckpointFile	<p>File name for checkpointing and restarting optimization. The file has the <code>.mat</code> data type. See “Work with Checkpoint Files” on page 11-56.</p> <p>Checkpointing takes time. This overhead is especially noticeable for functions that otherwise take little time to evaluate.</p>	<p>File name or file path, given as a string or character array. If you specify a file name without a path, <code>surrogateopt</code> saves the checkpoint file in the current folder.</p>
ConstraintTolerance	<p>Tolerance on nonlinear constraints, measured as the maximum of all nonlinear constraint function values, where positive values indicate a violation. This tolerance is an absolute (not relative) tolerance; see “Tolerances and Stopping Criteria”.</p>	<p>Positive scalar. Default is <code>1e-3</code>.</p>
Display	<p>Level of display returned at the command line.</p>	<ul style="list-style-type: none"> • <code>'final'</code> (default) — Exit message at the end of the iterations. • <code>'off'</code> or the equivalent <code>'none'</code> — No display. • <code>'iter'</code> — Iterative display; see “Command-Line Display” on page 17-52.
InitialPoints	<p>Initial points for solver.</p>	<p>Matrix of initial points, where each row is one point. Or, a structure with field <code>X</code>, whose value is a matrix of initial points, and these optional fields:</p> <ul style="list-style-type: none"> • <code>Fval</code>, a vector containing the values of the objective function at the initial points • <code>Ineq</code>, a matrix containing nonlinear inequality constraint values <p>See “Algorithm Control” on page 17-51. Default is <code>[]</code>.</p>

Option	Description	Values
MaxFunctionEvaluations	Maximum number of objective function evaluations, a stopping criterion.	Positive integer. Default is $\max(200, 50 \cdot nvar)$, where $nvar$ is the number of problem variables.
MaxTime	Maximum running time in seconds. The actual running time can exceed MaxTime because of the time required to evaluate an objective function or because of parallel processing delays.	Positive scalar. Default is Inf.
MinSampleDistance	Minimum distance between trial points generated by the adaptive sampler. See "Surrogate Optimization Algorithm" on page 11-3.	Positive scalar. Default is $1e-6$.
MinSurrogatePoints	Minimum number of random sample points to create at the start of a surrogate creation phase. See "Surrogate Optimization Algorithm" on page 11-3. When $BatchUpdateInterval > 1$, the minimum number of random sample points used to create a surrogate is the larger of MinSurrogatePoints and BatchUpdateInterval.	Integer at least $nvar + 1$. Default is $\max(20, 2 \cdot nvar)$, where $nvar$ is the number of problem variables.
ObjectiveLimit	Tolerance on the objective function value. If a calculated objective function value of a feasible point is less than ObjectiveLimit, the algorithm stops.	Double scalar value. Default is -Inf.
OutputFcn	Output function to report on solver progress or to stop the solver. See "Output Function" on page 17-53.	Function name, function handle, or cell array of function names or handles. Default is [].

Option	Description	Values
PlotFcn	Plot function to display solver progress or to stop solver. See “Plot Function” on page 17-55.	<p>Function name, function handle, or cell array of function names or handles. Built-in plot functions are:</p> <ul style="list-style-type: none"> • 'optimplotfvalconstr' (default) — Plot the best feasible objective function value found as a line plot. If there is no objective function, plot the maximum nonlinear constraint violation as a line plot. • The plot shows infeasible points as red and feasible points as blue. • If there is no objective function, the plot title shows the number of feasible solutions. • 'optimplotfval' — Plot the best objective function value found as a line plot. • 'optimplotx' — Plot the best solution found as a bar chart. • 'surrogateoptplot' — Plot the objective function value at each iteration, showing which phase of the algorithm produces the value and the best value found both in this phase and overall. See “Interpret surrogateoptplot” on page 11-25.

Option	Description	Values
UseParallel	<p>Boolean value indicating whether to compute objective function values in parallel.</p> <p>You cannot specify both <code>UseParallel = true</code> and <code>UseVectorized = true</code>. If you set both to true, the solver ignores <code>UseVectorized</code> and attempts to compute in parallel using a parallel pool, if possible.</p>	Boolean. Default is false. For algorithmic details, see “Parallel surrogateopt Algorithm” on page 11-10.
UseVectorized	<p>Boolean value indicating whether to compute objective function values in batches of size <code>BatchUpdateInterval</code>.</p> <p>You cannot specify both <code>UseParallel = true</code> and <code>UseVectorized = true</code>. If you set both to true, the solver ignores <code>UseVectorized</code> and attempts to compute in parallel using a parallel pool, if possible.</p>	Boolean. Default is false. For an example, see “Vectorized Surrogate Optimization for Custom Parallel Simulation” on page 11-92.

Example: `options = optimoptions('surrogateopt','Display','iter','UseParallel',true)`

problem — Problem structure

structure

Problem structure, specified as a structure with these fields:

- `objective` — Objective function, which can include nonlinear constraints, specified as a function name or function handle
- `lb` — Lower bounds for `x`
- `ub` — Upper bounds for `x`
- `solver` — 'surrogateopt'
- `Aineq` — Matrix for linear inequality constraints (optional)
- `bineq` — Vector for linear inequality constraints (optional)
- `Aeq` — Matrix for linear equality constraints (optional)
- `beq` — Vector for linear equality constraints (optional)
- `options` — Options created with `optimoptions`
- `rngstate` — Field to reset the state of the random number generator (optional)
- `intcon` — Field specifying integer-valued `x` components (optional)

Note These problem fields are required: `objective`, `lb`, `ub`, `solver`, and `options`.

Data Types: `struct`

checkpointFile — Path to checkpoint file

string | character vector

Path to a checkpoint file, specified as a string or character vector. A checkpoint file has the `.mat` extension. If you specify a file name without a path, `surrogateopt` uses a checkpoint file in the current folder.

A checkpoint file stores the state of an optimization for resuming the optimization. `surrogateopt` updates the checkpoint file at each function evaluation, so you can resume the optimization even when `surrogateopt` halts prematurely. For an example, see “Restart Surrogate Optimization from Checkpoint File” on page 18-201.

`surrogateopt` creates a checkpoint file when it has a valid `CheckpointFile` option.

You can change some options when resuming from a checkpoint file. See `opts`.

The data in a checkpoint file is in `.mat` format. To avoid errors or other unexpected results, do not modify the data before calling `surrogateopt`.

Warning Do not resume `surrogateopt` from a checkpoint file created with a different MATLAB version. `surrogateopt` can throw an error or give inconsistent results.

Example: `'checkfile.mat'`

Example: `"C:\Program Files\MATLAB\docs\checkpointNov2019.mat"`

Data Types: `char` | `string`

opts — Options for resuming from checkpoint file

`[]` (default) | `optimoptions` options from a restricted set

Options for resuming optimization from the checkpoint file, specified as `optimoptions` options (from a restricted set) that you can change from the original options. The options you can change are:

- `BatchUpdateInterval`
- `CheckpointFile`
- `Display`
- `MaxFunctionEvaluations`
- `MaxTime`
- `MinSurrogatePoints`
- `ObjectiveLimit`
- `OutputFcn`
- `PlotFcn`
- `UseParallel`
- `UseVectorized`

Example: `opts = optimoptions(options,'MaxFunctionEvaluations',400);`

Output Arguments

x — Solution

real vector

Solution, returned as a real vector. x has the same length as lb and ub .

fval — Objective function value at solution

real number

Objective function value at the solution, returned as a real number.

- When `objconstr` returns a scalar, `fval` is the scalar `objconstr(x)`.
- When `objconstr` returns a structure, `fval` is the value `objconstr(x).Fval`, the objective function value at x (if this value exists).

exitflag — Reason surrogateopt stopped

integer

Reason surrogateopt stopped, returned as one of the integer values described in this table.

Exit Flag	Description
10	Problem has a unique feasible solution due to one of the following: <ul style="list-style-type: none"> • All upper bounds ub are equal to the lower bounds lb. • The linear equality constraints $Aeq*x = beq$ and the bounds have a unique solution point. <p>surrogateopt returns the feasible point and function value without performing any optimization.</p>
3	Feasible point found. Solver stopped because too few new feasible points were found to continue.
1	The objective function value is less than <code>options.ObjectiveLimit</code> . This exit flag takes precedence over exit flag 10 when both apply.
0	The number of function evaluations exceeds <code>options.MaxFunctionEvaluations</code> or the elapsed time exceeds <code>options.MaxTime</code> . If the problem has nonlinear inequalities, the solution is feasible.
-1	The optimization is terminated by an output function or plot function.
-2	No feasible point is found due to one of the following: <ul style="list-style-type: none"> • A lower bound $lb(i)$ exceeds a corresponding upper bound $ub(i)$. Or one or more $ceil(lb(i))$ exceeds a corresponding $floor(ub(i))$ for i in <code>intcon</code>. In this case, surrogateopt returns $x = []$ and <code>fval = []</code>. • $lb = ub$ and the point lb is infeasible. In this case, $x = lb$, and <code>fval = objconstr(x).Fval</code>. • The linear and, if present, integer constraints are infeasible together with the bounds. In this case, surrogateopt returns $x = []$ and <code>fval = []</code>. • The bounds, integer, and linear constraints are feasible, but no feasible solution is found with nonlinear constraints. In this case, x is the point of least maximum infeasibility of nonlinear constraints, and <code>fval = objconstr(x).Fval</code>.

output — Information about optimization process

structure

Information about the optimization process, returned as a structure with these fields:

- `funcccount` — Total number of function evaluations.
- `elapsedtime` — Time spent running the solver in seconds, as measured by `tic/toc`.
- `message` — Reason why the algorithm stopped.
- `constrviolation` — Maximum nonlinear constraint violation, if any. `constrviolation = max(output.ineq)`.
- `ineq` — Nonlinear inequality constraint value at the solution `x`. If `objconstr` returns a structure, then `ineq = objconstr(x).Ineq`. Otherwise, `ineq` is empty.
- `rngstate` — State of the MATLAB random number generator just before the algorithm starts. Use this field to reproduce your results. See “Reproduce Results” on page 8-67, which discusses using `rngstate` for `ga`.

trials — Points evaluated

structure

Points evaluated, returned as a structure with these fields:

- `X` — Matrix with `nvars` columns, where `nvars` is the length of `lb` or `ub`. Each row of `X` represents one point evaluated by `surrogateopt`.
- `Fval` — Column vector, where each entry is the objective function value of the corresponding row of `X`.
- `Ineq` — Matrix with each row representing the constraint function values of the corresponding row of `X`.

The `trials` structure has the same form as the `options.InitialPoints` structure. So, you can continue an optimization by passing the `trials` structure as the `InitialPoints` option.

Algorithms

`surrogateopt` repeatedly performs these steps:

- 1 Create a set of trial points by sampling `MinSurrogatePoints` random points within the bounds, and evaluate the objective function at the trial points.
- 2 Create a surrogate model of the objective function by interpolating a radial basis function through all of the random trial points.
- 3 Create a merit function that gives some weight to the surrogate and some weight to the distance from the trial points. Locate a small value of the merit function by randomly sampling the merit function in a region around the incumbent point (best point found since the last surrogate reset). Use this point, called the adaptive point, as a new trial point.
- 4 Evaluate the objective at the adaptive point, and update the surrogate based on this point and its value. Count a "success" if the objective function value is sufficiently lower than the previous best (lowest) value observed, and count a "failure" otherwise.
- 5 Update the dispersion of the sample distribution upwards if three successes occur before `max(nvar, 5)` failures, where `nvar` is the number of dimensions. Update the dispersion downwards if `max(nvar, 5)` failures occur before three successes.
- 6 Continue from step 3 until all trial points are within `MinSampleDistance` of the evaluated points. At that time, reset the surrogate by discarding all adaptive points from the surrogate, reset the scale, and go back to step 1 to create `MinSurrogatePoints` new random trial points for evaluation.

For details, see “Surrogate Optimization Algorithm” on page 11-3.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for surrogateopt.

Version History

Introduced in R2018b

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set the 'UseParallel' option to true.

```
options = optimoptions('solvername','UseParallel',true)
```

For more information, see “How to Use Parallel Processing in Global Optimization Toolbox” on page 16-11.

See Also

patternsearch | optimoptions | packfcn | **Optimize**

Topics

“Surrogate Optimization”

“Local vs. Global Optima”

“Mixed Integer ga Optimization” on page 8-40

“Surrogate Optimization Options” on page 17-51

