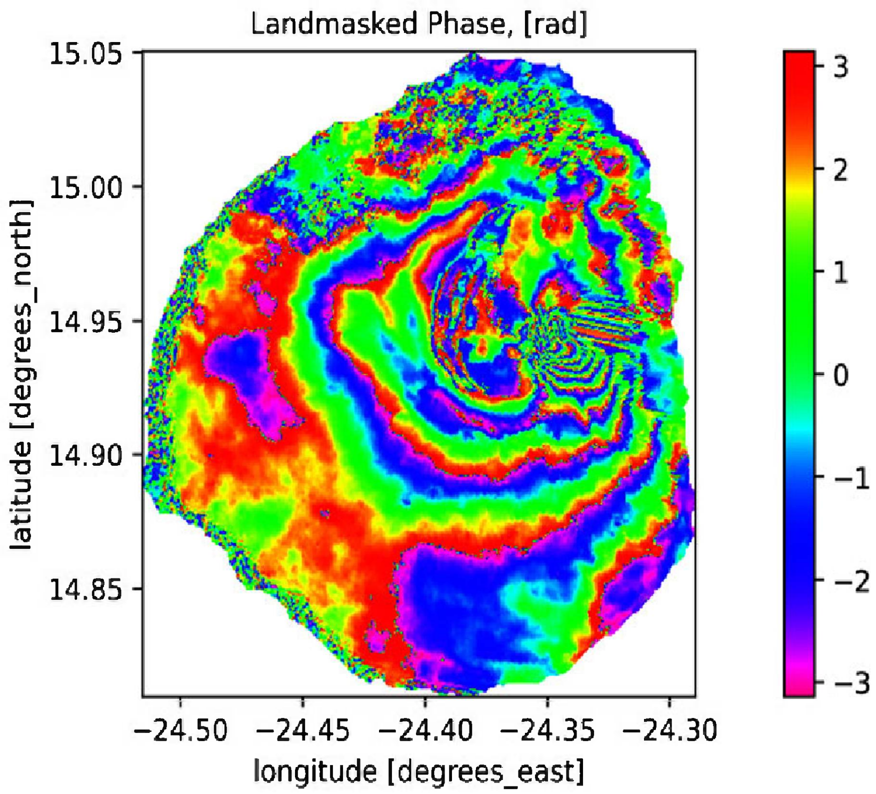
ALEXEY PECHNIKOV

PyGMTSAR: Sentinel-1 Python InSAR

An Introduction



PyGMTSAR: Sentinel-1 Python InSAR

An Introduction

Alexey Pechnikov

2023-07-01

Written by Alexey Pechnikov.

Annotation

The “PyGMTSAR: Sentinel-1 Python InSAR” book series serves as your gateway to mastering the innovative world of Sentinel-1 satellite interferometry using the open-source Python InSAR library, PyGMTSAR. Authored by the developer himself, these books act as hands-on guides for working with PyGMTSAR, whether through Jupyter notebooks or console Python scripts.

The book “PyGMTSAR: Sentinel-1 Python InSAR. An Introduction” employs Google Colab, a free-to-use cloud service, as an ideal platform for beginners. Readers can explore the applications of PyGMTSAR, from seismic activity tracking to infrastructure health assessment, through a series of interactive notebooks. Each notebook comes complete with adaptable instructions to facilitate personalized learning.

The guide also introduces Docker Desktop, an advanced open-source platform for containerization. The PyGMTSAR Docker image sets up a workspace similar to a traditional one, enabling more intense computations on your local computer and on cloud hosts. All the Google Colab examples are available.

This tutorial sheds light on the principles of lazy and delayed computations. It explains how Dask, an advanced task scheduler, intelligently partitions and schedules tasks. These insights enhance your ability to handle Big Data processing with PyGMTSAR efficiently, whether on your local machine or cloud-based systems.

Whether you’re a student, a researcher, or an industry professional with an interest in remote sensing and earth observation, the “PyGMTSAR: Sentinel-1 Python InSAR. An Introduction” book equips you with the necessary skills and knowledge to navigate Python-based satellite interferometry.

Table of Contents

1. Overview

In this chapter, we will explore two crucial elements: the fundamental concepts underlying satellite interferometry and an introduction to PyGMTSAR, a powerful tool for processing and interpreting InSAR data.

Section 1.1 provides a roadmap to understanding the basics of satellite interferometry, specifically Interferometric Synthetic Aperture Radar (InSAR). This advanced remote sensing technique involves the use of two or more synthetic aperture radar (SAR) images to generate highly accurate maps of surface deformation or digital elevation models of terrain. This process relies on the detailed calculation of the phase difference between return signals from two nearly identical images, known as interferograms. Furthermore, it highlights how PyGMTSAR is utilized in processing these images, performing operations like Doppler correction, topography correction, and applying Gaussian and Werner/Goldstein filters for noise reduction. The section also elaborates on the SBAS technique for displacement calculation and how the resulting time-series data can be analyzed to extract trends and seasonal movements.

Section 1.2 introduces PyGMTSAR, a powerful software designed to encapsulate the complexities of InSAR processing. The utility of PyGMTSAR lies in its ability to automate advanced algorithms for InSAR data processing, making the extraction of valuable insights from the data accessible regardless of your technical background. As an analogy, the principles of InSAR are compared to those of common Wi-Fi or 4G/5G cellular connectivity, illustrating PyGMTSAR’s intention to make the technology as user-friendly and accessible as possible. A notable feature of PyGMTSAR is its capability to run directly on Google Colab, thereby eliminating the need for local software installations. Finally, this section notes the potential of PyGMTSAR to handle large datasets efficiently, further contributing to its ease of use and power.

1.1. The Basics of Satellite Interferometry

Satellite Interferometry, specifically Interferometric Synthetic Aperture Radar (InSAR), is a remote sensing technique that utilizes two or more synthetic aperture radar (SAR) images to generate maps of surface deformation or digital elevation models of the terrain. A satellite emits a radar signal towards Earth; this signal interacts with the surface and then reflects back.

InSAR includes several specialized techniques, including Differential InSAR (DInSAR) and Time series DInSAR, which involve comparing more than two images over time to analyze surface changes more precisely or to track changes over time.

The “D” in DInSAR stands for “Differential”, indicating that it is used to observe phase changes between two images over a given time period. DInSAR is typically employed to monitor subsidence/uplift or lateral deformation. To separate the deformation signal from topographic contributions, the topographic phase is simulated using a reference Digital Elevation Model (DEM) and then subtracted from the interferogram.

Time series InSAR, also known as multi-temporal InSAR, is an extension of DInSAR. It involves the use of multiple SAR images collected over the same area at different time intervals to monitor and measure changes that occur over time, such as land displacement due to seismic, volcanic, or anthropogenic activities.

Instead of using just two images as in traditional DInSAR, time series InSAR involves the generation and analysis of multiple differential interferograms, each created using a different pair of images from the available time series. The main advantage of this approach is that it allows for the separation of the deformation signal from other unwanted effects (like atmospheric disturbances), leading to more accurate displacement maps.

There are several processing techniques used for time series analysis in InSAR, including the Persistent Scatterer Interferometry (PSI) and the Small BAseline Subset (SBAS) methods. These techniques have proven to be highly effective in areas with a high density of stable scatterers and moderate to high temporal decorrelation.

In summary, InSAR is related to distance changes from the satellite and is suitable for DEM generation. For surface changes monitoring, we need to estimate distance changes and this technique is called Differential InSAR (DInSAR). More generic analysis based on multiple DInSAR pairs is called Time series DInSAR.

In this book, we use the most common term InSAR while technically we are discussing Time series Differential InSAR. For the time series analysis, we apply the Small BAseline Subset (SBAS) method. PyGMTSAR allows for some inclusion of Persistent Scatterer Interferometry (PSI) in the SBAS analysis, and this direction looks promising. There is a lot of research being done to merge the best features of SBAS and PSI, and we are in the process of enhancing PyGMTSAR in this regard.

The term ‘synthetic aperture’ refers to a computational technique that involves continuously transmitting and receiving signals and integrating the waves received from various points on Earth’s surface. This approach, although increasing the complexity of image processing compared to instant snapshots, significantly enhances image resolution. A radar image is computational in its essence as it’s not captured in a single moment but post-processed from a long-duration measured signal. Available for download, Sentinel-1 SLC scenes are focused synthetic radar images and form the data source for PyGMTSAR computations.

An interferogram, entirely computational, is created from the phase difference of the return signals from two separate yet nearly identical images. To achieve nearly identical images, alignment is critical. The well-documented Sentinel-1 satellite orbit is used to calculate the geometric correction for a minor spatial difference known as the perpendicular baseline. PyGMTSAR can automatically download the orbit files for selected scenes.

Because of the high number of image pair combinations, it’s technically unfeasible to provide all pre-calculated interferograms for download, akin to the Sentinel-1 scenes themselves. PyGMTSAR computes interferograms for selected image pairs using downloaded Sentinel-1 scenes and orbit files. This operation is straightforward, hinging on simple math operations. However, for practical reasons, additional processing, such as Doppler correction using orbit information and topography correction using the Earth’s area covered in the Sentinel-1 images, is performed. PyGMTSAR automatically downloads and converts SRTM (Shuttle Radar Topography Mission) topography heights to the WGS84 ellipsoid from standalone EGM96 geoid for SRTM and other topography models.

Gaussian and Werner/Goldstein filters are applied to reduce radiometric noise and highlight changes, enhancing the visibility of interferogram fringes at the expense of spatial resolution. As of now, Gaussian and Werner/Goldstein filters are mandatory, but work is underway to make them optional and allow for persistent scatterers interferometry features. The well-known interferogram fringes result from phase measurement in the interval 2π, often referred to as the wrapped phase.

The SNAPHU (Statistical-Cost, Network-Flow Algorithm for Phase Unwrapping) program is used for 2D unwrapping to produce a continuous phase. PyGMTSAR supports both single-raster and tiled SNAPHU unwrapping using predefined or custom unwrapping configurations. Interestingly, we can visually estimate the changes without unwrapping by counting the so-called fringes (a phase change of 2π mapped as a full colormap range) and locating their centers, which can indicate deformation epicenters. By design, a single fringe signifies a 2π phase change, corresponding to a round-trip (to the ground and back to the satellite) change in distance equivalent to a single radar signal wavelength, or equivalently, a one-way change of half of this wavelength. In other words, while SNAPHU is a highly useful tool for satellite interferometry, it isn’t essential.

By using the continuous unwrapped phase, we can gauge the distance change between the satellite and each point on the ground. This change is partially related to surface deformation, but it also depends on atmospheric conditions and other factors. Fundamentally, InSAR measures Earth’s surface movements with remarkable precision, often down to a few millimeters or even better over a large series of interferograms.

Knowing the unwrapped phase in radians, it’s easy to calculate LOS (Line-of-sight) displacement in millimeters and its vertical and east-west projections using the scenes’ geometry. For better accuracy, PyGMTSAR projection transformations are based on a complete grid of the incidence angle computed for every ground pixel. The sum of the projections obtained for two orbits (ascending and descending) are the real displacement components. The displacements are related to the Sentinel-1 image capture date intervals, not to specific dates.

Small BAseline Subset (SBAS) displacements calculation applies a correlation-weighted least-squares algorithm to the interferogram displacements to produce continuous displacement timeseries for every pixel. More interferograms mean more displacements on intersected intervals and potentially better result accuracy. The SBAS technique simplifies atmospheric phase exclusion as atmospheric phase delays are included with different signs in different interferograms with common cloudy images.

The SBAS displacement often presents substantial noise, making it difficult to discern real-world changes. To overcome this, signal processing can be used for pixel-wise analysis to extract trends and seasonal movements from the time series. PyGMTSAR uses Seasonal-Trend Decomposition using LOESS (STL) for this purpose. Trend extraction applies the power of mathematical statistics for noise reduction. A seasonal trend tends to be vertical and repetitive by definition; thus, we can estimate it more accurately than merely relying on SBAS time series. The non-seasonal trend, averaged over the duration of the time series, yields the average velocity. These extracted features reflect real-world characteristics that can be validated by ground measurements, providing valuable insights for surface monitoring.

And still, that’s not the end of the interferometric processing journey. The produced results need to be geocoded from scene radar coordinates to geographic coordinates and exported in a common format. PyGMTSAR processing is based on NetCDF (Network Common Data Form) data files which can be opened directly in open-source QGIS, GDAL, and other GIS (geographic information system) software. Also, PyGMTSAR allows exporting 3D rasters in VTK format for interactive 3D visualization and analysis in open-source ParaView software.

1.2. Introduction to PyGMTSAR

Numerous InSAR software applications have been developed by various experts, from seasoned geophysicists to specialized software development companies, and even driven amateurs. However, PyGMTSAR carves out a unique position among them. It’s the creation of a fundamental radio physics scientist with extensive experience in computer science and applied programming. PyGMTSAR represents a significant investment of time and effort into the development of mathematical models of interferometry and holography, the construction of hardware to test these models in lab environments, and the education of students, postgraduates, and researchers in these concepts. This deep-rooted understanding and proficiency in the core principles and numerical calculation algorithms distinctively set PyGMTSAR apart.

The developer of PyGMTSAR isn’t just a programmer or a geophysicist, but a well-rounded researcher who has worked on every aspect of interferometry, from the initial theoretical modeling to the processing of collected data. This comprehensive perspective, coupled with a commitment to usability, has culminated in a tool that combines cutting-edge scientific accuracy with user-friendly design.

PyGMTSAR simplifies the complex realm of satellite interferometry, making it accessible to everyone, regardless of technical background or expertise. It stands on the belief that advanced scientific tools should be universally available to anyone interested in understanding and harnessing the power of satellite data. The mission of PyGMTSAR is clear — democratize access to advanced remote sensing technologies and inspire a new generation of users and innovators.

The handling of data in the ever-evolving field of satellite remote sensing can become a complicated task. PyGMTSAR rises to the occasion by streamlining the process, making it more manageable and accessible. It simplifies the most advanced algorithms for InSAR data processing for all users, regardless of their technical expertise.

While understanding InSAR can be challenging due to its computational intensity and complex principles, such as synthetic aperture and interferogram, these principles share similarities with those used in everyday technologies like Wi-Fi or 4G/5G cellular connectivity. The key is to encapsulate the complexities of InSAR into a user-friendly tool that doesn't require the user to fully comprehend the technical specifics. This is the ethos behind PyGMTSAR, aiming to be as accessible and straightforward to use as your cellphone or wireless modem.

One of PyGMTSAR's key features is its ability to run directly on Google Colab with a single click. This eliminates the need for local software installations or complicated procedures to generate results. Users can access live, annotated examples directly in their web browsers, allowing them to interactively change source datasets and processing parameters to observe different outcomes - an effective learning tool for understanding InSAR data processing.

The subsequent chapters will provide more specifics on effectively utilizing PyGMTSAR, offering a curated set of examples illustrating how to extract valuable insights from InSAR data. These examples are not only available on Google Colab but also in a Docker image, making them easily accessible to anyone keen on learning.

For those who prefer a more traditional approach, PyGMTSAR can be installed directly on a computer or used on cloud hosts, like Amazon or Google Cloud computing instances. Although this book doesn't cover this procedure, it's well-documented in the PyGMTSAR

Despite its simplicity and ease of use, PyGMTSAR is incredibly powerful and can handle large datasets with ease. Whether processing thousands of interferograms for multiple Sentinel-1 scenes stitched together at high resolution, or producing hundreds of multi-gigabyte displacement rasters on common hardware like an Apple Air laptop, PyGMTSAR maintains an impressive performance level.

PyGMTSAR bridges the gap between complex satellite data and user accessibility in the realm of satellite remote sensing and InSAR data processing. With a development approach centered around simplifying technical complexities, PyGMTSAR emerges as an invaluable resource for anyone seeking to navigate the world of InSAR.

2. Getting Started

In this chapter, you'll discover the first steps to using PyGMTSAR - a software package for satellite interferometry processing. The chapter is divided into two sections, focusing on how to set up and start working with PyGMTSAR in two environments: Google Colab and Docker Desktop.

Section 2.1 introduces Google Colab, a free cloud service, as a convenient way to run PyGMTSAR examples directly in your web browser. Google Colab, although not suitable for all workloads, is ideal for lighter tasks and provides an excellent starting point for beginners. Here, you'll find various example notebooks highlighting the application of PyGMTSAR in different areas like seismology, volcanology, hydrology, and infrastructure monitoring. The instructions provided in these notebooks will guide you on how to adjust the radar scenes and processing parameters to match your specific needs.

Section 2.2 shifts the focus to Docker Desktop, an open-source platform that allows you to run applications in isolated containers. This approach is more suited for those wanting to work offline or locally, or those whose workloads demand more computational power than what Google Colab can provide. Here, you will learn how to download and install Docker Desktop, create a DockerHub account, configure Docker Desktop, download the PyGMTSAR Docker image, and finally run the image using Docker Desktop. This provides an interactive workspace where you can access and use the PyGMTSAR examples in an environment closer to a traditional development setup.

2.1. Launching Online with Google Colab

Google Colaboratory, or Google is a free cloud service that allows you to develop and execute code in Python, directly in your browser. It is similar to Jupyter Notebook, and it offers the added benefits of leveraging Google’s cloud computing infrastructure.

Google Colab provides a host of advantages:

No setup Google Colab is ready to use immediately. PyGMTSAR example notebooks include all the commands to initialize the environment and perform the processing.
Free access to GPUs and Google Colab provides free access to powerful graphics processing units (GPUs) and tensor processing units (TPUs), which are useful for training machine learning models. For now, PyGMTSAR does not use GPUs and TPUs, but these might be helpful to speed up your own code.
Notebooks on Google Colab can be easily shared, allowing for seamless collaboration among teams. This feature can be beneficial for researchers and developers working in groups. Open the example PyGMTSAR notebooks and make your changes and share the updated notebook with everyone or just for selected users.
Integration with Google Drive and Google Colab integrates smoothly with Google Drive and GitHub, making it easy to load data, notebooks, and store your work. You can load Sentinel-1 scenes stored on your Google Drive and save the results.

Interactive tutorials and Google Colab notebooks can contain live code, equations, visualizations, and narrative text, making it a great tool for creating interactive tutorials and documentation. PyGMTSAR uses all the features to provide the example notebooks as the complete interactive learning tutorials.

Given these advantages, Google Colab is an excellent platform to run the PyGMTSAR examples, which provide a comprehensive, self-explained InSAR pipeline right in your web browser, producing detailed graphs and maps. You can use these examples as templates for your tasks, simply by replacing the used radar scenes and altering the processing parameters to meet your specific needs.

To access these notebooks directly in your browser, follow the provided links below. The links are sorted into thematic groups, and the complete list of examples is also available on the PyGMTSAR GitHub page.

Remember, while Google Colab is an excellent resource, it’s crucial to note that it’s not suitable for all use cases because of its limitations in terms of available RAM and processing time. For heavier workloads, you might need to consider more powerful hardware or cloud solutions.

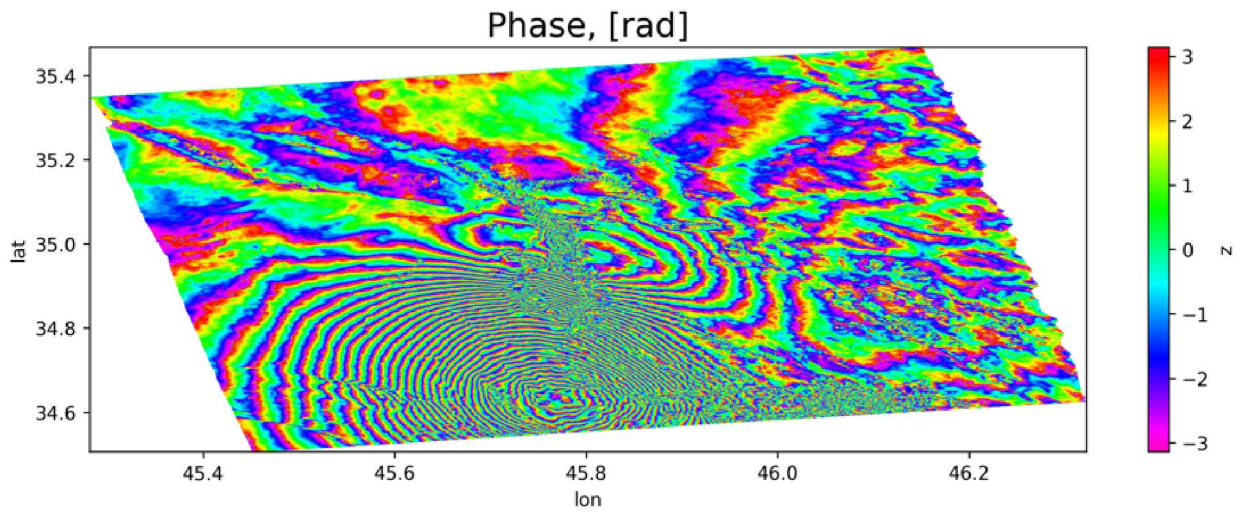
Seismology

InSAR plays a vital role in monitoring and studying seismic activities. It can detect ground deformation before, during, and after earthquakes. This technology provides valuable data on fault systems and magma movements, enhancing our understanding of these natural phenomena. The following examples show the application of InSAR data in studying and interpreting seismic events.

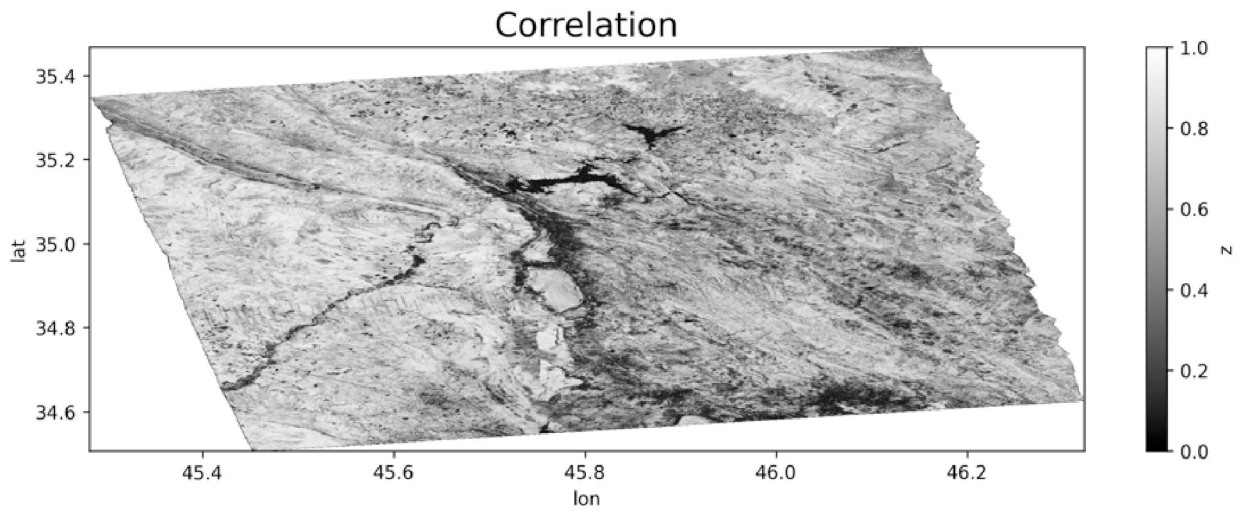
This notebook analyses the 2017 Iran–Iraq Earthquake, with a comparison to results from other software tools such as GMTSAR, GAMMA, and SNAP.

In this analysis, the notebook fetches Sentinel-1 scenes from the Alaska Satellite Facility (ASF). This facilitates the generation of an interferogram and coherence map for a single cropped subswath. It exports NetCDF rasters, which are compatible with QGIS, GDAL, and other GIS software. A comparative review of the results with GMTSAR, SNAP, and GAMMA software is also provided. Note: To generate an interferogram and Line-Of-Sight (LOS) displacement for your area of interest, simply replace the scene names.

The first image illustrates an interferogram of the 2017 Iran–Iraq Earthquake, a seismic event captured using InSAR data. The fringes visible in the image represent different amounts of ground movement that occurred during the event. Each cycle from one color back to the same color represents the half wavelength of displacement in Line-Of-Sight (LOS) direction. Well-defined fringes are indicative of significant ground deformation associated with the earthquake.



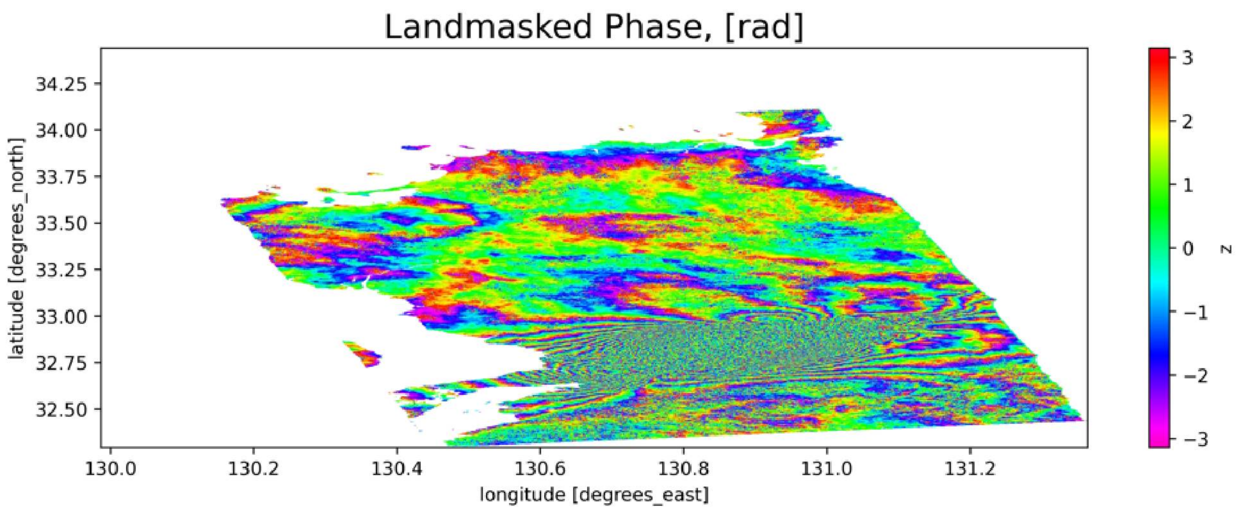
The second image depicts a coherence map, a measure of the quality of the interferometric phase data. High coherence values indicate areas where the phase difference is reliable, and this typically aligns with regions of stable scatterers, like rocks or bare ground. Areas of low coherence, on the other hand, might represent regions where the radar signal was disrupted, perhaps due to vegetation, surface changes, or atmospheric effects.



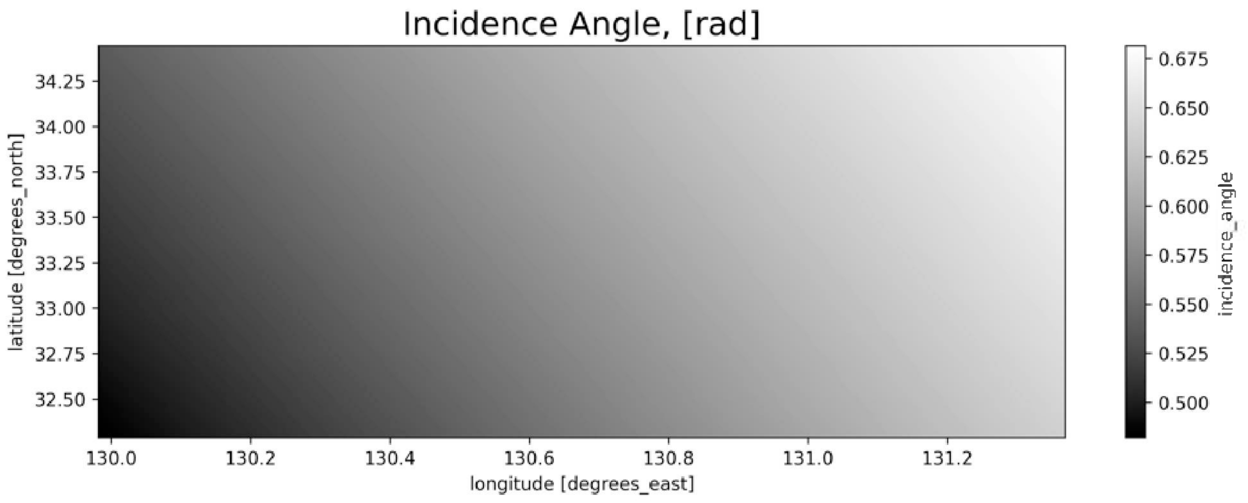
This notebook focuses on the 2016 Kumamoto Earthquake. It demonstrates the processing of InSAR data to create a co-seismic interferogram, which highlights earth's deformation because of the seismic event. The findings are contrasted with results from the ESA Sentinel 1 Toolbox on the Alaska Satellite Facility.

The example shows the processing of a single subswath with a land mask applied to the interferogram, unwrapped phase, and Line-Of-Sight (LOS), as well as east-west and vertical displacement results.

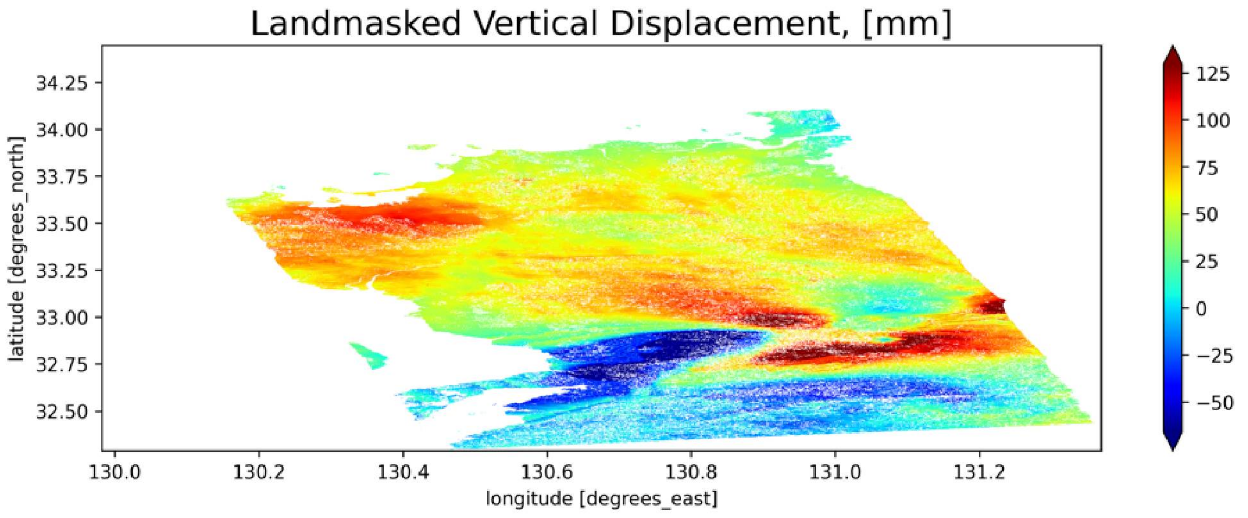
The first image shows the interferogram for the 2016 Kumamoto Earthquake, with a land mask applied. Water bodies are removed from the analysis because they produce a random, or "noisy," phase result in the interferogram due to their dynamic nature. The visible fringes in the image, especially those that are challenging to interpret, highlight the complex ground movements that occurred during the seismic event.



The second image displays an incidence angle map which corresponds to the angle at which the satellite's radar signal hits the ground. This information is crucial for converting the Line-Of-Sight (LOS) displacement to horizontal and vertical displacement components.



The third image illustrates the estimated vertical displacement for the 2016 Kumamoto Earthquake. The applied land mask removes the noisy offshore values.



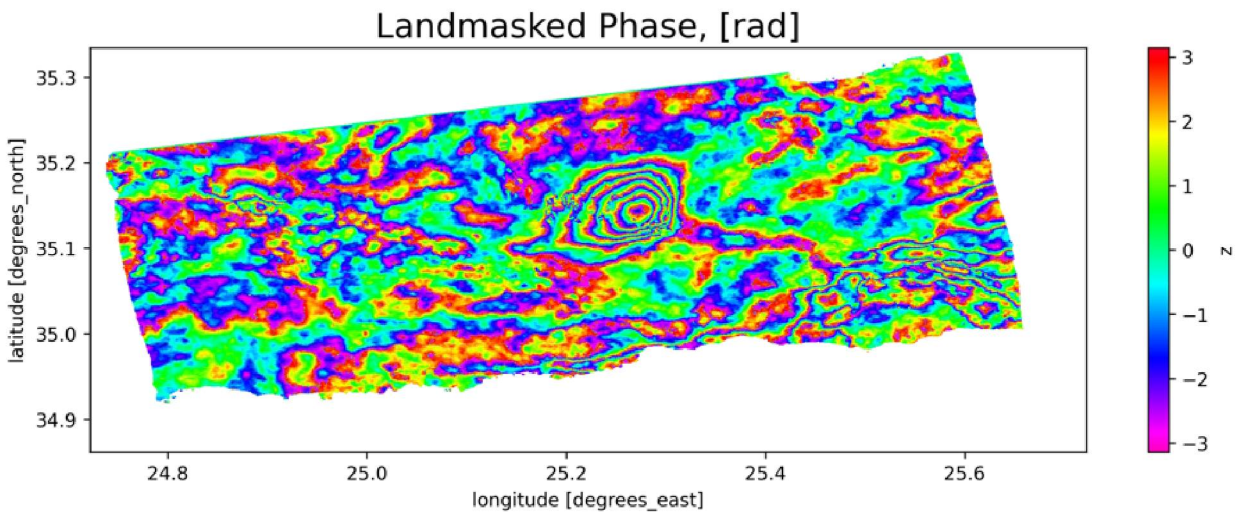
The 2021 Crete Earthquake Co-Seismic Interferogram notebook showcases the application of InSAR in processing data from a recent seismic event to understand the associated ground deformations. The results are contrasted with a report from the Centre of EO Research & Satellite Remote Sensing in Greece.

This example illustrates the processing of a single, cropped subswath with a land mask applied to the interferogram, unwrapped phase, and Line-Of-Sight (LOS), east-west, and vertical displacement results.

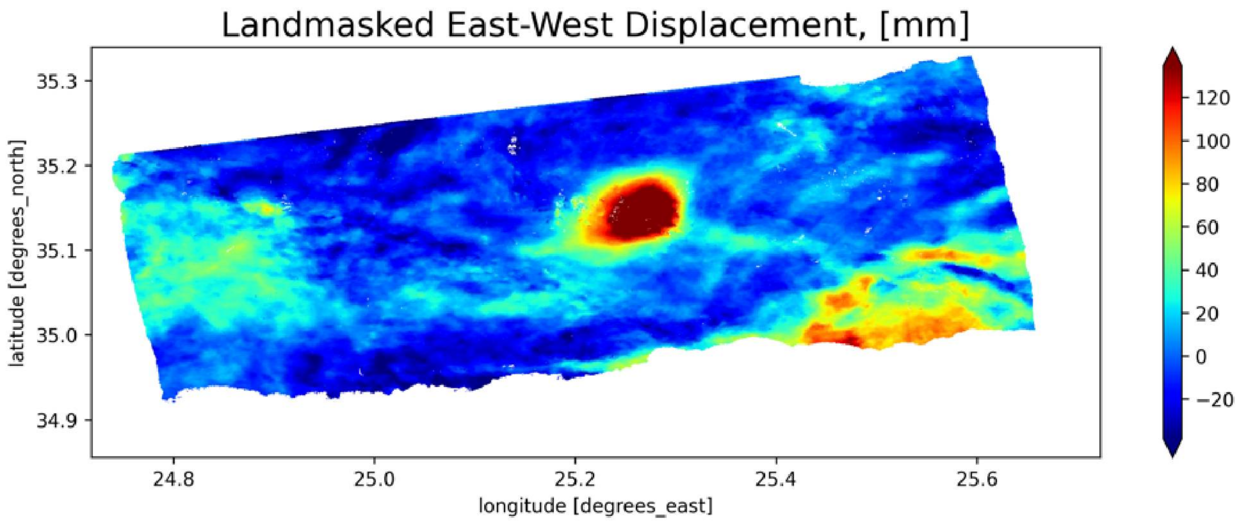
The first image showcases an interferogram of the 2021 Crete Earthquake. The fringes represent phase changes between the before and after images of the event, which correspond to the ground deformations caused by the earthquake. Each fringe cycle corresponds to a half-wavelength change in the distance between the satellite and the ground surface, translating to a relative ground movement.

The land mask applied to this image enhances the visibility of these fringes by eliminating the noise produced by the ocean. Offshore areas tend to exhibit more noise due to water motion and low radar reflectivity.

The epicenter of the earthquake can be visually pinpointed from the concentric fringes originating from a common center. The varying fringe patterns surrounding the epicenter indicate the diverse ground displacements caused by the seismic activity.



The second image presents the east-west component of the displacement caused by the earthquake, computed from the incidence angle map and the unwrapped phase of the interferogram. Again, a land mask is applied to remove offshore noise.



The 2023-02-06 Türkiye Earthquake Co-Seismic Interferogram notebook demonstrates the application of InSAR to process data from one of the most catastrophic seismic events in recent history. On 6 February 2023, a magnitude 7.8 earthquake impacted southern and central Turkey and northern and western Syria, causing widespread destruction.

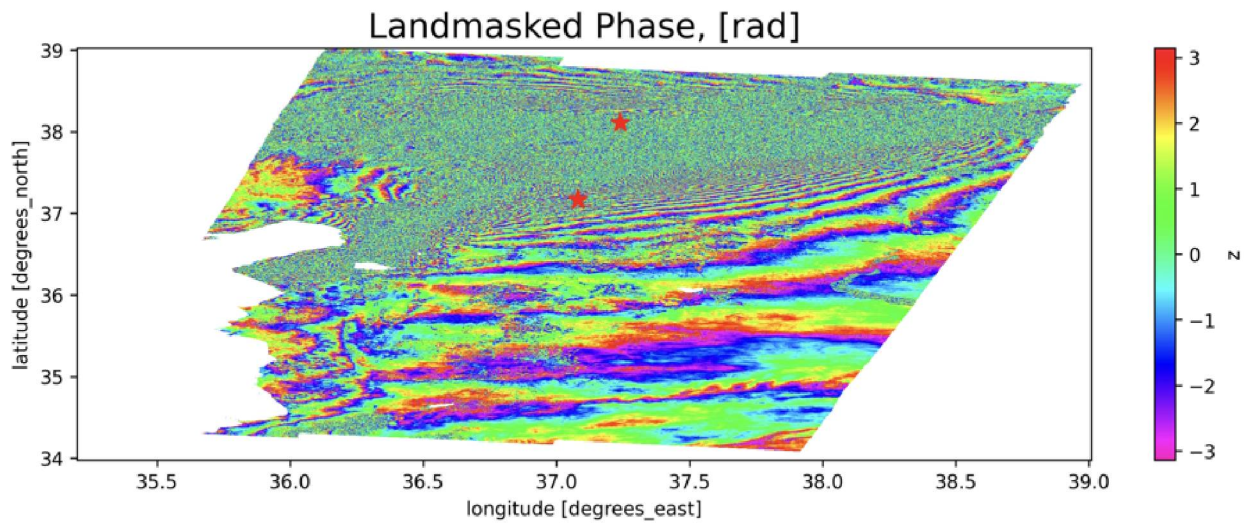
The example makes use of InSAR techniques to process Sentinel-1 Scenes, obtained from the Alaska Satellite Facility (ASF). This processing involves stitching 3 scenes, merging subswaths, detrending phase, and performing lazy exporting of NetCDF rasters (compatible with QGIS, GDAL, and other GIS software).

This case illustrates some tricks to process a large volume of data on Google Colab effectively. Note that you can replace the scene names in this notebook to produce an interferogram for your specific area of interest.

The first image displays an interferogram derived from the Sentinel-1 scenes of the 2023-02-06 Türkiye Earthquake. The colorful fringes represent phase changes between the pre- and post-seismic event, illustrating the ground deformations caused by the earthquake.

This interferogram covers a large area, which corresponds to the substantial extent of the earthquake's impact. The heterogeneous pattern of the fringes might suggest various types of ground movements such as landslides and surface ruptures associated with the seismic event. These different movements can be further studied for a comprehensive understanding of the event.

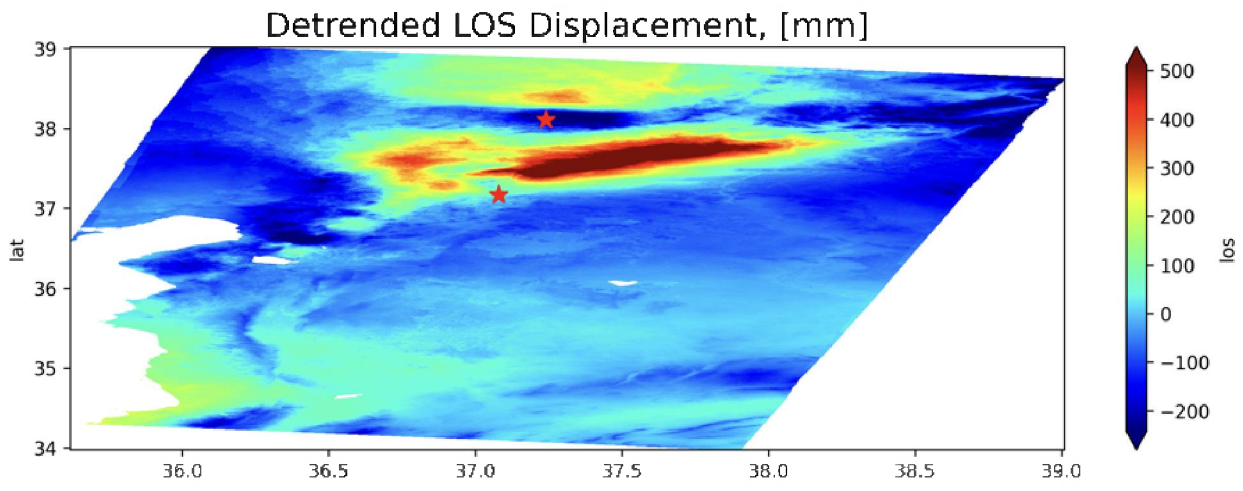
A land mask is applied to this interferogram to eliminate noise, particularly in the lower left corner where a water body is located.



The second image showcases the Line-Of-Sight (LOS) displacement map of the same event, after detrending the phase. The detrending process aims to remove the linear ramp present in the original phase map, which can distort the final displacement results.

Despite the noisiness of the original interferogram, the phase unwrapping tool SNAPHU manages to unwrap the phase, providing a continuous displacement pattern. The result may have some inaccuracies due to unrecognized fringes, but the overall displacement pattern gives a reasonable approximation of the ground movements during the earthquake.

The colors in the map represent the magnitude and direction of displacement along the satellite's line of sight, providing a clearer visualization of the ground deformations caused by the earthquake.



Volcanology

InSAR is a useful tool in volcanology. It uses the phase differences in radar waves bounced back from the Earth's surface to construct detailed maps of surface deformation with high spatial resolution and precision. This technology is especially valuable for monitoring volcanoes, as it can cover sizeable areas and work in all weather conditions, day or night.

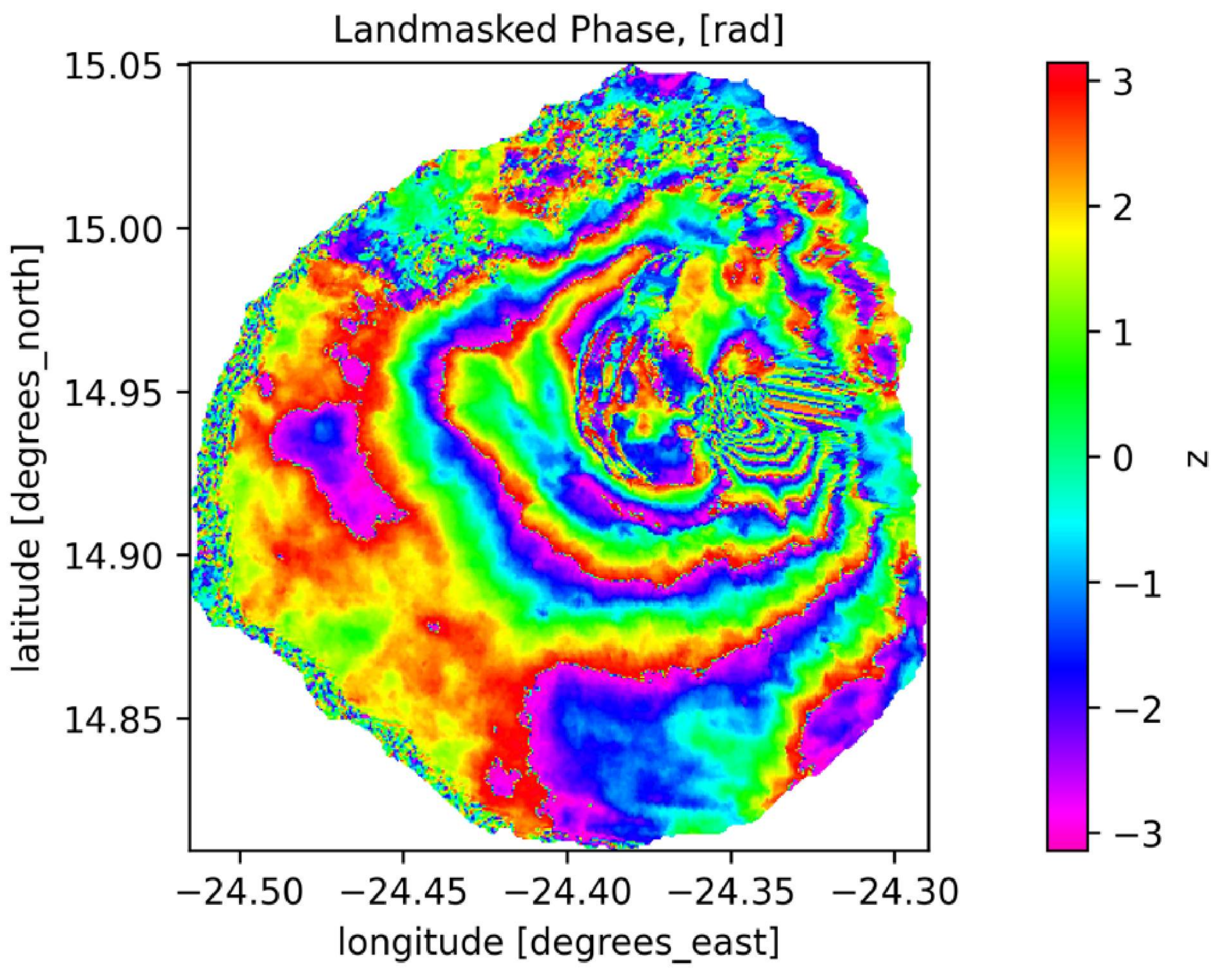
The ability of InSAR to detect minute changes in the Earth's surface (down to a fraction of the wavelength of the radar wave, typically a few centimeters) makes it particularly effective for monitoring volcanic activity. Before an eruption, magma rising towards the surface can cause the ground to swell. This inflation can be picked up by InSAR, providing a warning sign of a possible impending eruption.

Similarly, after an eruption, the ground can deflate as magma chambers are empty. Again, InSAR can detect these changes, providing valuable data on the volume of magma involved and the mechanics of the eruption. This kind of information can help scientists better understand a volcano's behavior and improve eruption forecasting.

This notebook explores the eruption of the Pico do Fogo volcano on Fogo Island in Cape Verde, which occurred on November 23, 2014. The study uses InSAR data to investigate the geophysical changes related to this volcanic event, offering insights into the eruption dynamics and the subsequent impacts on the local environment.

In this example, a single cropped subswath is processed with a land mask applied to the interferogram. The output includes the interferogram, unwrapped phase, and Line-Of-Sight (LOS), east-west, and vertical displacement maps. These results give an overview of the earth's deformation due to the eruption.

The image is a phase map showing the deformation of the Pico do Fogo volcano due to an eruption. The land mask filters out the water bodies. The fringes indicate surface deformation due to the volcanic activity and the eruption. In this case, it seems the whole island experienced some level of deformation due to the eruption, which would have been caused by the movement of magma beneath the surface.



Hydrology

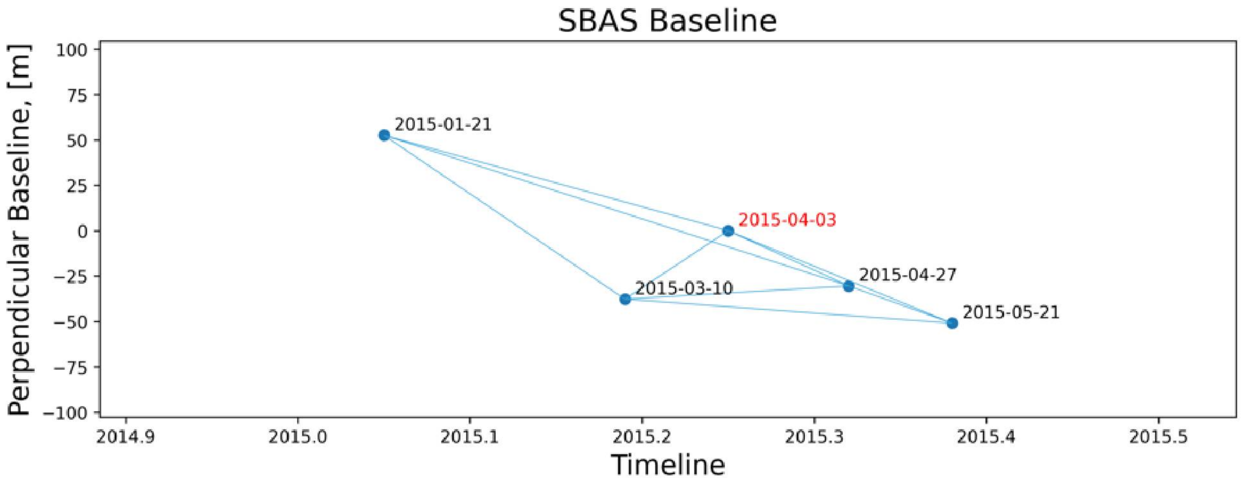
InSAR is a powerful tool that can monitor changes in water levels and subsurface movements. This technology has been successfully used in various hydrological studies, including tracking the rate of groundwater extraction, mapping wetland water levels, and observing glacier movements.

This notebook guides you through the entire process of using PyGMTSAR to analyze changes in water levels in the Imperial Valley using Small BAseline Subset (SBAS).

In this example, you will learn how to implement the SBAS approach, a powerful InSAR technique specifically designed to detect slow ground deformation over time. The SBAS approach involves selecting a subset of interferograms with small temporal and spatial baselines, which effectively reduces the impact of decorrelation and atmospheric delays on the results.

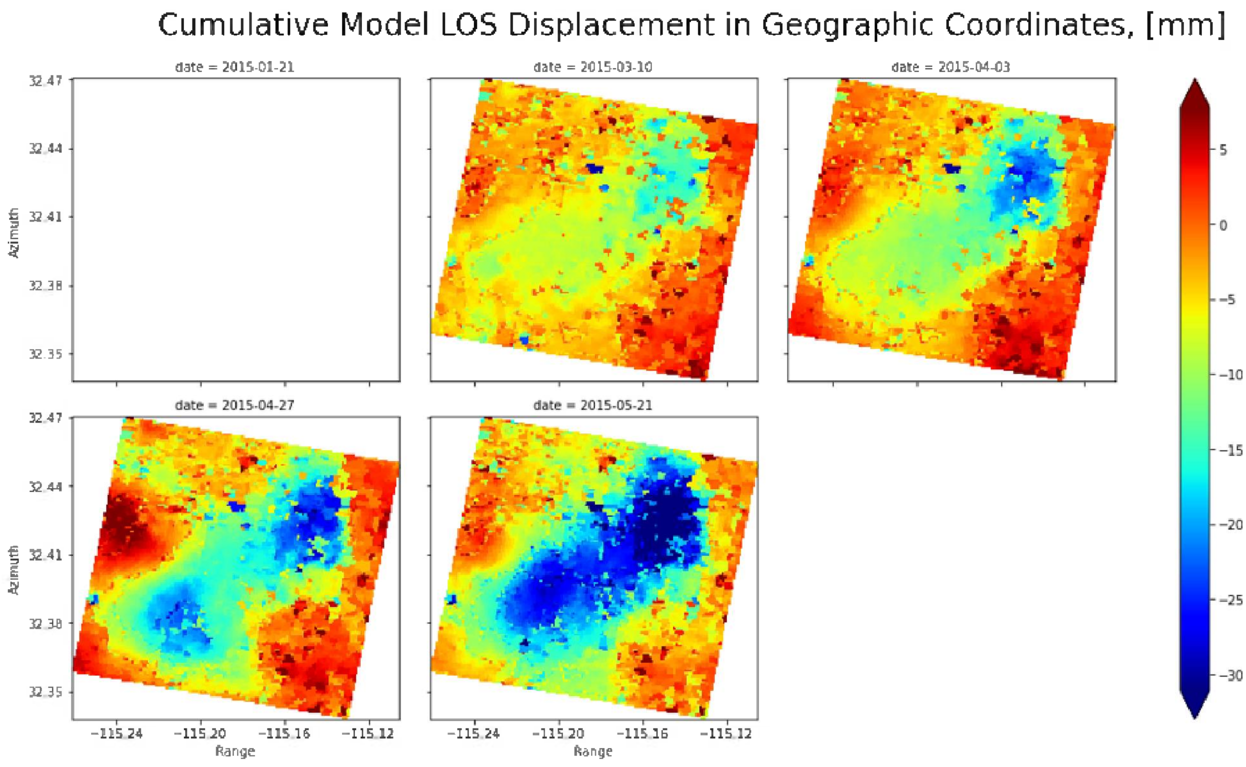
And the example introduces you to the detrending technique used in PyGMTSAR to remove atmospheric noise, thus significantly improving the quality of the resulting interferograms and making the observed water level changes more accurate and reliable.

The first image is a baseline chart for the SBAS InSAR technique. It shows all the interferograms that were used in the analysis, arranged by their acquisition dates (X-axis) and the perpendicular baseline (Y-axis). Each point on the plot represents an individual Sentinel-1 scene, and each edge represents an interferogram.



The second image illustrates the cumulative Line of Sight (LOS) displacements over time in the Imperial Valley. This map of displacement provides a clear visualization of the relative movement of the ground surface due to changes in water levels.

In regions where groundwater is extracted, we often observe a seasonal pattern in subsidence and uplift - subsidence typically occurs during the dry months when more water is pumped out, and uplift occurs during the wet months when aquifers are replenished. Hence, the map not only provides valuable information on the spatial distribution of subsidence/uplift but also potentially reveals insights about the region’s groundwater dynamics.



Cumulative Model LOS Displacement in Geographic Coordinates, [mm]
date = 2015-01-21
date = 2015-03-10
date = 2015-04-03
date = 2015-04-27
date = 2015-05-21
Azimuth
Range
32.47
32.44
32.41
32.38
32.35
−115.24
−115.20
−115.16
−115.12
5
0
−5
−10
−15
−20
−25
−30

Infrastructure Monitoring

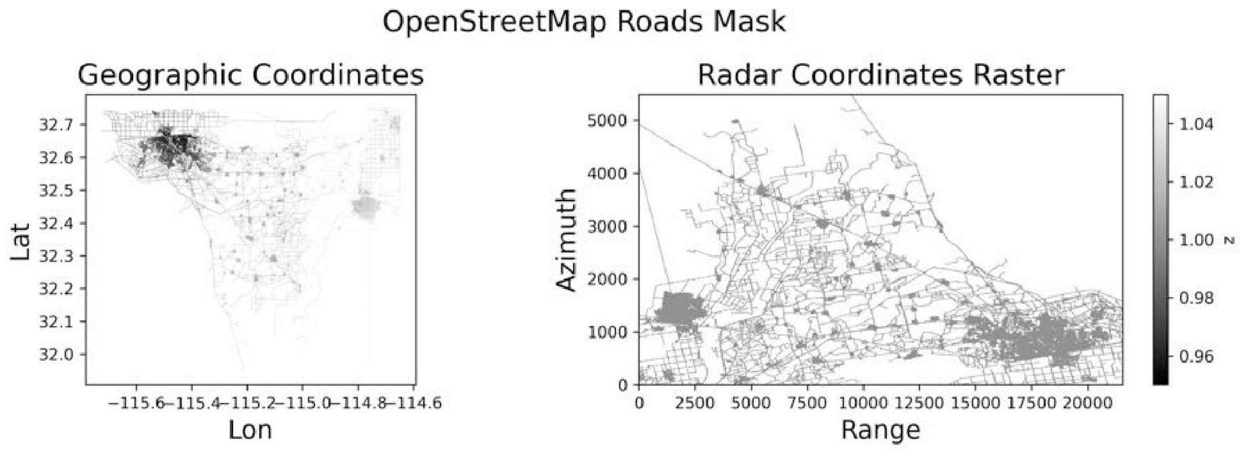
InSAR is used to monitor infrastructure like dams, bridges, and buildings. It can detect subtle movements and deformations that might suggest potential structural problems or failures.

For now, PyGMTSAR applies Goldstein and Gaussian filters to the processed interferograms, making the results more smooth and highlighting fridges. That makes the outputs more suitable for the kinds of analysis explained above. Be careful with spatial accuracy for small infrastructure objects monitoring. When monitoring infrastructure, accurate identification and assessment of subtle changes can mean the difference between timely intervention and significant structural failure.

This notebook presents an analysis of OpenStreetMap’s road infrastructure to monitor road subsidence. That’s just an example and the most subsidence are related to seasonal water level changes for the area.

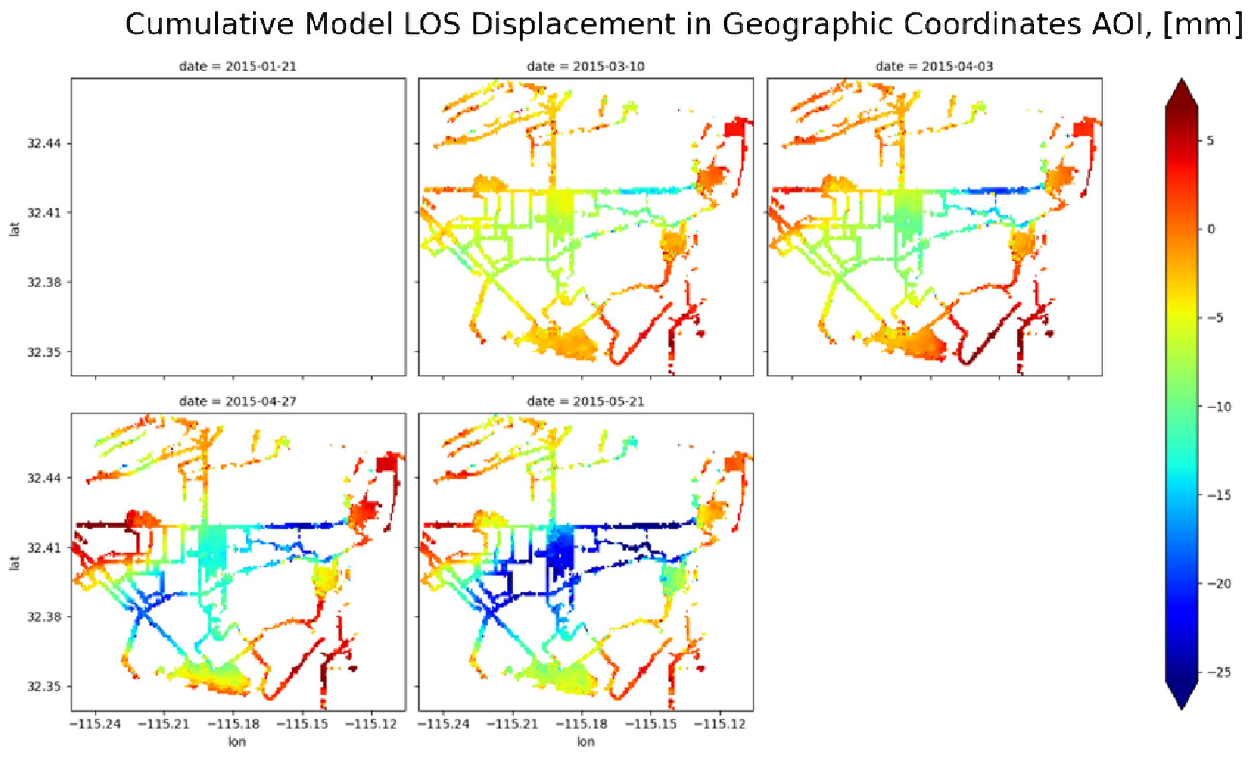
The first image portrays a vector and a rasterized representations of road infrastructure derived from OpenStreetMap (OSM) data. These roads are converted from their original geographic coordinates into radar coordinates, to allow their alignment with the radar-based interferometric data.

Such a road mask can be crucial in studies looking to identify ground movement affecting infrastructure, such as roads. It enables the extraction of InSAR data specific to these regions, allowing for detailed analysis of any detected ground deformation.



The second image presents a cumulative Line-Of-Sight (LOS) displacement map. Here, the displacements observed across multiple radar acquisitions are summed to illustrate the total changes over time. These shifts could be due to a variety of factors, with seasonal fluctuations in water levels being a common cause in this region.

The roads, previously identified and masked from the OSM data, are included on this map. This enables an investigation into how changes in ground movement, possibly due to fluctuating water levels, impact the stability and integrity of road infrastructure. With a long enough time series of data, this analysis can detect not only common seasonal changes but also trends in road movement that may indicate potential issues or the overall health state of the infrastructure.


Cumulative Model LOS Displacement in Geographic Coordinates AOI, [mm]
date = 2015-01-21
date = 2015-03-10
date = 2015-04-03
date = 2015-04-27
date = 2015-05-21
lat
lon
32.44
32.41
32.38
32.35
−115.24
−115.21
−115.18
−115.15
−115.12
5
0
−5
−10
−15
−20
−25

2.2. Running Locally with Docker Desktop

Docker is an open source platform that uses OS-level virtualization to deliver software in packages called containers. A Docker container is a standalone, executable package that includes everything needed to run an application: the code, a runtime, libraries, environment variables, and config files.

If you prefer to work offline or locally, or if your project necessitates a more powerful setup than what's available on Google Colab, you can make use of Docker. The author of PyGMTSAR has prepared a Docker image with the same examples as those on Google Colab. This can be found on the DockerHub

Here's why consider Docker:

You can avoid the common software issue where an application works on one machine but not on another because of differences in their environment. Containers with different PyGMTSAR versions can be run on the same host even simultaneously and can be transferred between different computers and operation systems. Define exactly available processor cores, memory, and disk space.

Docker ensures consistency across multiple computers and operation systems. The ready-to-use PyGMTSAR images include reproducible examples which work equally on any host. Use the stable and well tested

PyGMTSAR images and containers or make your own ones and do not worry about portability.

Docker provides control over your environments and allows configurations to be versioned and reused. Operate the stable and well tested PyGMTSAR images, which can be rebuilt and reused locally or from DockerHub.

To get PyGMTSAR running on your computer, you will need to download the PyGMTSAR Docker image and run it using Docker Desktop. Docker Desktop is a user-friendly and open source interface for Docker, providing a GUI and additional features for managing containers. If you do not have Docker Desktop already installed, the following steps will guide you through the installation process. Note, however, that you could use the lower-level Docker tool as an alternative to Docker Desktop, though it may be more challenging to navigate.

Installing Docker Desktop

First, you need to download and install Docker Desktop on your operating system. Just follow these simple guides:

For Windows users: Install Docker Desktop on
For Linux users: Install Docker Desktop on
For Mac users: Install Docker Desktop on

How to Register on DockerHub

Before you can explore the PyGMTSAR collection of container images on DockerHub, you'll need to set up an account. DockerHub is a cloud-based platform where Docker images can be found, shared, and managed. It acts as a connection point between your local Docker installation and Docker's public repository.

Docker Desktop offers various methods to download prepackaged software images. You can use console commands provided on DockerHub software pages if you're an advanced user, or use the Docker Desktop's user-friendly interface.

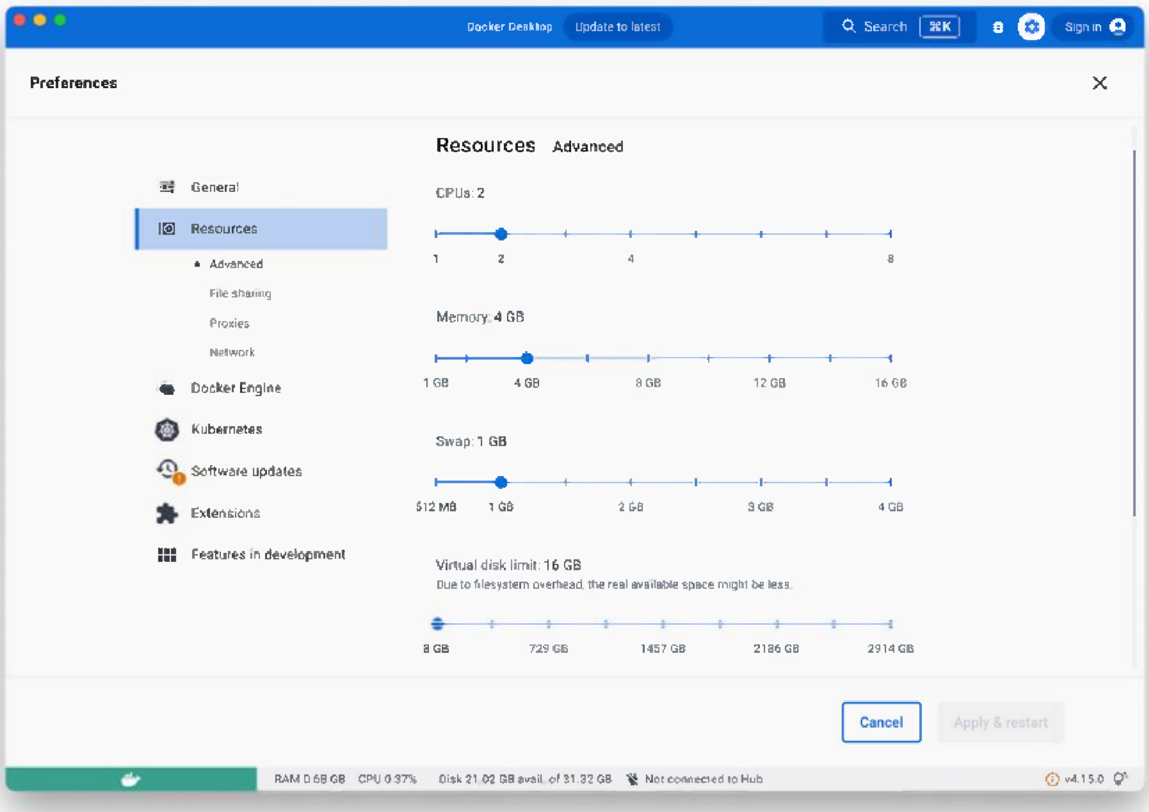
To register on DockerHub through Docker Desktop, just click "Sign in" in the top right corner of the application window. This action will direct you to a sign-in page where you can create a new account.

If you'd rather register directly on a web browser, follow this link: Create a Docker Keep your Docker ID and password handy; you'll need them to download and manage Docker images.

Having a DockerHub account enables you to download and use any public images, create and manage your own Docker images, and even share them with others.

How to Configure Docker Desktop

To access Docker Desktop settings, click on the gear icon at the top right corner and navigate to the “Resources” section:



CPUs: 2

Memory: 4 GB

Swap: 1 GB

Virtual disk limit: 16 GB
Due to filesystem overhead, the real available space might be less.

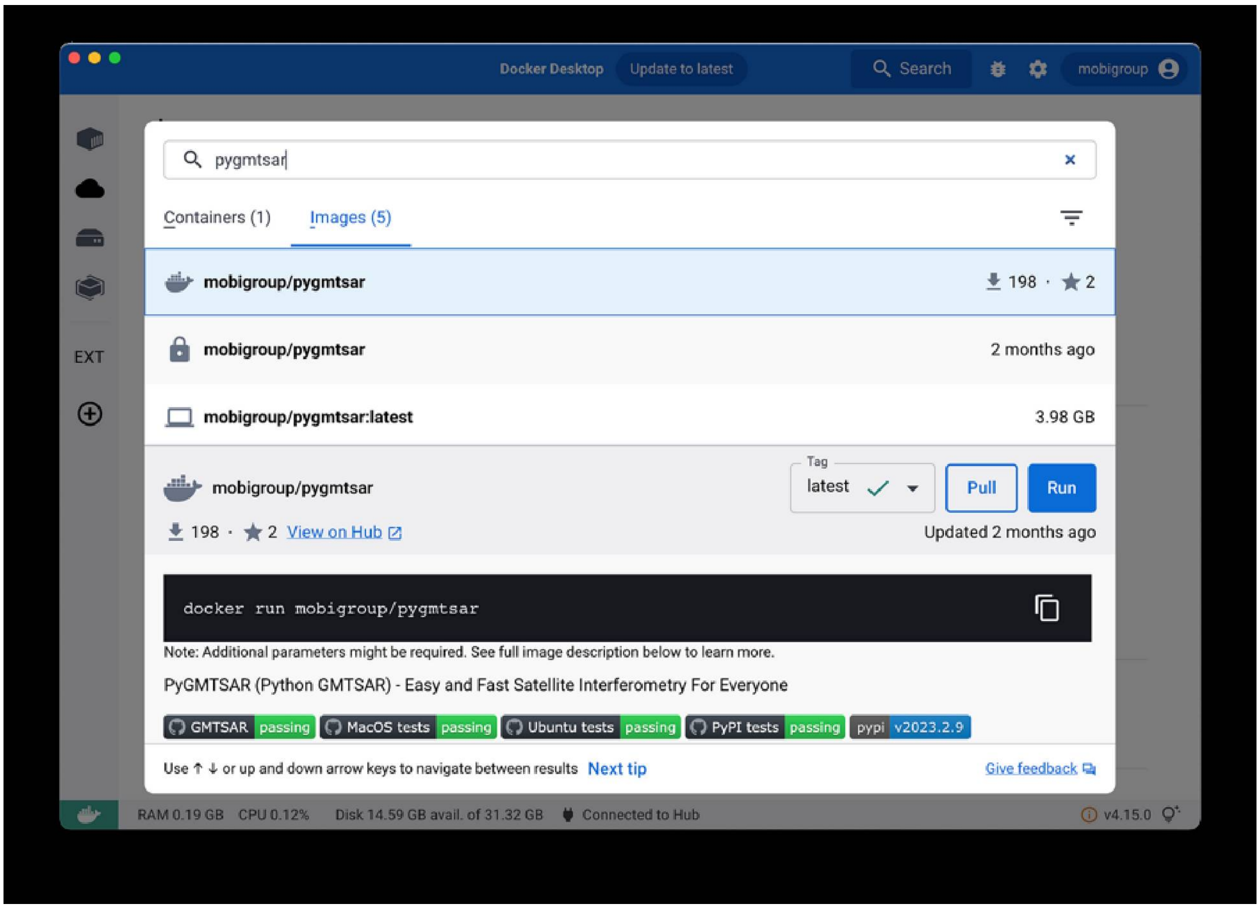
RAM 0.68 GB CPU 0.37% Disk 21.02 GB avail. of 31.32 GB Not connected to Hub v4.15.0

If you plan to run examples from the mobigroup/pygmtsar image, you’ll need to allocate 2 CPU cores, 4 GB RAM, 1 GB swap, and a virtual disk of 16+ GB. If you intend to execute all the notebooks sequentially, set the virtual disk limit to 200+ GB. Remember, satellite interferometry processing can take up significant storage!

If you aim to launch the larger mobigroup/pygmtsar-large image, your virtual system should have 4 CPU cores, 16 GB RAM, 1 GB swap, and a 500 GB virtual disk. This configuration allows you to produce 34 interferograms, each of 3 GB size, along with their corresponding correlation files. It also enables detrending and SBAS analysis. Even a task this heavy can be performed on an Apple Silicon iMac or Air with 16 GB RAM, as Docker Desktop can utilize all available RAM.

How to Download the PyGMTSAR Docker Image

Once you've registered and logged in, look for the "pygmtsar" in the search box located in the "Images" tab:



```
docker run mobigroup/pygmtsar
```

Note: Additional parameters might be required. See full image description below to learn more.

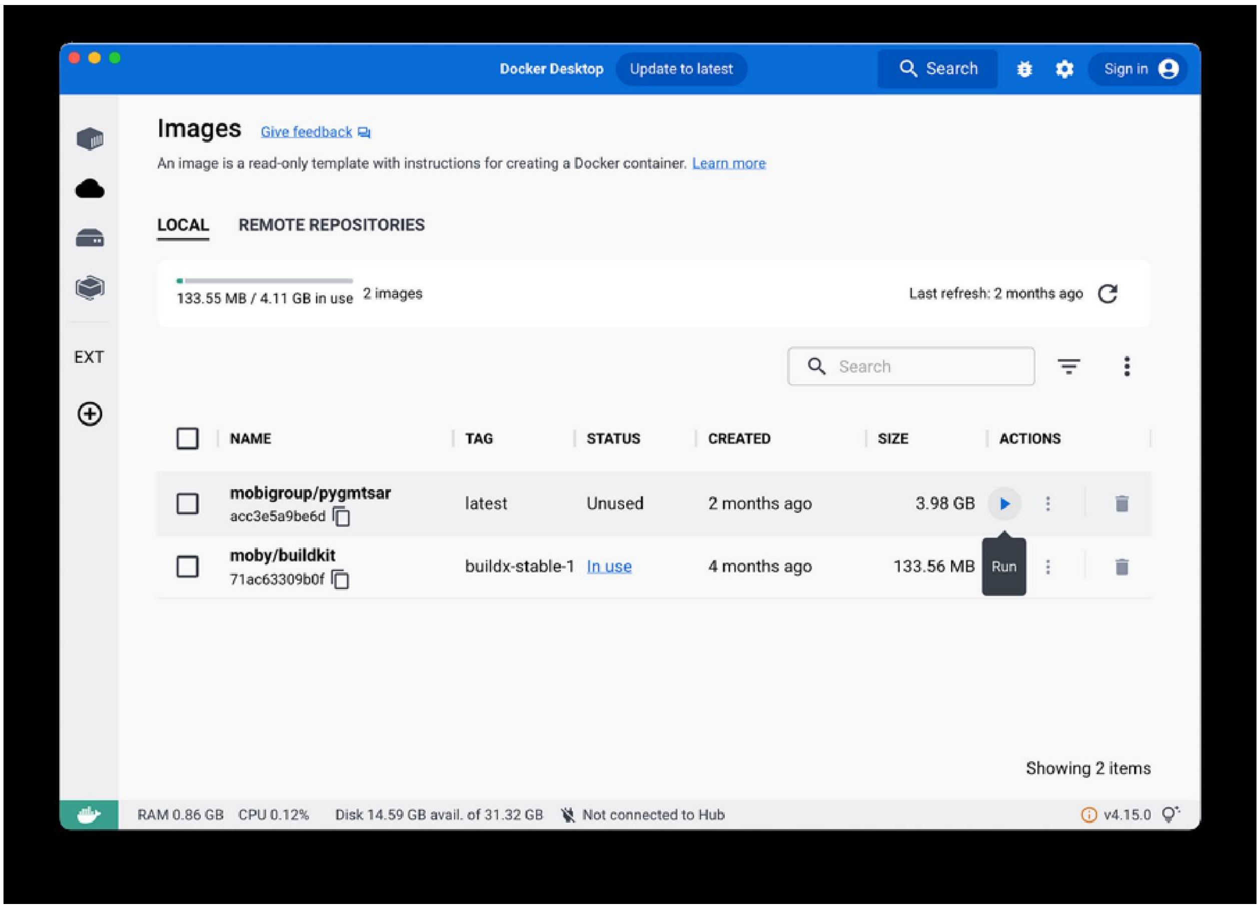
PyGMTSAR (Python GMTSAR) - Easy and Fast Satellite Interferometry For Everyone

Use ↑ ↓ or up and down arrow keys to navigate between results

Two options will appear: mobigroup/pygmtsar is a standard 1 GB PyGMTSAR image that meets the requirements of most users, and mobigroup/pygmtsar-large is a more advanced 50 GB image designed for expert users. Choose the one that fits your needs and click on the "Pull" button to download it onto your computer.

Steps to Run the PyGMTSAR Docker Image

In Docker Desktop, the “Images” section contains all the software images you’ve downloaded. You can manage these images from here. To run the PyGMTSAR image you downloaded, click the triangular button, as shown in the image below.



Images Give feedback

An image is a read-only template with instructions for creating a Docker container. Learn more

133.55 MB / 4.11 GB in use 2 images

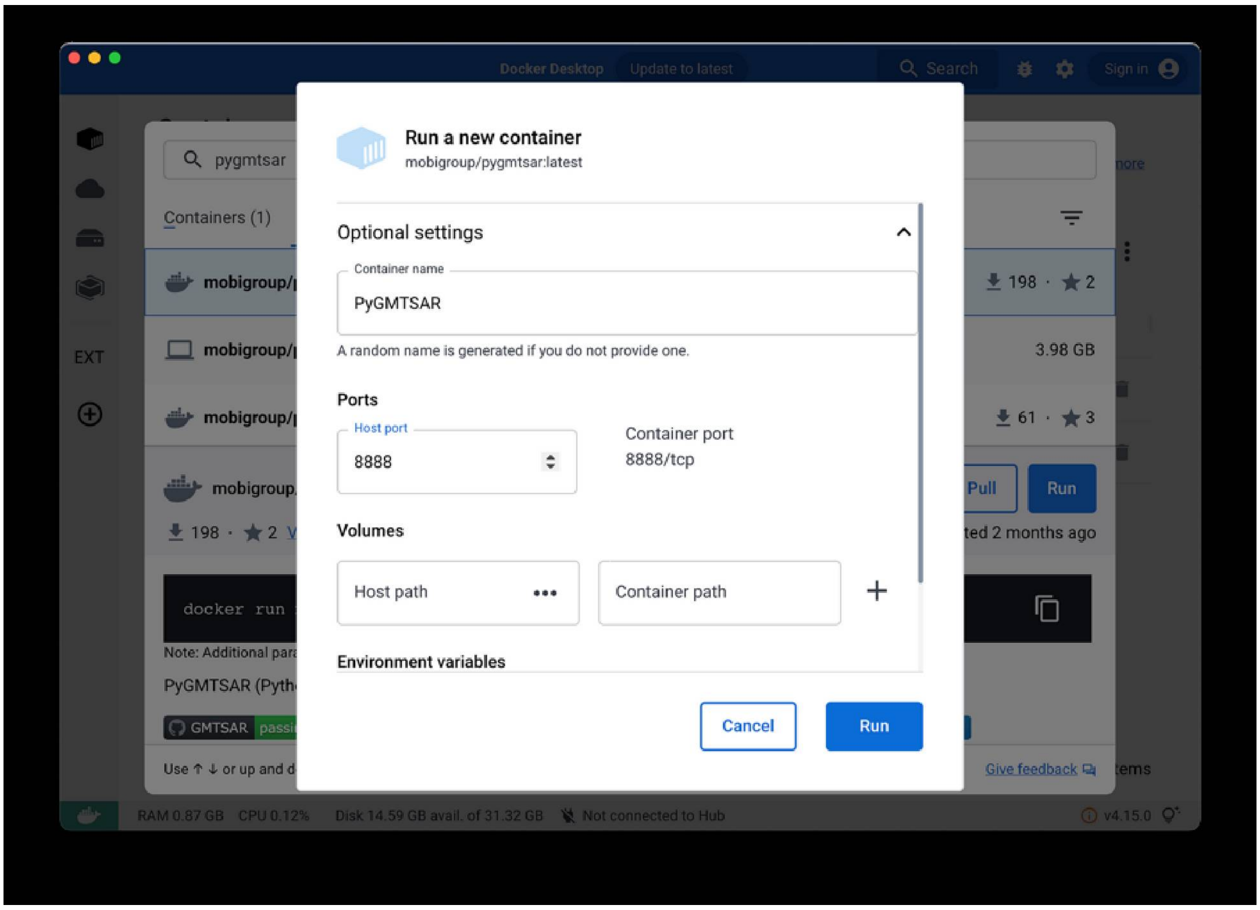
Last refresh: 2 months ago

NAME	TAG	STATUS	CREATED	SIZE	ACTIONS
mobigroup/pygmtsar acc3e5a9be6d	latest	Unused	2 months ago	3.98 GB	
moby/buildkit 71ac63309b0f	buildx-stable-1	In use	4 months ago	133.56 MB	

Showing 2 items

RAM 0.86 GB CPU 0.12% Disk 14.59 GB avail. of 31.32 GB Not connected to Hub v4.15.0

PyGMTSAR will connect with you through a webpage in your browser, utilizing network port 8888. Set it as shown in the image below and, if you want, give it a name, such as “PyGMTSAR”:


Run a new container
mobigroup/pygmtsar:latest

Optional settings

Container name: PyGMTSAR

A random name is generated if you do not provide one.

Ports

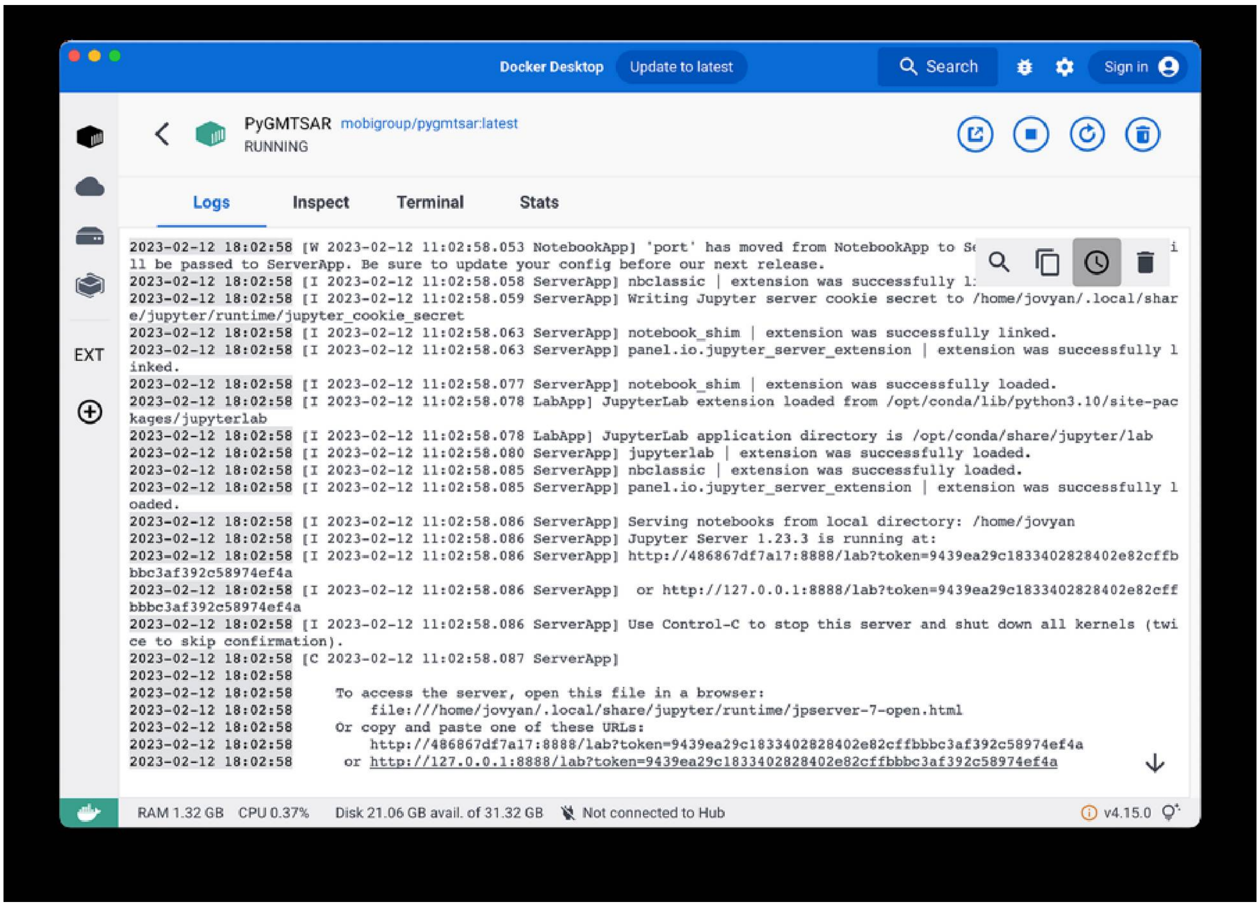
Host port: 8888
Container port: 8888/tcp

Volumes

Host path
Container path

Environment variables

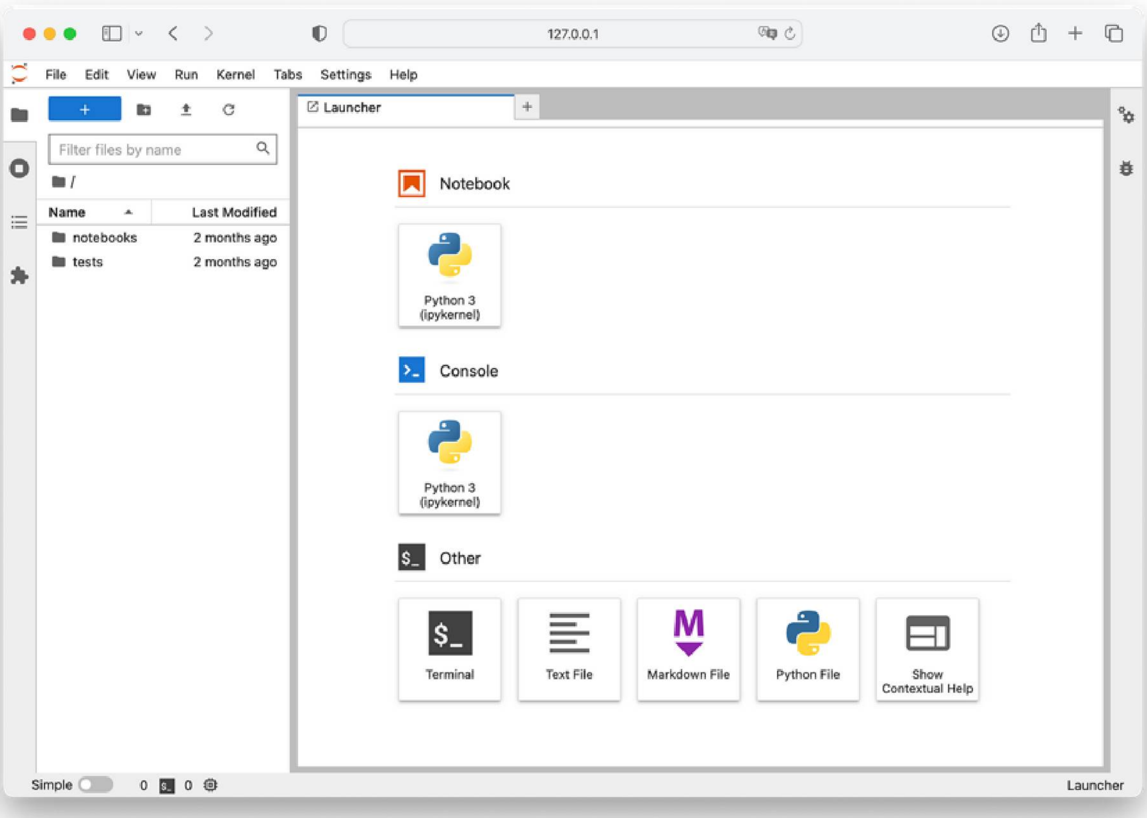
After setting it up, click the “Run” button in the form above to start the software:



```
2023-02-12 18:02:58 [W 2023-02-12 11:02:58.053 NotebookApp] 'port' has moved from NotebookApp to Se                                  i
ll be passed to ServerApp. Be sure to update your config before our next release.
2023-02-12 18:02:58 [I 2023-02-12 11:02:58.058 ServerApp] nbclassic | extension was successfully l:
2023-02-12 18:02:58 [I 2023-02-12 11:02:58.059 ServerApp] Writing Jupyter server cookie secret to /home/jovyan/.local/shar
e/jupyter/runtime/jupyter_cookie_secret
2023-02-12 18:02:58 [I 2023-02-12 11:02:58.063 ServerApp] notebook_shim | extension was successfully linked.
2023-02-12 18:02:58 [I 2023-02-12 11:02:58.063 ServerApp] panel.io.jupyter_server_extension | extension was successfully l
inked.
2023-02-12 18:02:58 [I 2023-02-12 11:02:58.077 ServerApp] notebook_shim | extension was successfully loaded.
2023-02-12 18:02:58 [I 2023-02-12 11:02:58.078 LabApp] JupyterLab extension loaded from /opt/conda/lib/python3.10/site-pac
kages/jupyterlab
2023-02-12 18:02:58 [I 2023-02-12 11:02:58.078 LabApp] JupyterLab application directory is /opt/conda/share/jupyter/lab
2023-02-12 18:02:58 [I 2023-02-12 11:02:58.080 ServerApp] jupyterlab | extension was successfully loaded.
2023-02-12 18:02:58 [I 2023-02-12 11:02:58.085 ServerApp] nbclassic | extension was successfully loaded.
2023-02-12 18:02:58 [I 2023-02-12 11:02:58.085 ServerApp] panel.io.jupyter_server_extension | extension was successfully l
oaded.
2023-02-12 18:02:58 [I 2023-02-12 11:02:58.086 ServerApp] Serving notebooks from local directory: /home/jovyan
2023-02-12 18:02:58 [I 2023-02-12 11:02:58.086 ServerApp] Jupyter Server 1.23.3 is running at:
2023-02-12 18:02:58 [I 2023-02-12 11:02:58.086 ServerApp] http://486867df7a17:8888/lab?token=9439ea29c1833402828402e82cffb
bbc3af392c58974ef4a
2023-02-12 18:02:58 [I 2023-02-12 11:02:58.086 ServerApp]  or http://127.0.0.1:8888/lab?token=9439ea29c1833402828402e82cff
bbbc3af392c58974ef4a
2023-02-12 18:02:58 [I 2023-02-12 11:02:58.086 ServerApp] Use Control-C to stop this server and shut down all kernels (twi
ce to skip confirmation).
2023-02-12 18:02:58 [C 2023-02-12 11:02:58.087 ServerApp]
2023-02-12 18:02:58
2023-02-12 18:02:58     To access the server, open this file in a browser:
2023-02-12 18:02:58         file:///home/jovyan/.local/share/jupyter/runtime/jpserver-7-open.html
2023-02-12 18:02:58     Or copy and paste one of these URLs:
2023-02-12 18:02:58         http://486867df7a17:8888/lab?token=9439ea29c1833402828402e82cffbbbc3af392c58974ef4a
2023-02-12 18:02:58      or http://127.0.0.1:8888/lab?token=9439ea29c1833402828402e82cffbbbc3af392c58974ef4a
```

RAM 1.32 GB CPU 0.37% Disk 21.06 GB avail. of 31.32 GB Not connected to Hub
v4.15.0

Next, click on the bottom link in the "Logs" page to open a new webpage and begin working:



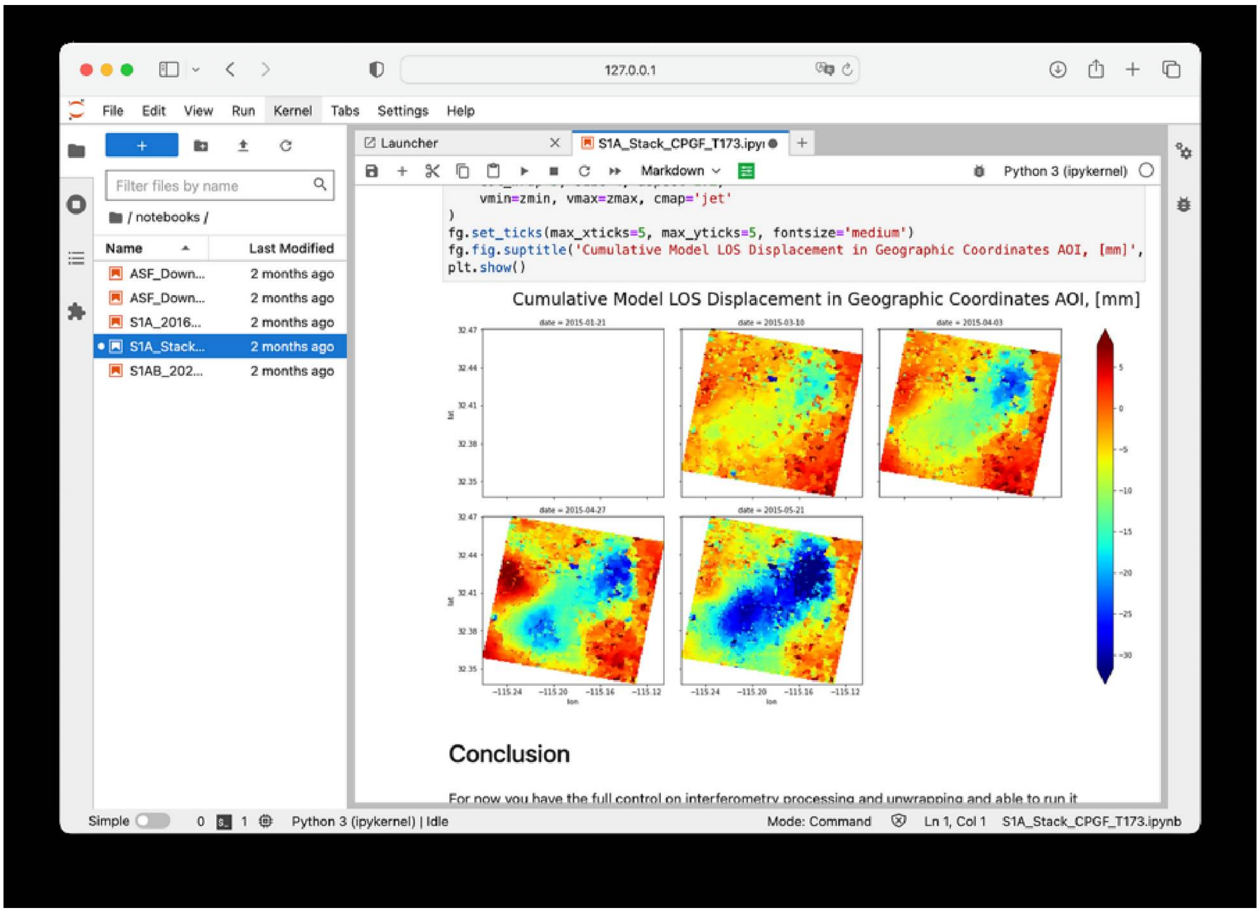
Name	Last Modified
notebooks	2 months ago
tests	2 months ago

The interface presents an interactive workspace: the left panel shows all the files and directories in your Docker image, while the right panel acts as an interactive editor, viewer, and console, among other functions. There are two key directories, “notebooks” and “tests”.

“notebooks” contains interactive Jupyter notebook examples that guide you through various PyGMTSAR processes, allowing you to interact with the data and results in real time.

In contrast, “tests” contains Python script examples for those who prefer using console scripts. These scripts work in batch mode and produce the same visual outputs as the interactive ones. Importantly, these scripts are used for continuous integration testing in the PyGMTSAR GitHub repository. You can access the outputs of these tests on the PyGMTSAR GitHub Actions page.

To get started, click on an example in the left panel. You can navigate through the example by scrolling, reading the instructions, and viewing the maps:



```
    vmin=zmin, vmax=zmax, cmap='jet'
)
fg.set_ticks(max_xticks=5, max_yticks=5, fontsize='medium')
fg.fig.suptitle('Cumulative Model LOS Displacement in Geographic Coordinates AOI, [mm]',
plt.show()
```

Conclusion

For now you have the full control on interferometry processing and unwrapping and able to run it

To run the whole example, select “Kernel” -> “Restart Kernel and Run All Cells…” from the toolbar menu (Note: the exact menu item may vary depending on the image version).

This interface allows you to monitor the execution steps, pause and resume processing as needed, alter parameters, and view the resulting outputs. It provides the same functionality as the Google Colab live online examples, but without third-party timeouts and processing limits. This makes Docker Desktop a robust tool for comprehensive analysis and research.

3. Exploring PyGMTSAR

Chapter 3 unveils the principles and functionality of PyGMTSAR, a powerful tool designed for efficient processing of interferometric synthetic aperture radar (InSAR) data. This section serves as a comprehensive guide for using this software, shedding light on its operational aspects without delving too deep into the intricacies of its core algorithms.

In Section 3.1, we present the operational facets and the architecture of PyGMTSAR. This Python library is versatile, designed to operate across a variety of hardware, from standard laptops to high-performance workstations. We explore how PyGMTSAR streamlines processing tasks, making execution as simple as a single click, even in the absence of local software installation. This feature is powered by its compatibility with multiple environments, including interactive Jupyter notebooks on Google Colab, Docker containers, and GitHub Action runners. Underpinning this scalability is Dask, a distributed process manager, which optimizes the execution of PyGMTSAR operations. Dask segments large operations into manageable tasks and executes them in an optimal order, efficiently leveraging all available resources.

In Section 3.2, we delve into the principle of lazy and delayed numeric computations that PyGMTSAR implements. This strategy allows operations to be deferred, enabling quick access to input and output rasters. We provide a detailed explanation of how to initialize the Dask parallel and distributed scheduler, a critical component for managing parallel processing tasks. Additionally, we cover on-the-fly transformations, such as geocoding, and discuss the trade-off between memory consumption and processing speed. To monitor and understand the software's internal parallel processing, we introduce the Dask Dashboard and the PyGMTSAR progress indicator. These tools offer valuable insights into your code's performance, helping identify potential bottlenecks or areas for improvement and ensuring that computations are efficient and resource-effective.

In Section 3.3, the critical role of the core SBAS object that PyGMTSAR manipulates during processing is highlighted. This object is central to PyGMTSAR's functionality, acting as the primary data structure with which the software interacts. The SBAS object serves a dual purpose: it stores essential data and enables effective management and processing of this information.

In Section 3.4, we explore the internal operations on NetCDF grids, focusing on PyGMTSAR's capabilities to read and write in NetCDF and GeoTIFF, and to export data into VTK files. Given that InSAR time-series analysis deals with large datasets and generates equally large outputs, a deep understanding of PyGMTSAR's data operation features becomes crucial for efficient data processing.

Section 3.5 focuses on the essential steps of InSAR processing that PyGMTSAR executes. From data acquisition to reframing, image co-registration, interferogram formation, geocoding, phase unwrapping, phase detrending, displacement map creation, displacement projection, time-series analysis, trend analysis, and data exporting, each step is thoroughly explored. This section also introduces the machine learning algorithms integrated into PyGMTSAR that simplify these processes for users. Leveraging advanced techniques, PyGMTSAR provides not just an efficient, but also a robust and high-quality InSAR processing pipeline.

The principles and functioning of PyGMTSAR covered in this chapter form a robust foundation for understanding the examples provided in Google Colab and in Docker images. By following these principles and keeping the operation of delayed computation in mind, you can effectively use all the InSAR processing steps.

3.1. Understanding PyGMTSAR

PyGMTSAR is a Python library designed to analyze a vast amount of satellite interferometry data across a broad range of hardware. Its core concept is to facilitate the reproducible processing of interferometry tasks with a single click, even without the need for local software installation. This is achievable in an interactive Jupyter notebook environment on Google Colab, in Docker containers, and as console scripts on more limited environments like GitHub Action runners. Simultaneously, PyGMTSAR enables the execution of the same notebooks and scripts on more powerful computers, utilizing all available processing power. This multifaceted approach likely requires further explanation.

While Python is renowned as a popular programming language, it is often considered slow for numerical calculations. However, it is exceptionally adept at building scalable big data processing systems. A crucial component of distributed high-performance computations is a cluster resource manager. This type of software is installed on any Linux/Unix cluster and supercomputer to perform ordered execution of a set of small dependent tasks effectively. The fundamental idea is pretty simple — break down large operations into manageable chunks, construct a dependencies graph, and run enough processes in the right order to utilize all processors while ensuring sufficient memory for all calculations.

PyGMTSAR utilizes a distributed process manager, similar to how large computer clusters use systems like Slurm: cluster management and job scheduling Fortunately, Dask offers more flexibility than traditional Unix/Linux resource managers and can be run on a variety of platforms. This includes Google Colab, within a Docker container, on a local laptop, a high-performance workstation, or even on a computer cluster. Python boasts remarkable flexibility and speed compared to shell scripts used on Unix/Linux clusters to manage tasks. Moreover, Dask excels in autoscaling, enabling the full utilization of all processors for nearly any amount of available memory. In other words, it works effectively on any hardware, organizing the processing appropriately.

All these advantages of Dask can be realized if the software can work on distributed data chunks of variable size while using predictable memory. PyGMTSAR was designed to comply with these requirements. Although it can run effectively on Linux/Unix cluster managers, it achieves the best results with Dask — zero administration automated scalable processing on any hardware.

Native PyGMTSAR processing functions based on well-known scientific libraries to process the huge amount of data effectively. Low-level operations in scientific Python libraries are mostly coded on compiled languages C and Fortran and these are very scalable and powerful. Python language is glue for the optimized and well-tuned core blocks for multidimensional array manipulation in Numpy, machine learning algorithms in Scipy and so on. PyGMTSAR Python code never operates by a single pixel but always calls the basic operations on data blocks (chunks) like 512x512 raster patch (by default).

While PyGMTSAR strives to deliver optimal performance and uses high performance Python libraries, it is still relies on some of third-party binary tools. Although these tools perform well, they weren't designed for an interactive environment with lazy computations. These are the GMTSAR InSAR (InSAR processing system based on GMT) and the SNAPHU (Statistical-Cost, Network-Flow Algorithm for Phase Unwrapping) software. Consequently, some operations are separate processes using binary tools, making them less flexible compared to the core Python code. All binary tools called internally by PyGMTSAR are wrapped to be Dask-compatible. This required a set of patches to GMTSAR sources in the upstream repository, which is why PyGMTSAR requires a modern version of GMTSAR, adapted for it. In the future, GMTSAR binaries are planned to be fully replaced by PyGMTSAR alternatives that offer better performance and scalability. This process is, however, time-consuming. Frequently, it demands devising new methods to achieve the same outcomes, utilizing multiple processor cores and limited memory while preserving performance.

3.2. Lazy and Delayed Computations

The fastest calculations are those that are skipped, and the most effective algorithms do less work to achieve the same results. This is the essence of lazy, or delayed, numerical computations. In this approach, many operations are deferred, enabling you to obtain outputs almost instantly. However, the actual computations are performed only when you begin to use the values (delayed), and only for those specific values (lazy). If some results are never requested, the computations for those will never be carried out. This advantage makes Dask superior to traditional Linux/Unix task managers.

This concept can be likened to inspecting a new, unfurnished house. You can see it, touch it, and estimate its value, but it’s essentially empty. If you bring in your bed, you can sleep in it; if you bring your coffee machine, you can brew coffee, and so on. The same principle applies in programming. PyGMTSAR processing functions provide you with an ‘empty house,’ and it only ‘furnishes’ the rooms you visit and the items you actually need at the moment. This will be illustrated using some examples in the following sections.

When working with delayed or lazy computations, we’re dealing with operations that are not immediately executed. Instead, these computations are divided into a series of atomic tasks and scheduled to be run when required. This necessitates the initialization of a software called a parallel and distributed scheduler prior to performing computations. Fortunately, Dask is a modern, self-configurable scheduler, and its initialization is straightforward.

Initializing the Dask Scheduler

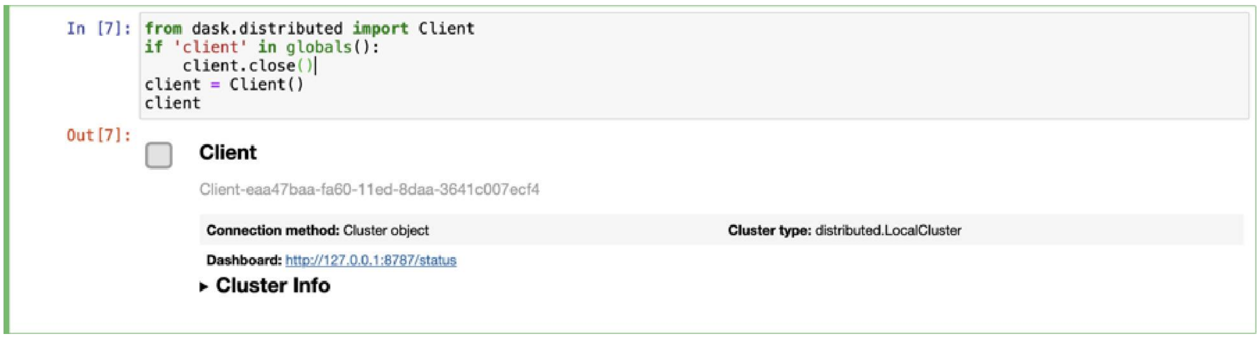
In the code snippet provided below, we demonstrate the initialization of the simplest single-host setup, although variations are included in the example Jupyter notebooks. By default, the straightforward “distributed.LocalCluster” scheduler is initiated. While this might not be the most efficient choice for processing large data stacks, it’s user-friendly and suits most typical projects. For further details, please refer to the Dask API

Distributing and scheduling processing jobs can be complex, and there may be times when things don’t go as planned, particularly if the execution is manually interrupted or an error occurs during processing. If such a situation arises, simply return to the Dask scheduler initialization cell and rerun it once or twice. This action will clear out any incomplete jobs, free up resources, and allow you to continue from where you left off.

However, it’s important to note that in the context of PyGMTSAR, the management of Dask tasks is further optimized through the use of the joblib library. This additional layer of resource management enhances reliability, as direct Dask code can occasionally fail to properly clean up resources in the event of an error or interruption. Thus, while PyGMTSAR handles these scenarios, there might be potential issues with your own code.

```
from dask.distributed import Client
if 'client' in globals():
    client.close()
client = Client()
```

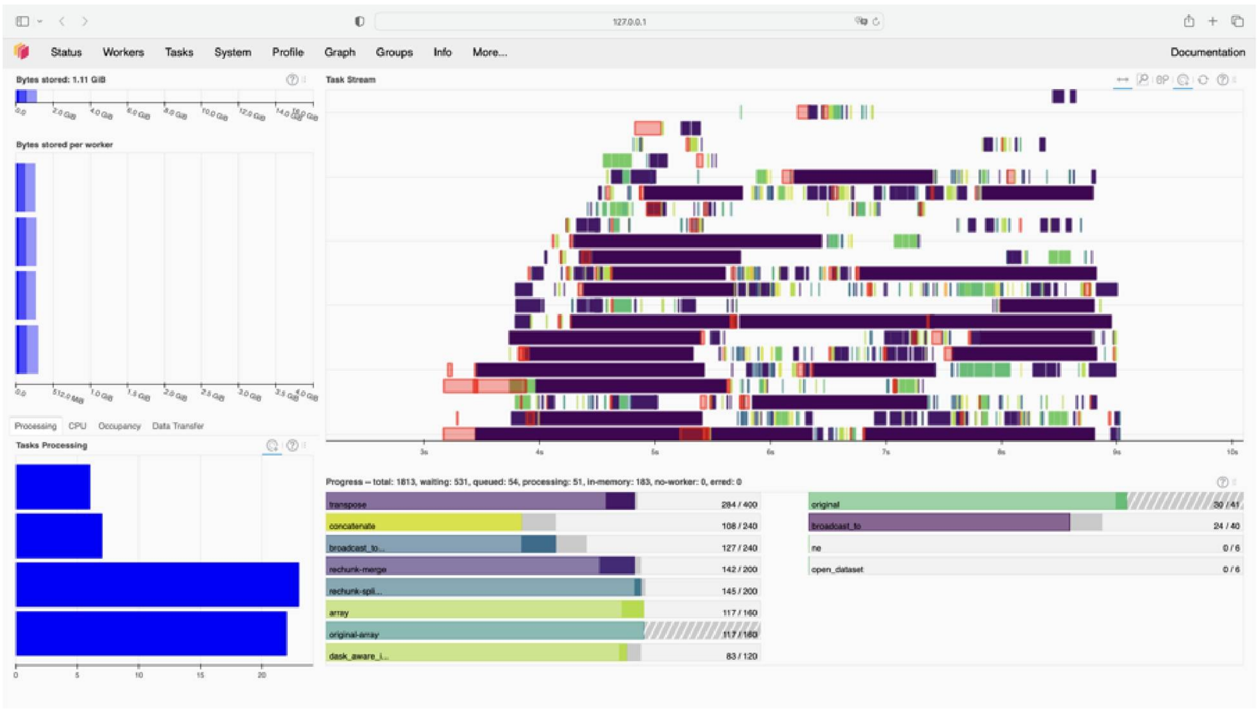
client



```
In [7]: from dask.distributed import Client
        if 'client' in globals():
            client.close()
        client = Client()
        client
```

```
Out[7]:
Client
Client-eaa47baa-fa60-11ed-8daa-3641c007ecf4
Connection method: Cluster object          Cluster type: distributed.LocalCluster
Dashboard: http://127.0.0.1:8787/status
▸ Cluster Info
```

One noteworthy feature is the option of monitoring internal parallel processing through the Dask “Dashboard” link provided in the cluster initialization cell. This tool offers useful insights into how your code manages parallel processing and can help identify potential performance issues by providing information on processor, memory, and network activity. However, remember the Dask dashboard is not available on Google Colab because it requires an additional network connection to serve the dashboard.



In summary, using the simple initialization code, we have a processing scheduler up and running in our Jupyter notebook or Python shell, which manages all parallel processing tasks. This setup offers several benefits, though it has certain limitations too, which we will explore in the following examples.

Performing Lazy or Delayed Computations

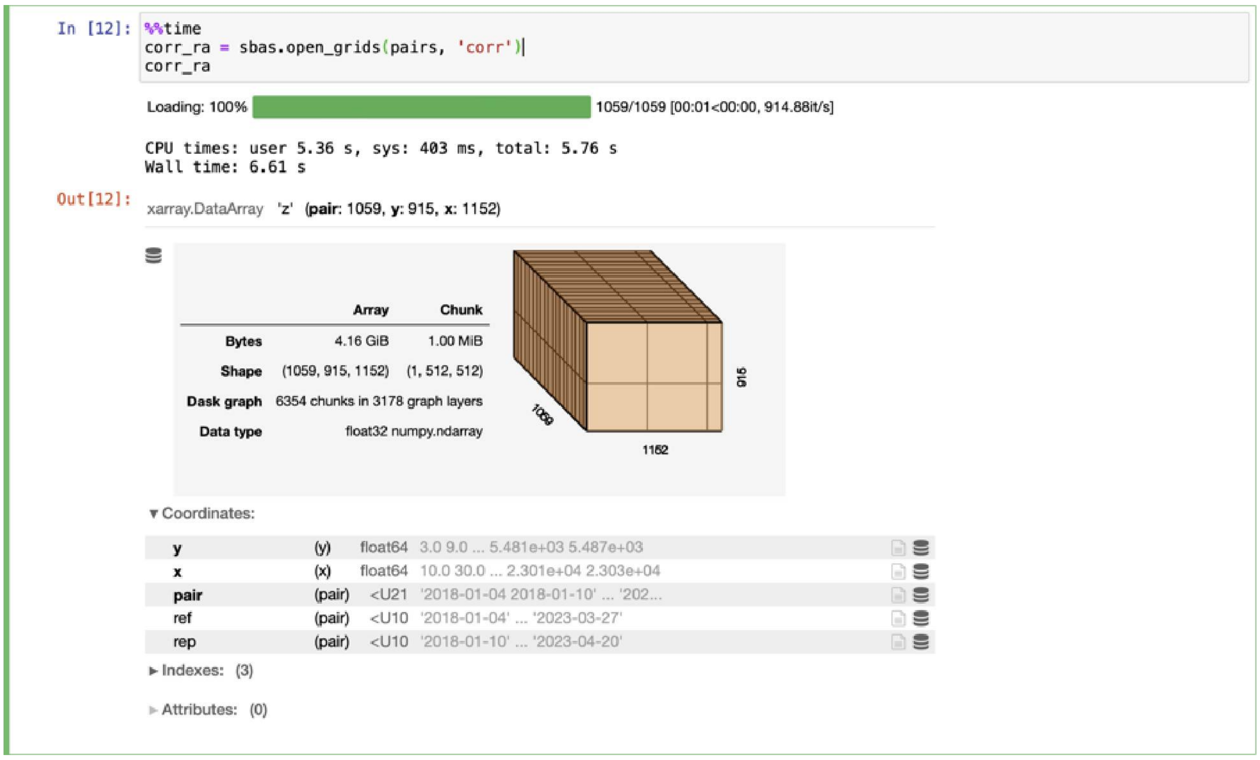
By using the Dask scheduler, we are able to execute Python code in a parallel and memory-efficient way across a distributed cluster or a single multicore computer. For educational purposes and to measure processing times, we employ a Jupyter “magic cell”, where the first line of code is This isn’t typically necessary for everyday tasks as PyGMTSAR provides progress indicators for operations that take a long time to run.

The processing pipeline typically begins with opening input datasets from NetCDF files lazily using the PyGMTSAR open_grids() function. From there, we can directly apply certain computations to these datasets through the open_grids() function arguments and also by operating on the returned lazy object using functions from NumPy, Xarray, and other Dask-compatible libraries. Upon reaching the desired result, we generate outputs, such as plots and NetCDF files, which are saved to disk. As long as every processing step supports delayed operations, the process remains effective and does not require any code modifications, as all the complexity is handled inside PyGMTSAR itself. The ideal user-level lazy processing code has no special modifications. To illustrate this, we’ll show a few common data processing examples.

Let’s begin by opening a large stack of correlation grids, specifically 1059 rasters, from NetCDF files in radar coordinates. This process takes only a few seconds (6.61s):

```
%%time
```

corr_ra = sbas.open_grids(pairs, 'corr')
corr_ra



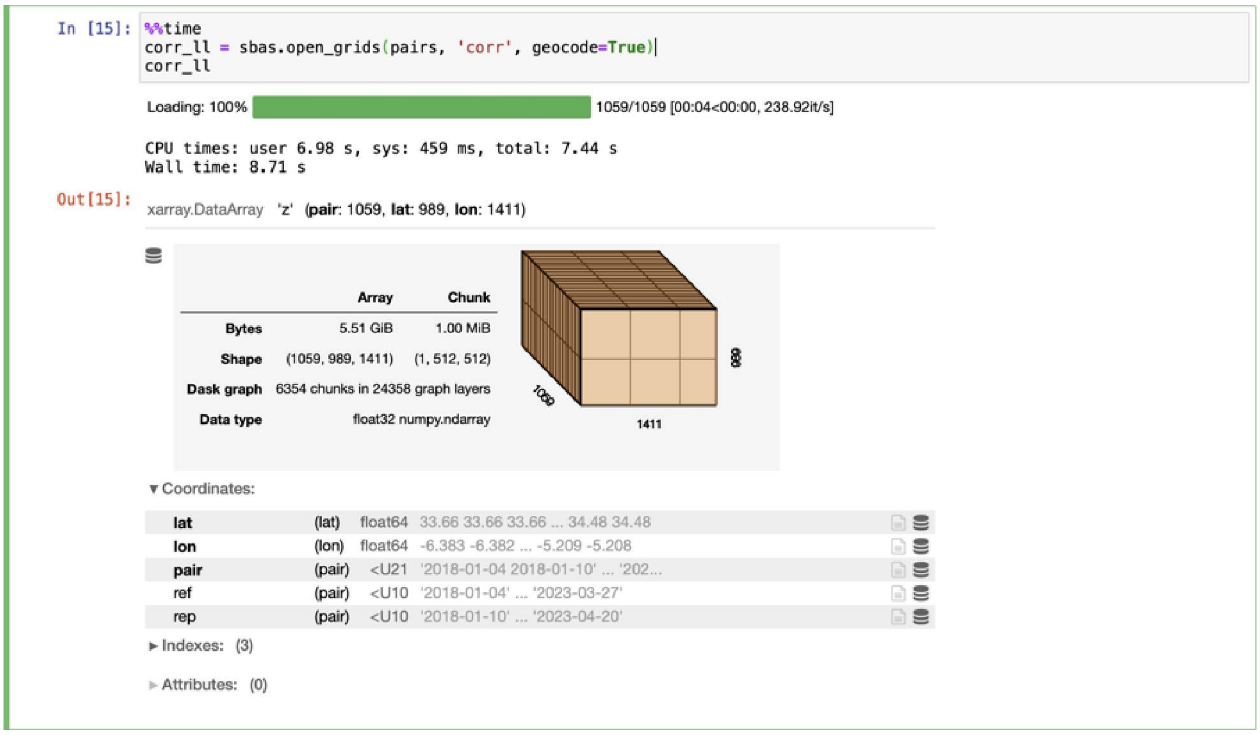
```
In [12]: %%time
         corr_ra = sbas.open_grids(pairs, 'corr')
         corr_ra

         Loading: 100%  1059/1059 [00:01<00:00, 914.88it/s]

         CPU times: user 5.36 s, sys: 403 ms, total: 5.76 s
         Wall time: 6.61 s

Out[12]: xarray.DataArray 'z' (pair: 1059, y: 915, x: 1152)

                      Array                  Chunk
         Bytes        4.16 GiB               1.00 MiB
         Shape        (1059, 915, 1152)      (1, 512, 512)
         Dask graph   6354 chunks in 3178 graph layers
         Data type    float32 numpy.ndarray

         ▾ Coordinates:
           y      (y)     float64  3.0 9.0 ... 5.481e+03 5.487e+03
           x      (x)     float64  10.0 30.0 ... 2.301e+04 2.303e+04
           pair   (pair)  <U21     '2018-01-04 2018-01-10' ... '202...
           ref    (pair)  <U10     '2018-01-04' ... '2023-03-27'
           rep    (pair)  <U10     '2018-01-10' ... '2023-04-20'
         ▸ Indexes: (3)
         ▸ Attributes: (0)
```

The Dask object is nicely visualized, providing detailed information such as array dimensions, coordinates and their limits, data type, and data chunks. While most of the properties are self-explanatory, data chunking is a crucial concept for lazy computations. This entails dividing the dataset into a collection of atomic data units — chunks — and Dask manages these chunks to plan and execute its tasks. The default chunk size is 512 (or rather, 512x512) for a 2D raster, which is typically sufficient. In Google Colab, there is no need to adjust this parameter, but for special cases, it can be fine-tuned during PyGMTSAR initialization and by using the chunksize argument in open_grids() and other functions. The 3D raster stack illustrated above has dimensions (1059, 915, 1152) and a chunk size of (1,512,512), resulting in a total of 6354 chunks. Therefore, to read the complete dataset, Dask performs as many as 6354 reading operations. However, Dask can't read a single pixel from a single grid, but only a block of raster pixels (512,512). Given the float32 data type, the single block size is 512 * 512 * 4 bytes = 1 which is adequate for standard InSAR computations. It might seem conservative and slightly better performance can be achieved with larger chunk sizes, like 1024 (1024x1024) or 2048 (2048x2048), particularly for interferogram processing, unwrapping, and detrending.

Challenges arise in steps that require pixel-wise processing, such as SBAS time series calculation, seasonal trend decomposition, etc. In such cases, the processing is performed for the full stack depth (1059 rasters) on 512x512 raster blocks, handling 1059 * 512 * 512 * 4 bytes = 264 MB data chunks. The actual analysis can require significantly more memory than the source data size, especially with numerous tasks running in parallel. To utilize 8 cores, Dask executes 8 parallel processing tasks on each 264 MB sub-stack. If the analysis requires 20 times more RAM than the data block, the total memory consumption equates to 264 MB * 8 cores * 20 = 42 GB memory. Fortunately, PyGMTSAR's core functions can estimate memory requirements and fine-tune Dask's chunk size for optimal performance, enabling it to run smoothly on any hardware. Without a solid understanding of the underlying principles and constraints, it is advisable not to adjust the parameters manually. Generally, allowing PyGMTSAR to manage these technical aspects automatically results in the best outcomes for most InSAR scenarios. However, for highly demanding projects and powerful hardware, PyGMTSAR does permit manual parameter tuning.

And we can open the same rasters with on-the-fly geocoding applied to transform rasters into geographic coordinates in a matter of seconds (8.71s):

```
%%time
corr_ll = sbas.open_grids(
    pairs,
    'corr',
    geocode=True
)
corr_ll
```



```
In [15]: %%time
         corr_ll = sbas.open_grids(pairs, 'corr', geocode=True)
         corr_ll
```

```
Loading: 100%  1059/1059 [00:04<00:00, 238.92it/s]

CPU times: user 6.98 s, sys: 459 ms, total: 7.44 s
Wall time: 8.71 s
```

```
Out[15]: xarray.DataArray 'z' (pair: 1059, lat: 989, lon: 1411)
```

	Array	Chunk
Bytes	5.51 GiB	1.00 MiB
Shape	(1059, 989, 1411)	(1, 512, 512)
Dask graph	6354 chunks in 24358 graph layers	
Data type	float32 numpy.ndarray	

▼ Coordinates:

```
lat     (lat)   float64  33.66 33.66 33.66 ... 34.48 34.48
lon     (lon)   float64  -6.383 -6.382 ... -5.209 -5.208
pair    (pair)  <U21     '2018-01-04 2018-01-10' ... '202...
ref     (pair)  <U10     '2018-01-04' ... '2023-03-27'
rep     (pair)  <U10     '2018-01-10' ... '2023-04-20'
```

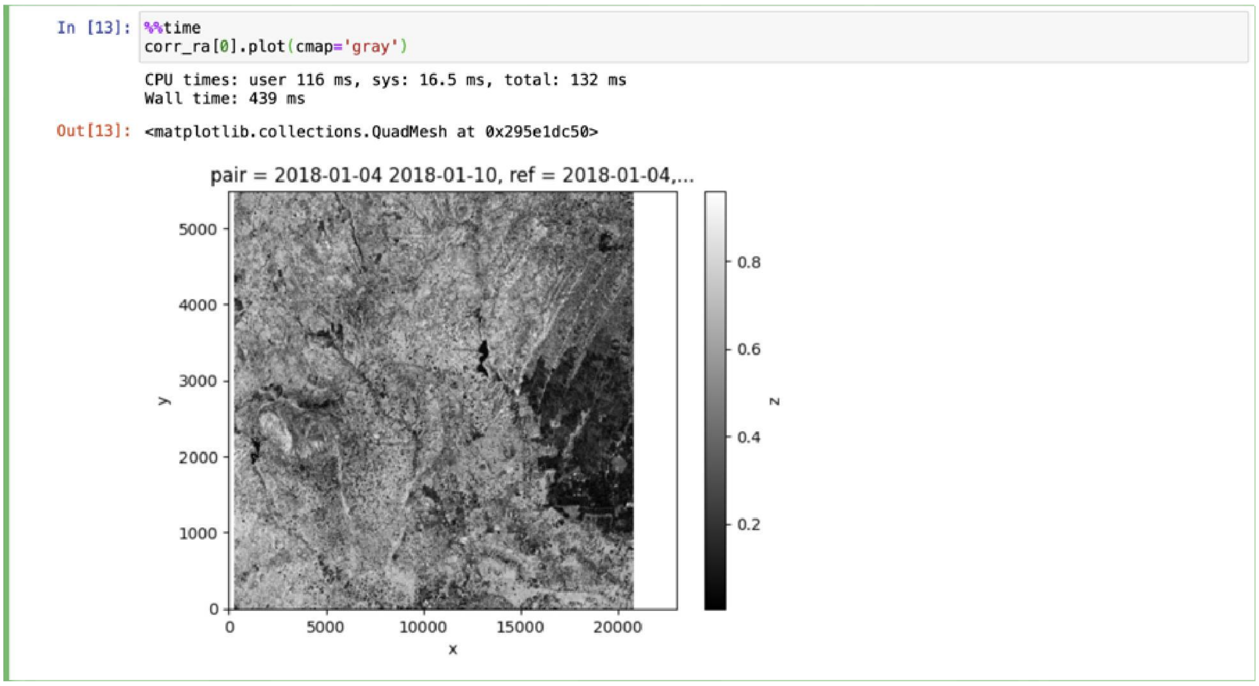
► Indexes: (3)

► Attributes: (0)

This is possible because only the data structures are pulled from the disk, not the actual values. And more, the Dask scheduler split the job to read the rasters stack into a set of parallel tasks to read the rasters properties right now and read the only required chunks of rasters later in case if we really need them. It's workable to transform the structures much faster than the real values. As a result, we see the output grid in geographic coordinates and the coordinates values—it looks like the complete result for us and that's all we need right now. We do not spend processing time and memory to perform any operations on the data yet.

Using the lazy grid stack object, we can map one grid in radar coordinates as usual (0.439s):

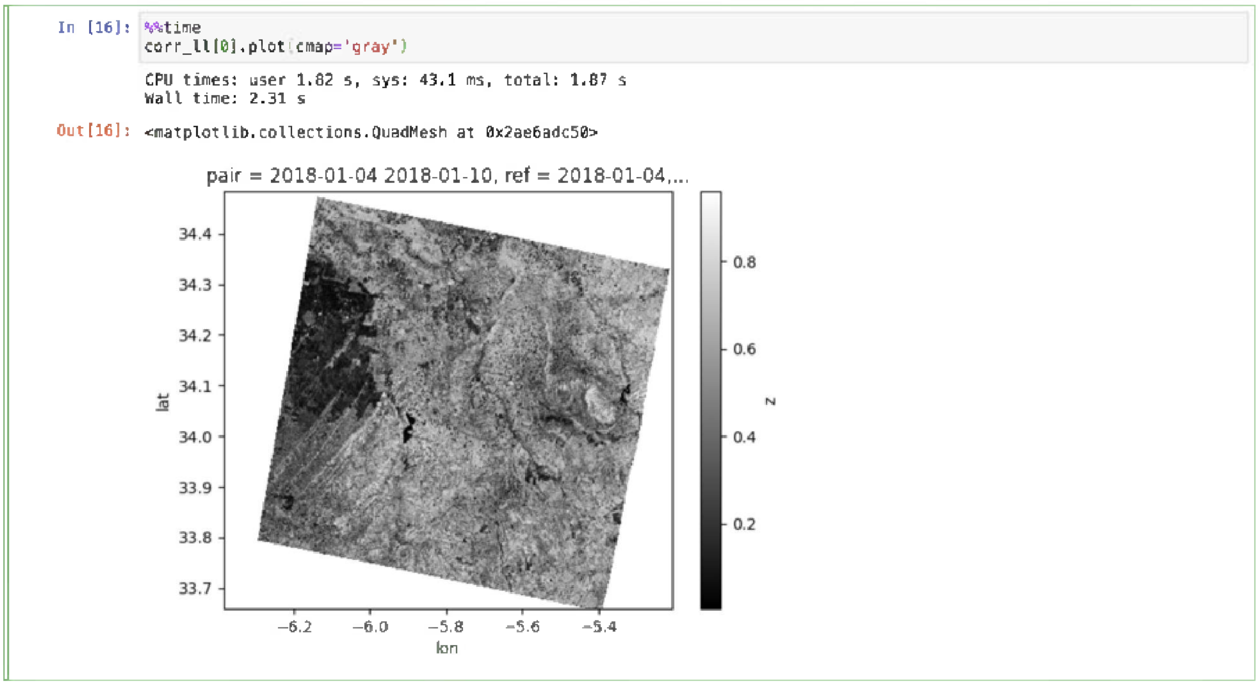
```
%%time
corr_ra[0].plot(cmap='gray')
```



```
In [13]: %%time
         corr_ra[0].plot(cmap='gray')

         CPU times: user 116 ms, sys: 16.5 ms, total: 132 ms
         Wall time: 439 ms

Out[13]: <matplotlib.collections.QuadMesh at 0x295e1dc50>
```

It works seamlessly, and it’s fast because here the data should be just read from the disk and mapped. As expected, the map in geographic coordinates requires a longer time to be mapped because it should be geocoded on-the-fly (2.31s):

```
%%time
corr_ll[0].plot(cmap='gray')
```

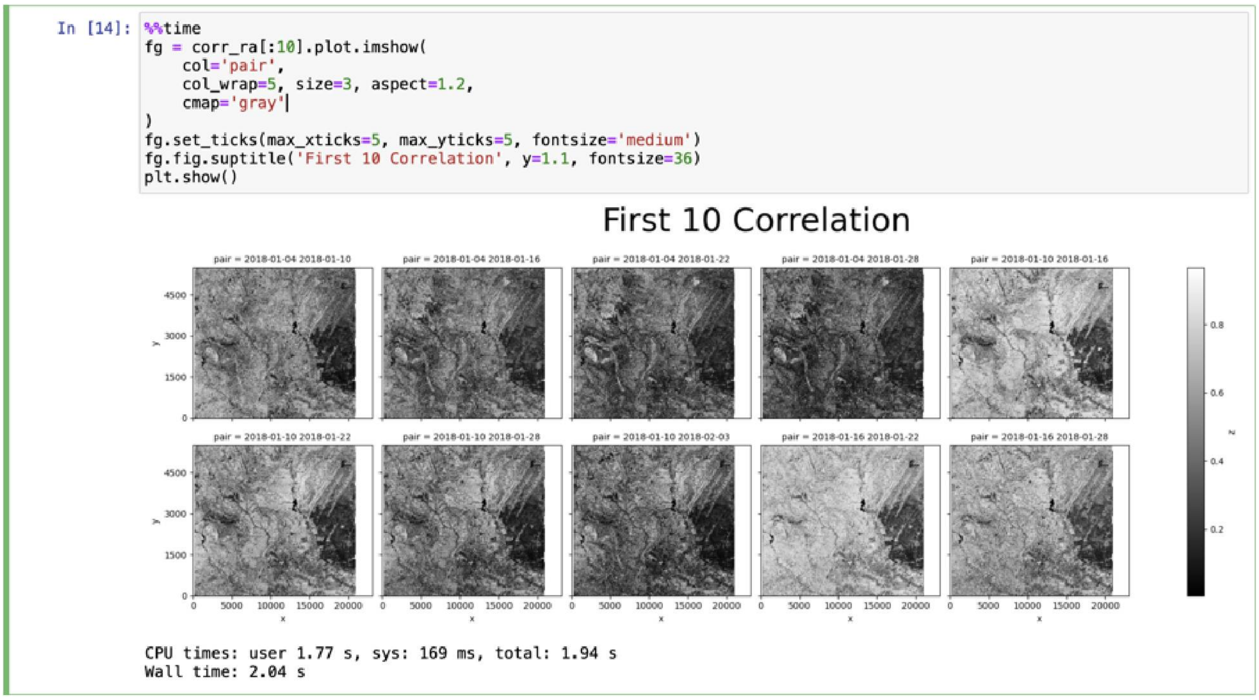


```
In [16]: %%time
         corr_ll[0].plot(cmap='gray')

         CPU times: user 1.82 s, sys: 43.1 ms, total: 1.87 s
         Wall time: 2.31 s

Out[16]: <matplotlib.collections.QuadMesh at 0x2ae6adc50>
```

Here still there is no excessive memory consumption to hold all the large grids in memory (RAM) and there are no 1000+ grid geocoding operations when we map just one grid. Dask scheduler performs only the required tasks to process and map exactly one grid. This standalone Python code does not require any modifications to the delayed processing.

The set of maps can be plotted effortlessly as well, with the plotting time here about 5x longer for 10 images (2.04s):

```
%%time
fg = corr_ra[:10].plot.imshow(
    col='pair',
    col_wrap=5, size=3,
    aspect=1.2, cmap='gray'
)
fg.set_ticks(max_xticks=5, max_yticks=5,
          fontsize='medium')
fg.fig.suptitle('First 10 Correlation',
              y=1.1, fontsize=36)
plt.show()
```



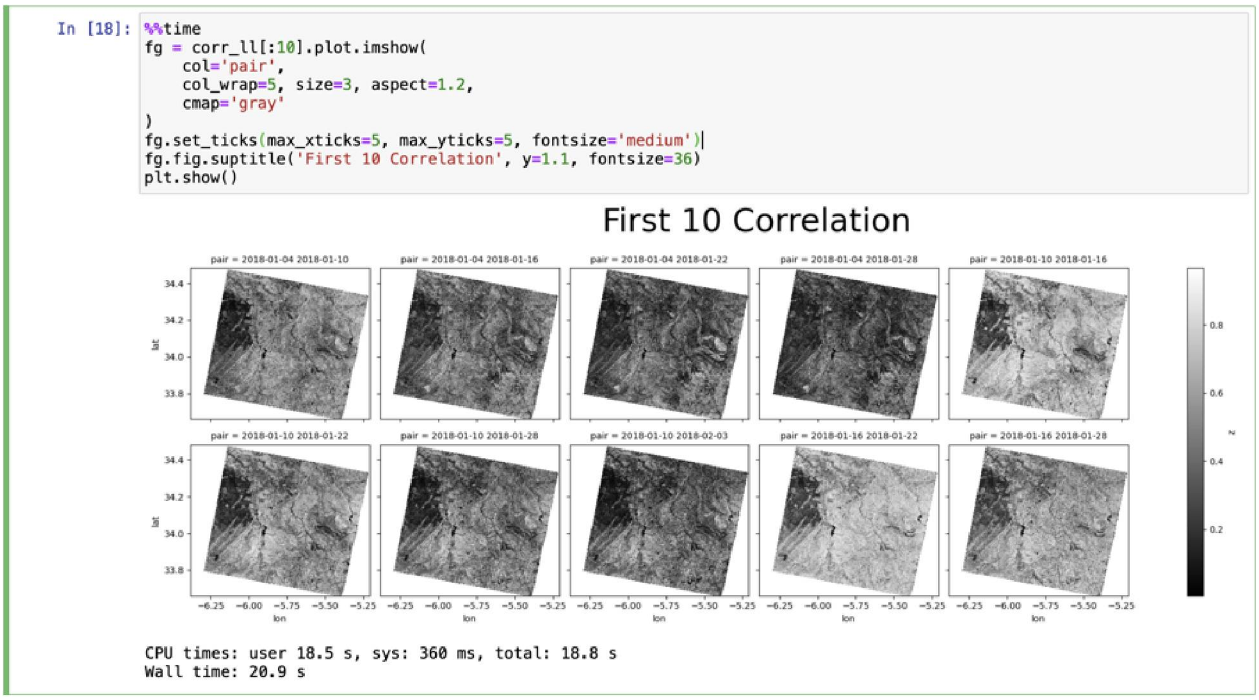
```
In [14]: %%time
         fg = corr_ra[:10].plot.imshow(
             col='pair',
             col_wrap=5, size=3, aspect=1.2,
             cmap='gray'
         )
         fg.set_ticks(max_xticks=5, max_yticks=5, fontsize='medium')
         fg.fig.suptitle('First 10 Correlation', y=1.1, fontsize=36)
         plt.show()
```

First 10 Correlation
pair = 2018-01-04 2018-01-10
pair = 2018-01-04 2018-01-16
pair = 2018-01-04 2018-01-22
pair = 2018-01-04 2018-01-28
pair = 2018-01-10 2018-01-16
pair = 2018-01-10 2018-01-22
pair = 2018-01-10 2018-01-28
pair = 2018-01-10 2018-02-03
pair = 2018-01-16 2018-01-22
pair = 2018-01-16 2018-01-28
y
x
z
0 1500 3000 4500
0 5000 10000 15000 20000
0.2 0.4 0.6 0.8

```
CPU times: user 1.77 s, sys: 169 ms, total: 1.94 s
Wall time: 2.04 s
```

And the same set of geocoded grids requires roughly 10x more time to be processed and mapped compared to a single grid (20.9s):

```
%%time
fg = corr_ll[:10].plot.imshow(
   col='pair',


   col_wrap=5, size=3,
   aspect=1.2, cmap='gray'
)
fg.set_ticks(max_xticks=5, max_yticks=5,
          fontsize='medium')
fg.fig.suptitle('First 10 Correlation',
              y=1.1, fontsize=36)
plt.show()
```


First 10 Correlation
pair = 2018-01-04 2018-01-10
pair = 2018-01-04 2018-01-16
pair = 2018-01-04 2018-01-22
pair = 2018-01-04 2018-01-28
pair = 2018-01-10 2018-01-16
pair = 2018-01-10 2018-01-22
pair = 2018-01-10 2018-01-28
pair = 2018-01-10 2018-02-03
pair = 2018-01-16 2018-01-22
pair = 2018-01-16 2018-01-28
lat
lon
z

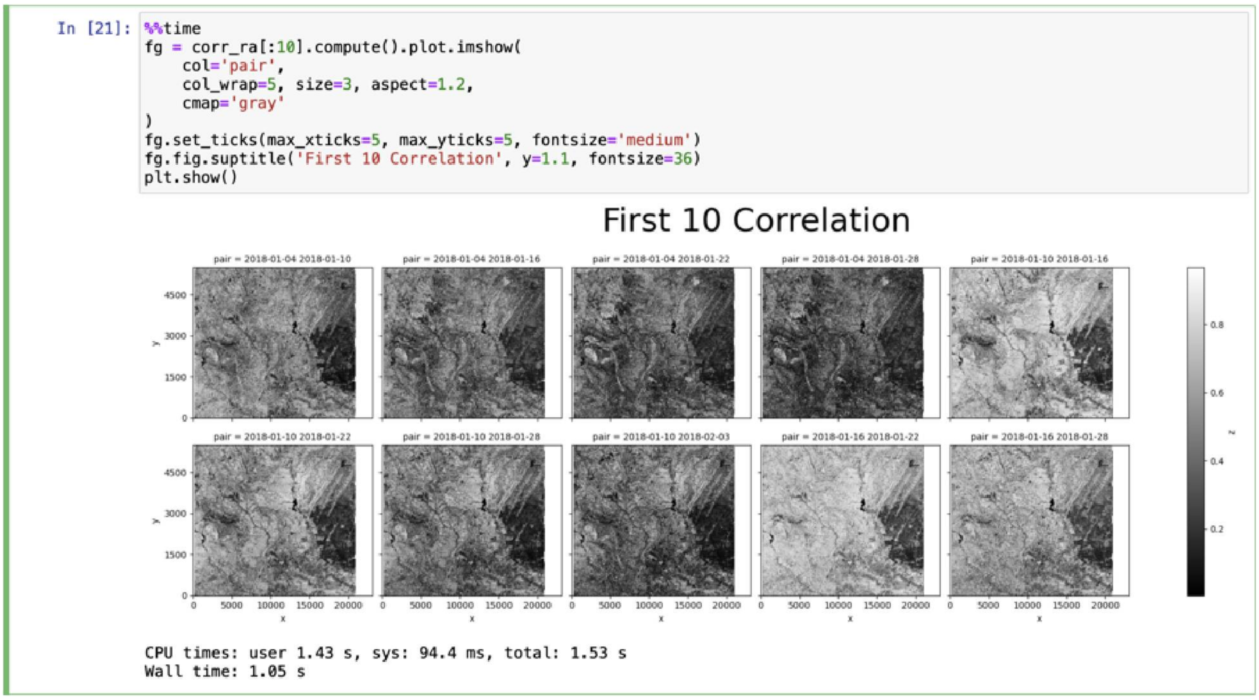
```
In [18]: %%time
         fg = corr_ll[:10].plot.imshow(
             col='pair',
             col_wrap=5, size=3, aspect=1.2,
             cmap='gray'
         )
         fg.set_ticks(max_xticks=5, max_yticks=5, fontsize='medium')
         fg.fig.suptitle('First 10 Correlation', y=1.1, fontsize=36)
         plt.show()
```

```
CPU times: user 18.5 s, sys: 360 ms, total: 18.8 s
Wall time: 20.9 s
```

While the extended processing time may seem reasonable, it’s not necessarily optimal. The code above is capable of working well with a single grid in a large stack, a set of grids, or all grids at once. However, we can accelerate operations if we know that the processing grids fit into the available memory—although failure in our estimations may lead to a system crash or extremely prolonged processing time.

The Dask compute() command executes all processing instantly, optimally as possible, and stores the complete result in memory. The critical point is to examine the lazy object first to ensure it fits comfortably into memory before we materialize it using In the next step, we prompt the necessary data grids to load from the disk—this step involves no computations but only data reading into memory—prior to plotting in radar coordinates, which takes around 1.05 seconds:

```
%%time
fg = corr_ra[:10].compute().plot.imshow(
   col='pair',
   col_wrap=5, size=3,
   aspect=1.2, cmap='gray'
)
fg.set_ticks(max_xticks=5, max_yticks=5,
          fontsize='medium')
fg.fig.suptitle('First 10 Correlation',
             y=1.1, fontsize=36)
plt.show()
```



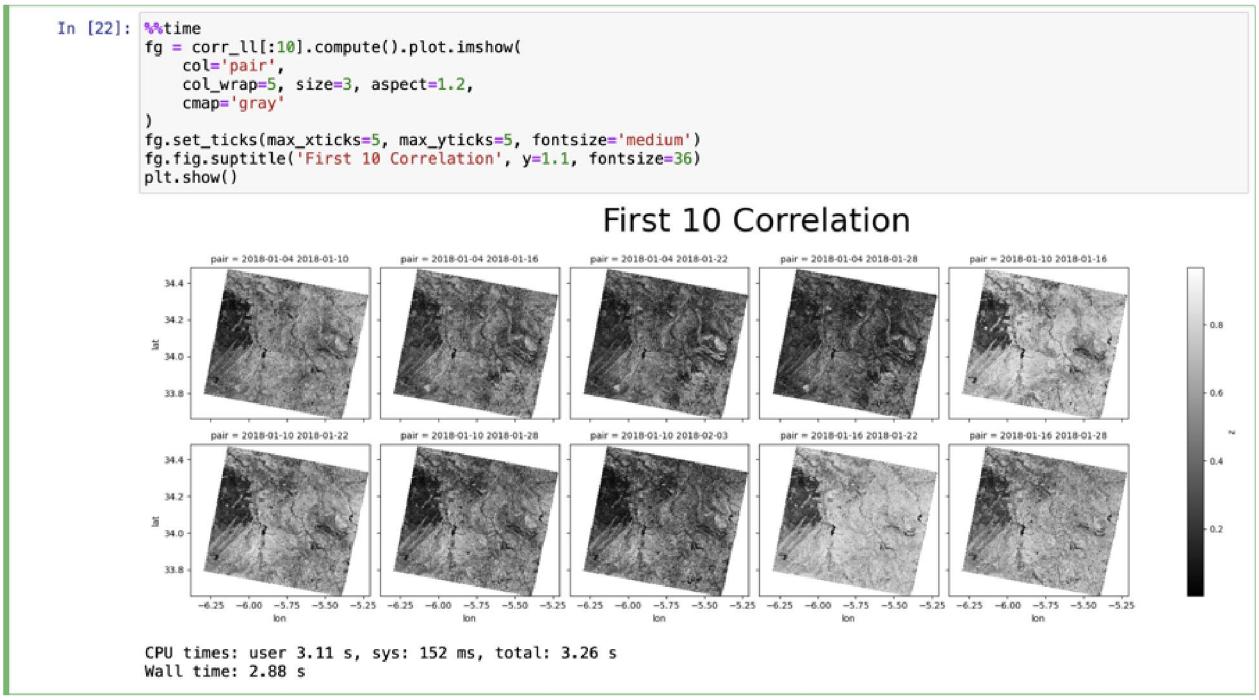
```
In [21]: %%time
         fg = corr_ra[:10].compute().plot.imshow(
             col='pair',
             col_wrap=5, size=3, aspect=1.2,
             cmap='gray'
         )
         fg.set_ticks(max_xticks=5, max_yticks=5, fontsize='medium')
         fg.fig.suptitle('First 10 Correlation', y=1.1, fontsize=36)
         plt.show()
```

First 10 Correlation
pair = 2018-01-04 2018-01-10
pair = 2018-01-04 2018-01-16
pair = 2018-01-04 2018-01-22
pair = 2018-01-04 2018-01-28
pair = 2018-01-10 2018-01-16
pair = 2018-01-10 2018-01-22
pair = 2018-01-10 2018-01-28
pair = 2018-01-10 2018-02-03
pair = 2018-01-16 2018-01-22
pair = 2018-01-16 2018-01-28
y
x
0 1500 3000 4500
0 5000 10000 15000 20000
0.2 0.4 0.6 0.8
z

```
CPU times: user 1.43 s, sys: 94.4 ms, total: 1.53 s
Wall time: 1.05 s
```

And in the following step, we plot in geographic coordinates. This step requires not only loading the necessary data grids from the disk but also performing geocoding computations. The entire process takes approximately 2.88 seconds:

```
%%time
fg = corr_ll[:10].compute().plot.imshow(


    col='pair',
    col_wrap=5, size=3,
    aspect=1.2, cmap='gray'
)
fg.set_ticks(max_xticks=5, max_yticks=5,
        fontsize='medium')
fg.fig.suptitle('First 10 Correlation',
           y=1.1, fontsize=36)
plt.show()
```



```
In [22]: %%time
         fg = corr_ll[:10].compute().plot.imshow(
             col='pair',
             col_wrap=5, size=3, aspect=1.2,
             cmap='gray'
         )
         fg.set_ticks(max_xticks=5, max_yticks=5, fontsize='medium')
         fg.fig.suptitle('First 10 Correlation', y=1.1, fontsize=36)
         plt.show()
```

First 10 Correlation
pair = 2018-01-04 2018-01-10
pair = 2018-01-04 2018-01-16
pair = 2018-01-04 2018-01-22
pair = 2018-01-04 2018-01-28
pair = 2018-01-10 2018-01-16
pair = 2018-01-10 2018-01-22
pair = 2018-01-10 2018-01-28
pair = 2018-01-10 2018-02-03
pair = 2018-01-16 2018-01-22
pair = 2018-01-16 2018-01-28
lat
34.4
34.2
34.0
33.8
lon
−6.25
−6.00
−5.75
−5.50
−5.25
z
0.8
0.6
0.4
0.2

```
CPU times: user 3.11 s, sys: 152 ms, total: 3.26 s
Wall time: 2.88 s
```

In conclusion, we compare the performance of the two approaches: Dask-managed automated processing and user-managed processing. The Dask-managed approach took 2.04 seconds for processing original rasters and 20.9 seconds for geocoded ones. On the other hand, the user-managed method took only 1.05 seconds for original rasters and 2.88 seconds for geocoded ones.

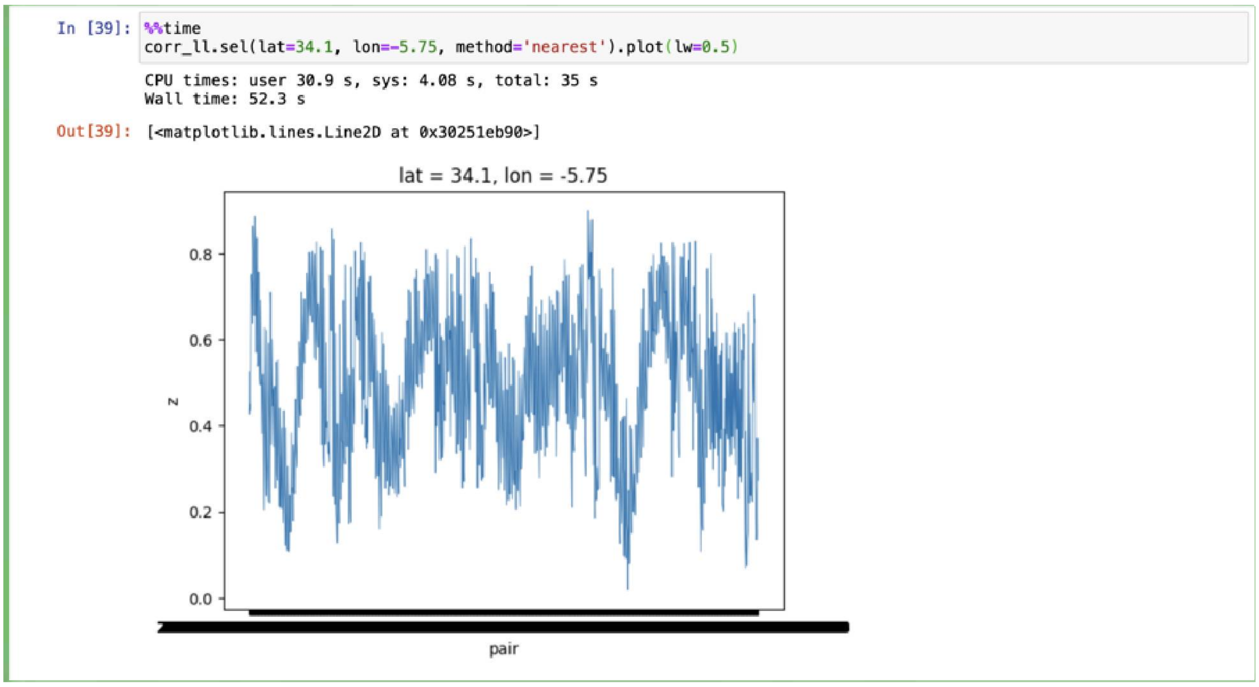
The extended processing time of the Dask-managed approach is a trade-off for its low memory consumption and robust processing. It processes and maps grids by chunks without storing all grids in memory at once, thus saving memory resources. However, for geocoding computations before plotting, the process requires 10 grids to be stored in memory simultaneously. The forced compute() call does this even if there isn't enough memory available.

This demonstrates the bargain between memory consumption and processing speed. If memory resources are abundant, user-managed processing can be quicker. Conversely, if memory is limited, the Dask-managed approach provides a more memory-efficient yet slower solution. In case when you guaranteed have more than sufficient memory on your computer you might gain profit using compute() command otherwise avoid it completely.

Pixel-wise line plots are effective as well, albeit managing vast amounts of data and operations internally. Plotting a single pixel for every one of the 1059 grids in our lazy stack implies that Dask must read and process the full 512x512 chunks. Thus, instead of a single pixel per raster, Dask operates on a 512 * 512 = 262144 pixel block on each raster. While this is still 6 times (the amount of Dask chunks per raster) better than processing the complete rasters, it is less efficient than retaining the complete rasters and all processing results in memory. In most cases, the potential speedup does not justify the added complexity and potential pitfalls of optimization. However, this approach proves beneficial for challenging projects involving enormous rasters where each 2D raster is comprised of numerous Dask chunks. Indeed, for our chosen case using 90 meter resolution, there are only 6 Dask chunks per raster, but for 15 meter resolution, we have roughly (90/15) * (90/15) * 6 = 216 chunks, meaning the single-pixel operation on all grids necessitates processing only 1/216th of the data.

Let's illustrate this theory with a single-pixel time series plot (52.3s):

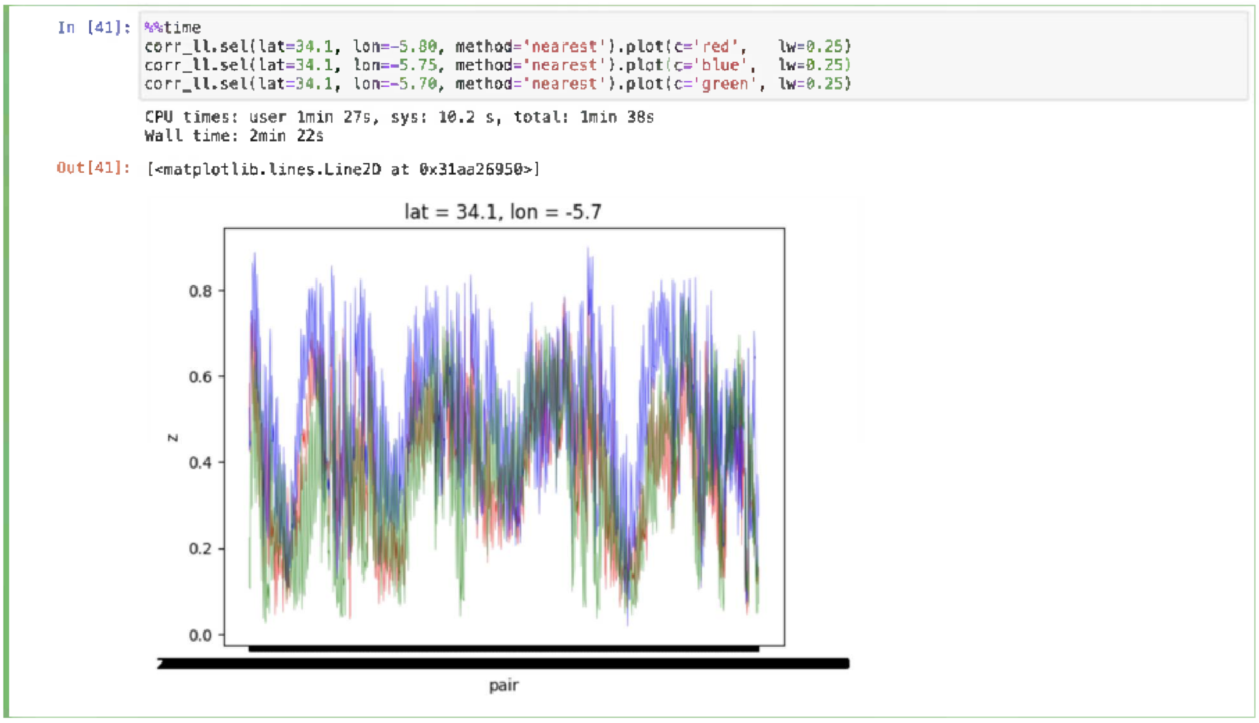
```
%%time
corr_ll.sel(
    lat=34.1, lon=-5.75,
    method='nearest'
).plot(lw=0.5)
```



```
In [39]: %%time
         corr_ll.sel(lat=34.1, lon=-5.75, method='nearest').plot(lw=0.5)

         CPU times: user 30.9 s, sys: 4.08 s, total: 35 s
         Wall time: 52.3 s

Out[39]: [<matplotlib.lines.Line2D at 0x30251eb90>]
```

Although we understand the necessary processing well, it doesn’t seem particularly swift. This even presents a more stringent trade-off between memory consumption and processing speed than for the 2D plots mentioned earlier. The method is both straightforward and quick for a single plot. However, when generating multiple plots, the process time increases significantly. As an example, it takes 2min 22s to plot 3 pixel timeseries together:

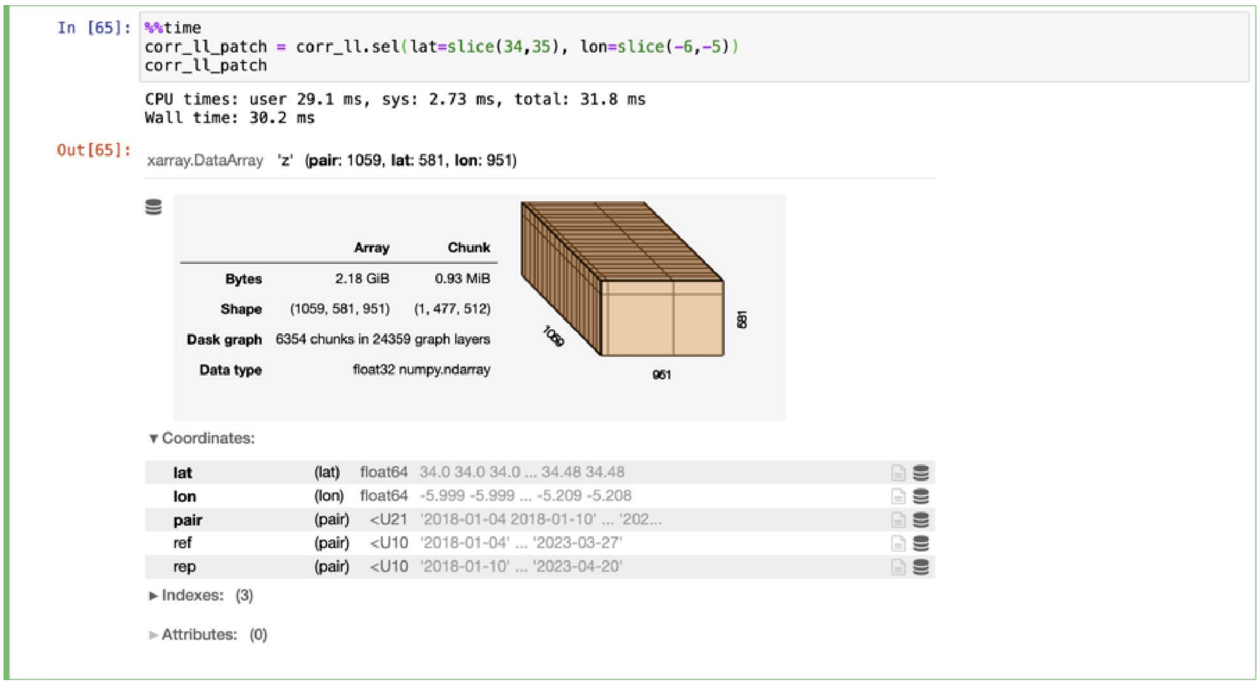
```
%%time
corr_ll.sel(
   lat=34.1, lon=-5.80,
   method='nearest'
).plot(c='red', lw=0.25)

corr_ll.sel(
   lat=34.1, lon=-5.75,
   method='nearest'
).plot(c='blue', lw=0.25)
corr_ll.sel(
   lat=34.1, lon=-5.70,
   method='nearest'
).plot(c='green', lw=0.25)
```



```
In [41]: %%time
         corr_ll.sel(lat=34.1, lon=-5.80, method='nearest').plot(c='red',   lw=0.25)
         corr_ll.sel(lat=34.1, lon=-5.75, method='nearest').plot(c='blue',  lw=0.25)
         corr_ll.sel(lat=34.1, lon=-5.70, method='nearest').plot(c='green', lw=0.25)

         CPU times: user 1min 27s, sys: 10.2 s, total: 1min 38s
         Wall time: 2min 22s

Out[41]: [<matplotlib.lines.Line2D at 0x31aa26950>]
```

If you wish to interactively plot multiple plots for exploration, the compute() function described earlier can be a timesaver if used judiciously. It allows us to select the necessary subset and compute it to conserve memory, for instance, a one-square-degree patch:

```
%%time
corr_ll_patch = corr_ll.sel(
    lat=slice(34,35),
    lon=slice(-6,-5)
)
corr_ll_patch
```



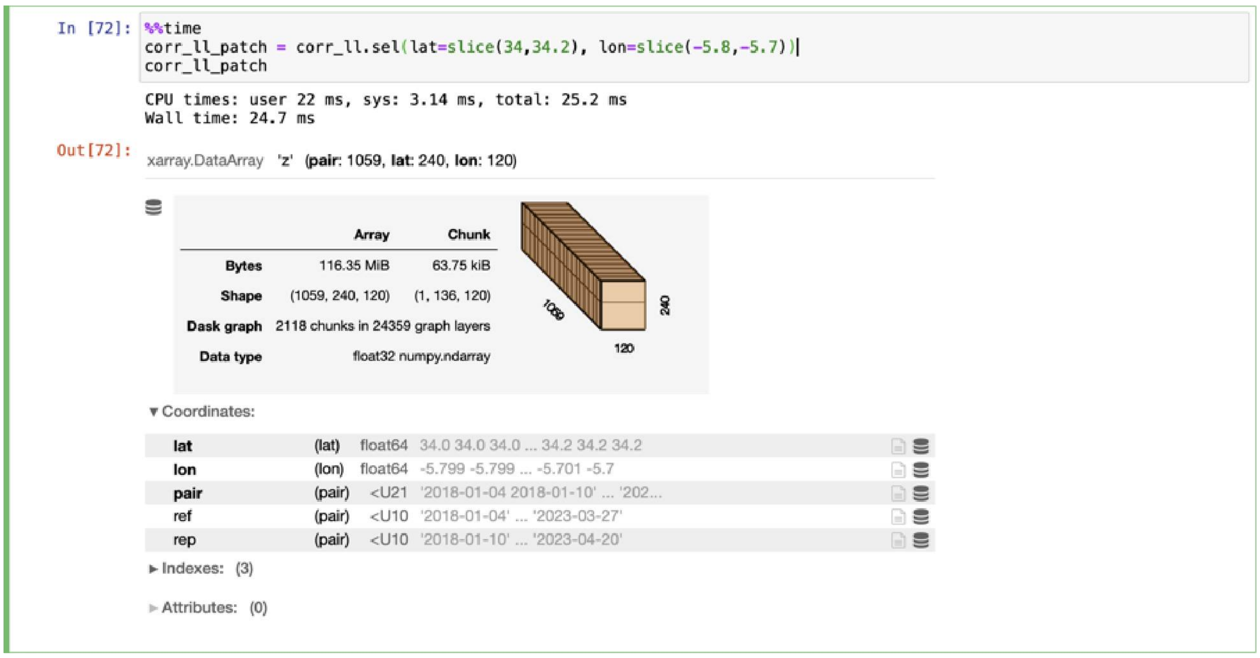
```
In [65]: %%time
         corr_ll_patch = corr_ll.sel(lat=slice(34,35), lon=slice(-6,-5))
         corr_ll_patch

         CPU times: user 29.1 ms, sys: 2.73 ms, total: 31.8 ms
         Wall time: 30.2 ms

Out[65]: xarray.DataArray 'z' (pair: 1059, lat: 581, lon: 951)
```

	Array	Chunk
Bytes	2.18 GiB	0.93 MiB
Shape	(1059, 581, 951)	(1, 477, 512)
Dask graph	6354 chunks in 24359 graph layers	
Data type		float32 numpy.ndarray

```
▾ Coordinates:
  lat    (lat)   float64  34.0 34.0 34.0 ... 34.48 34.48
  lon    (lon)   float64  -5.999 -5.999 ... -5.209 -5.208
  pair   (pair)  <U21     '2018-01-04 2018-01-10' ... '202...
  ref    (pair)  <U10     '2018-01-04' ... '2023-03-27'
  rep    (pair)  <U10     '2018-01-10' ... '2023-04-20'
▸ Indexes: (3)
▸ Attributes: (0)
```

However, the selected patch is barely smaller than the full grid and includes all the chunks. While the chunks are cropped, they still require excessive computations. We might reduce it considerably more for the target plots:

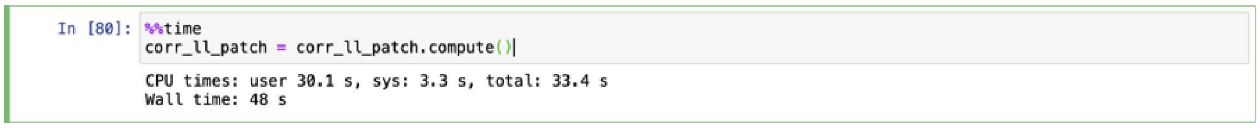
```
%%time
corr_ll_patch = corr_ll.sel(
    lat=slice(34,34.2),
    lon=slice(-5.8,-5.7)
)
corr_ll_patch
```



```
In [72]: %%time
         corr_ll_patch = corr_ll.sel(lat=slice(34,34.2), lon=slice(-5.8,-5.7))
         corr_ll_patch

         CPU times: user 22 ms, sys: 3.14 ms, total: 25.2 ms
         Wall time: 24.7 ms

Out[72]: xarray.DataArray 'z' (pair: 1059, lat: 240, lon: 120)

                         Array                Chunk
         Bytes           116.35 MiB           63.75 kiB
         Shape           (1059, 240, 120)     (1, 136, 120)
         Dask graph      2118 chunks in 24359 graph layers
         Data type                            float32 numpy.ndarray

         ▾ Coordinates:
           lat     (lat)   float64  34.0 34.0 34.0 ... 34.2 34.2 34.2
           lon     (lon)   float64  -5.799 -5.799 ... -5.701 -5.7
           pair    (pair)  <U21     '2018-01-04 2018-01-10' ... '202...
           ref     (pair)  <U10     '2018-01-04' ... '2023-03-27'
           rep     (pair)  <U10     '2018-01-10' ... '2023-04-20'
         ▸ Indexes: (3)
         ▸ Attributes: (0)
```

This patch is much smaller and has only two chunks while covering the area for the plots, so we can continue with it. And now that we're certain the patch is small and sufficient, we can call the compute() function to compute it (48s):

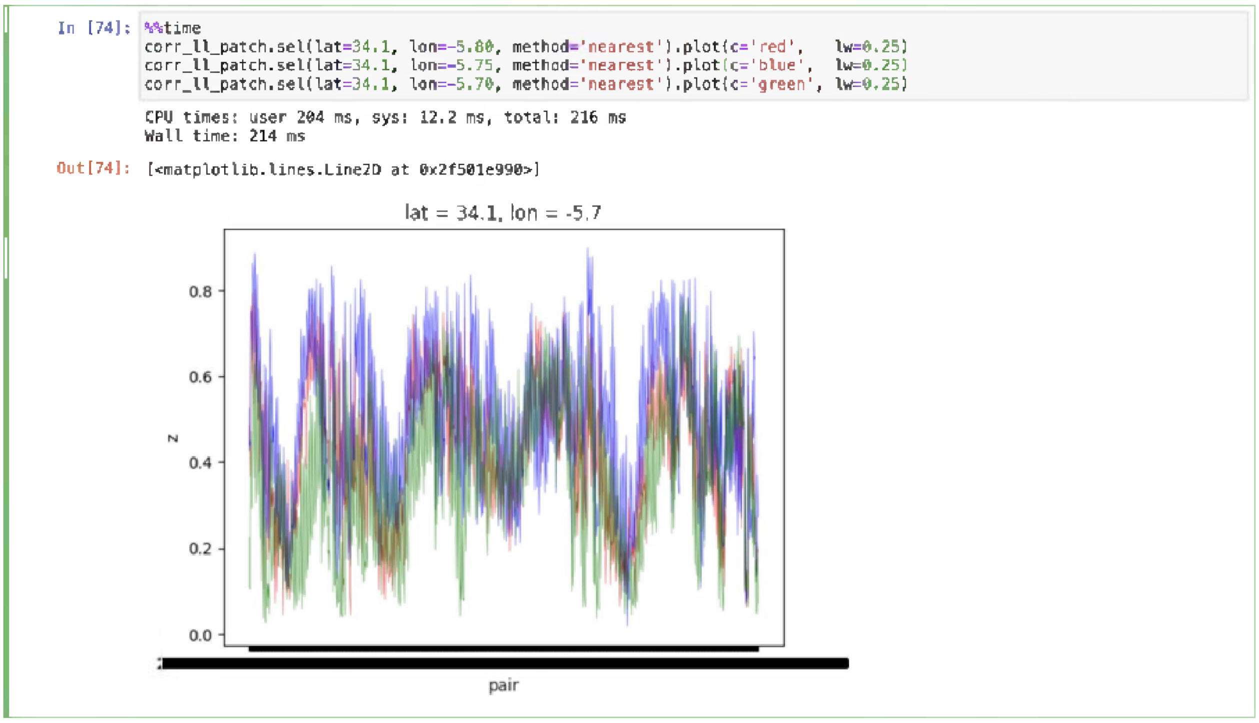
%%time
corr_ll_patch = corr_ll_patch.compute()



```
In [80]: %%time
         corr_ll_patch = corr_ll_patch.compute()

         CPU times: user 30.1 s, sys: 3.3 s, total: 33.4 s
         Wall time: 48 s
```

And having computed the small dataset, which doesn't require much memory, we can swiftly produce the same plots (214ms):

```
%%time
corr_ll_patch.sel(lat=34.1, lon=-5.80,
                  method='nearest').plot(
                      c='red', lw=0.25)
corr_ll_patch.sel(lat=34.1, lon=-5.75,
                  method='nearest').plot(
                      c='blue', lw=0.25)
corr_ll_patch.sel(lat=34.1, lon=-5.70,
                  method='nearest').plot(
                      c='green', lw=0.25)
```



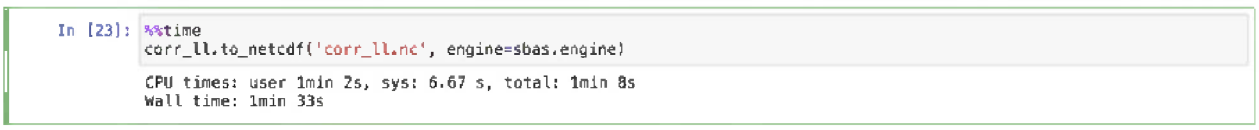
```
In [74]: %%time
         corr_ll_patch.sel(lat=34.1, lon=-5.80, method='nearest').plot(c='red',   lw=0.25)
         corr_ll_patch.sel(lat=34.1, lon=-5.75, method='nearest').plot(c='blue',  lw=0.25)
         corr_ll_patch.sel(lat=34.1, lon=-5.70, method='nearest').plot(c='green', lw=0.25)

         CPU times: user 204 ms, sys: 12.2 ms, total: 216 ms
         Wall time: 214 ms

Out[74]: [<matplotlib.lines.Line2D at 0x2f501e990>]
```

PyGMTSAR encourages the use of a fully lazy computation approach over premature optimization. It’s been extensively tested on a wide array of hardware configurations, proving itself to be both robust and fast. While manual optimization may grant you a few seconds of speedup for plotting smaller grids, it could lead to errors due to insufficient memory when dealing with larger grids. The time lost in recomputing numerous steps can far outweigh the brief moments saved. However, when careful optimizations are truly needed, and performed judiciously, the expected results can be achieved. This process merely requires caution and a check for the data size before invoking the compute() function on your data.

One of the advantages of utilizing a system like Dask for lazy computations is the predictability it provides. Your tasks will be processed efficiently and within a predictable time frame, regardless of whether you're using a common laptop or a powerful workstation. This predictability enables the implementation of features such as progress indicators for the long-time operations, such as saving results to disk. Let's examine some examples of how this can be accomplished.

All grids can be efficiently saved to a NetCDF file using a simple command. To ensure compatibility with lazy processing, we specify as the default engine might not be compatible. While the operation may sometimes take a while (in this case, 1 minute and 33 seconds), it runs effectively.

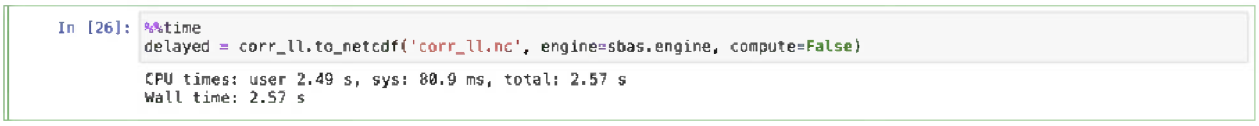
```
%%time
corr_ll.to_netcdf(
    'corr_ll.nc',
    engine=sbas.engine
)
```



```
In [23]: %%time
         corr_ll.to_netcdf('corr_ll.nc', engine=sbas.engine)

         CPU times: user 1min 2s, sys: 6.67 s, total: 1min 8s
         Wall time: 1min 33s
```

This operation can be monitored on the Dask dashboard if it’s available. However, it’s important to note that the dashboard isn’t accessible on Google Colab currently. As resources in this environment are somewhat limited, we usually can’t generate a large stack of substantial rasters here, and the saving operation is quick enough not to warrant a progress indication. Nevertheless, it’s still possible to produce sizable datasets even on Google Colab, and monitoring the process can be beneficial. We can convert the command above to a lazy one using the argument which defers the file saving operation:

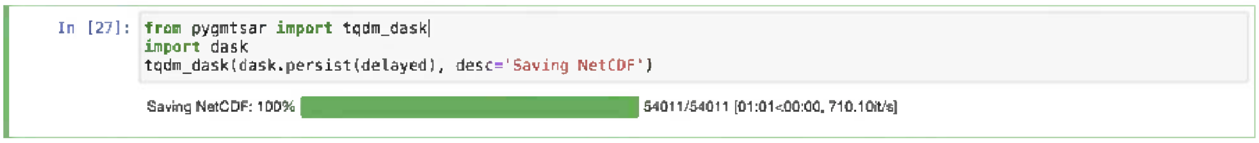
```
%%time
delayed = corr_ll.to_netcdf(
    'corr_ll.nc',
    engine=sbas.engine,
    compute=False
)
```



```
In [26]: %%time
         delayed = corr_ll.to_netcdf('corr_ll.nc', engine=sbas.engine, compute=False)

         CPU times: user 2.49 s, sys: 80.9 ms, total: 2.57 s
         Wall time: 2.57 s
```

PyGMTSAR provides the tqdm_dask progress indicator to monitor the processing of a lazy object in both Jupyter notebooks and console Python scripts:

```
from pygmtsar import tqdm dask
import dask
tqdm_dask(
    dask.persist(delayed),
    desc='Saving NetCDF'
)
```



```
In [27]: from pygmtsar import tqdm_dask
         import dask
         tqdm_dask(dask.persist(delayed), desc='Saving NetCDF')
```

```
Saving NetCDF: 100%  54011/54011 [01:01<00:00, 710.10it/s]
```

This handy trick makes your work more comfortable and is included in some of the example Google Colab notebooks that generate large output rasters.

Moreover, the benefits of Dask-guided lazy processing go beyond just this. A significant advantage is Dask's ability to automatically repeat failed operations. This means that issues such as disk access or network timeouts during distributed processing on a cluster can almost always be self-corrected, ensuring that we still get the expected results.

In essence, the delayed computation approach allows you to perform as many operations as needed without worrying about hardware limitations. It's all about performing smarter calculations, not harder ones. As a result, PyGMTSAR operates uniformly across a wide range of hardware, from Google Colab and decade-old basic MacBook Air laptops to modern multicore servers or computer clusters.

For a deeper understanding of lazy computing, consider exploring the Python libraries that enable such efficient computing. These include:

Enables lazy computations and provides an interface for parallelizing Python applications across computation clusters.
Supports grid processing with N-D labeled arrays and datasets in Python.

Fundamental package for scientific computing in Python, especially suited for numerical computations.
A Python-based ecosystem of open-source software for mathematics, science, and engineering.

A set of tools to provide lightweight pipelining in Python, particularly designed for tasks that are computationally expensive and can be executed in parallel.

These tools offer robust capabilities for managing complex computational tasks and can help make your coding more efficient and powerful.

3.3. The Primary SBAS Object

At the heart of PyGMTSAR lies the SBAS class and its corresponding object. The class is named after the well-known SBAS (Small Baseline Subset) method for InSAR processing. This component forms the backbone of the entire InSAR data processing and analysis pipeline within the PyGMTSAR framework.

The SBAS class is essential to the entire processing pipeline. It offers all user-level functions necessary for managing and analyzing InSAR data within PyGMTSAR, encapsulating all the InSAR processing building blocks we have explored in the previous sections.
The SBAS object serves as the central data structure within PyGMTSAR. It stores references to all the subswaths of Sentinel-1 scenes along with their corresponding orbits in a tabular format. This table, realized as a GeoPandas GeoDataFrame, includes a "geometry" column that provides the approximate boundaries of the scenes, as derived from Sentinel-1 SLC GeoTIFF ground control points (GCPs). Each subswath and scene date combination corresponds to a unique record. Initially, the subswath numbers are single digits (1, 2, or 3), but post-interferogram generation, these subswaths can be merged, and table records grouped for combined subswaths 12, 23, or 123.

Additionally, we discuss more advanced functionalities of the SBAS object, such as finding previously processed SBAS pairs and identifying SBAS dates. These features extend beyond the examples provided on

Google Colab and help us understand the usage of the PyGMTSAR Docker image mobigroup/pygmtsar-large shared on DockerHub.

Initializing the SBAS Object

The initialization process is straightforward and requires the definition of two directories:

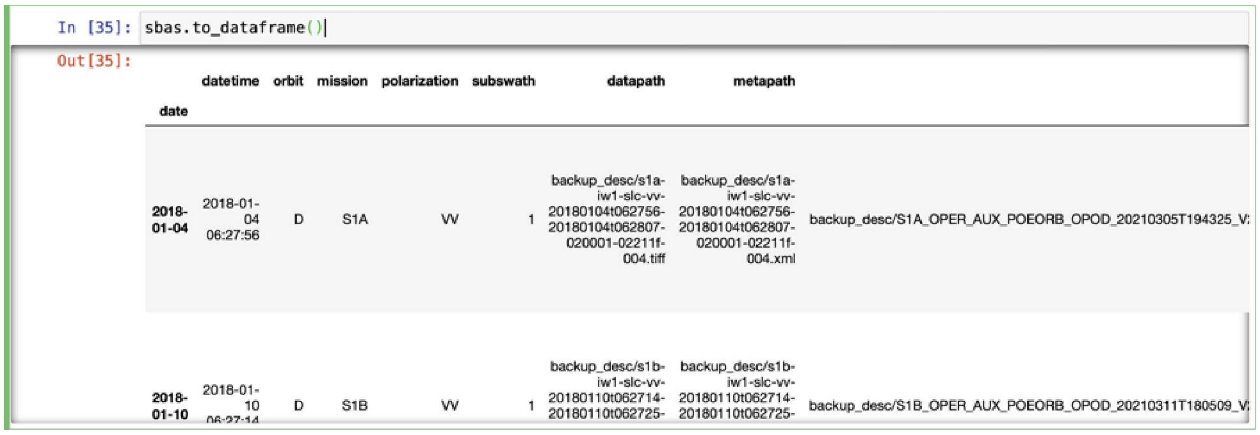
The data directory contains the Sentinel-1 SLC scenes and orbits. This directory can be read-only, as no changes will be made to it.
The working directory is where all processing will take place and the resulting grids will be stored. If this directory already exists, it will be dropped and recreated from scratch for the processing.

Here’s how you can set it up:

```
from pygmtsar import SBAS
sbas = SBAS(DATADIR, basedir=WORKDIR)
```

You can retrieve the current state of the SBAS object using the following command:

```
sbas.to_dataframe()
```



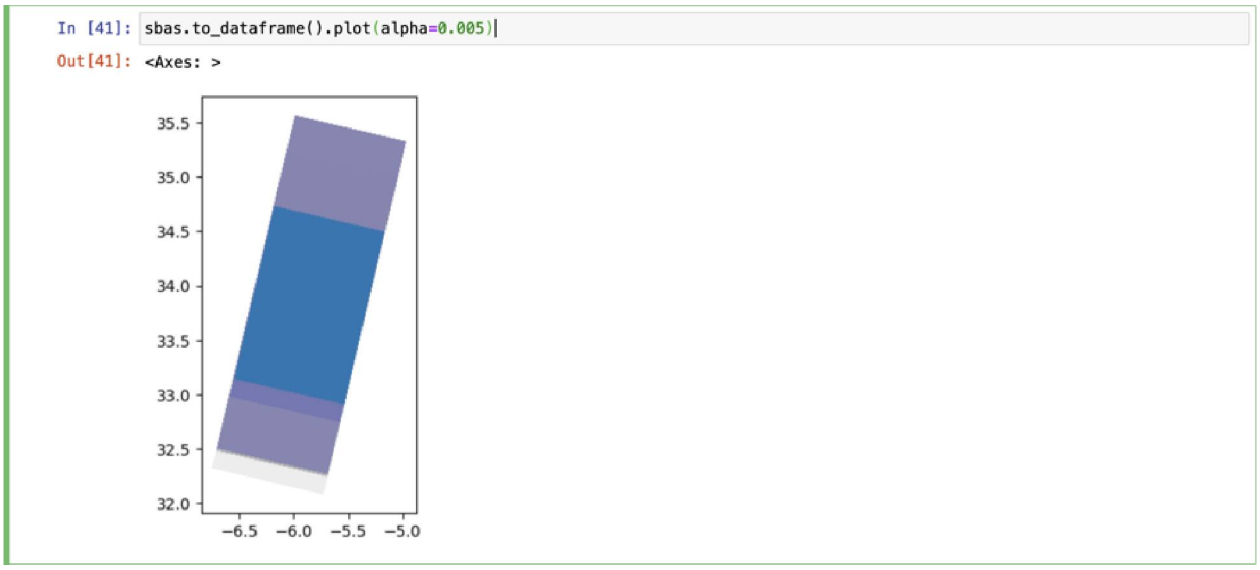
```
In [35]: sbas.to_dataframe()
Out[35]:
```

	datetime	orbit	mission	polarization	subswath	datapath	metapath	
date								
2018-01-04	2018-01-04 06:27:56	D	S1A	VV	1	backup_desc/s1a-iw1-slc-vv-20180104t062756-20180104t062807-020001-02211f-004.tiff	backup_desc/s1a-iw1-slc-vv-20180104t062756-20180104t062807-020001-02211f-004.xml	backup_desc/S1A_OPER_AUX_POEORB_OPOD_20210305T194325_V:
2018-01-10	2018-01-10 06:27:14	D	S1B	VV	1	backup_desc/s1b-iw1-slc-vv-20180110t062714-20180110t062725-	backup_desc/s1b-iw1-slc-vv-20180110t062714-20180110t062725-	backup_desc/S1B_OPER_AUX_POEORB_OPOD_20210311T180509_V:

Plotting the SBAS GeoDataFrame is simple, while for a large stack of interferograms, it’s essential to adjust opacity due to overlapping areas:

---

sbas.to_dataframe().plot(alpha=0.005)

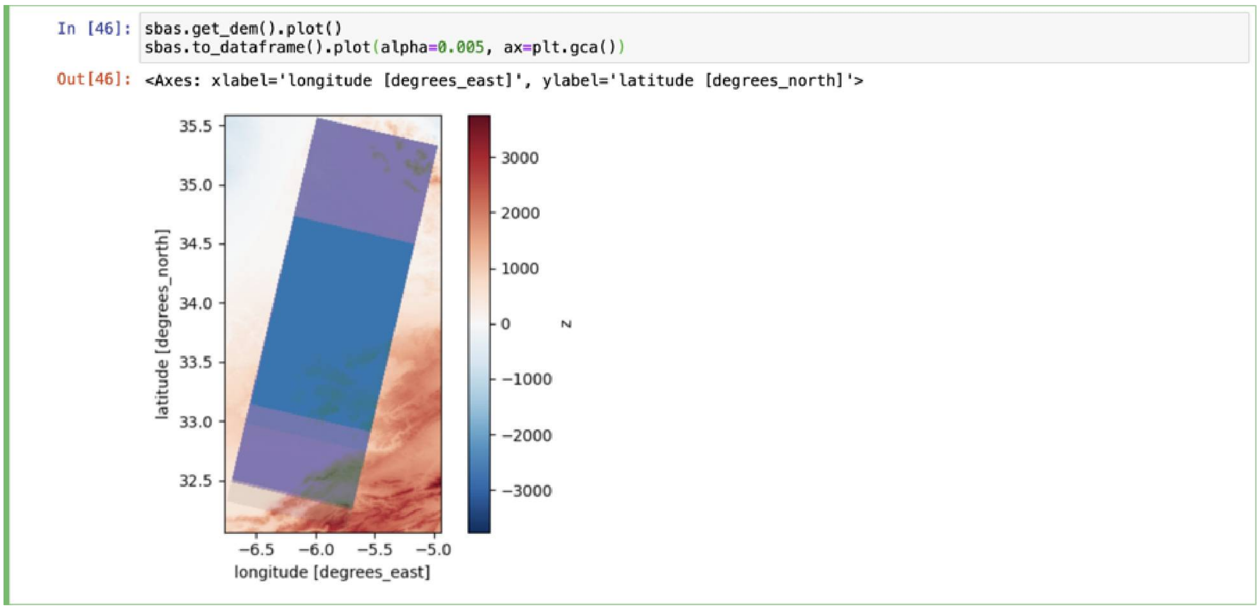


```
In [41]: sbas.to_dataframe().plot(alpha=0.005)
Out[41]: <Axes: >
```

Scenes can be plotted atop the DEM (Digital Elevation Model) as follows:

---

sbas.get_dem().plot()
sbas.to_dataframe().plot(alpha=0.005, ax=plt.gca())



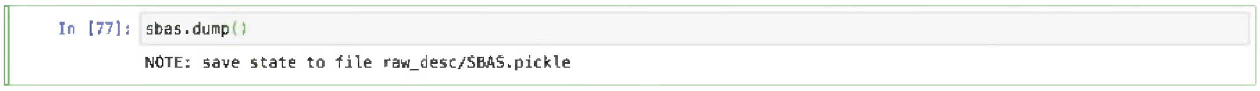
```
In [46]: sbas.get_dem().plot()
         sbas.to_dataframe().plot(alpha=0.005, ax=plt.gca())

Out[46]: <Axes: xlabel='longitude [degrees_east]', ylabel='latitude [degrees_north]'>
```

Saving and Restoring the SBAS Object

It's not always necessary to discard all results and rerun all steps from the beginning. You can save your current state within the working directory and close the notebook, then continue your work later, even on a different host, using the SBAS.restore() command.

To dump the current state into the working directory defined during SBAS initialization, use:

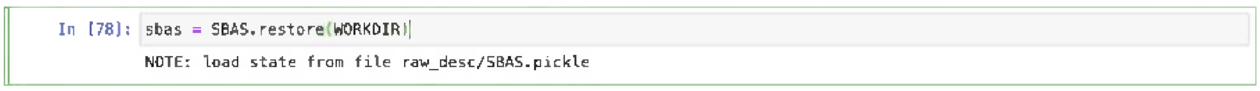
```
sbas.dump()
```



```
In [77]: sbas.dump()
         NOTE: save state to file raw_desc/SBAS.pickle
```

Later, you can restore the current state in the same or different notebook or script to continue. Use:

```
sbas = SBAS.restore(WORKDIR)
```



```
In [78]: sbas = SBAS.restore(WORKDIR)
         NOTE: load state from file raw_desc/SBAS.pickle
```

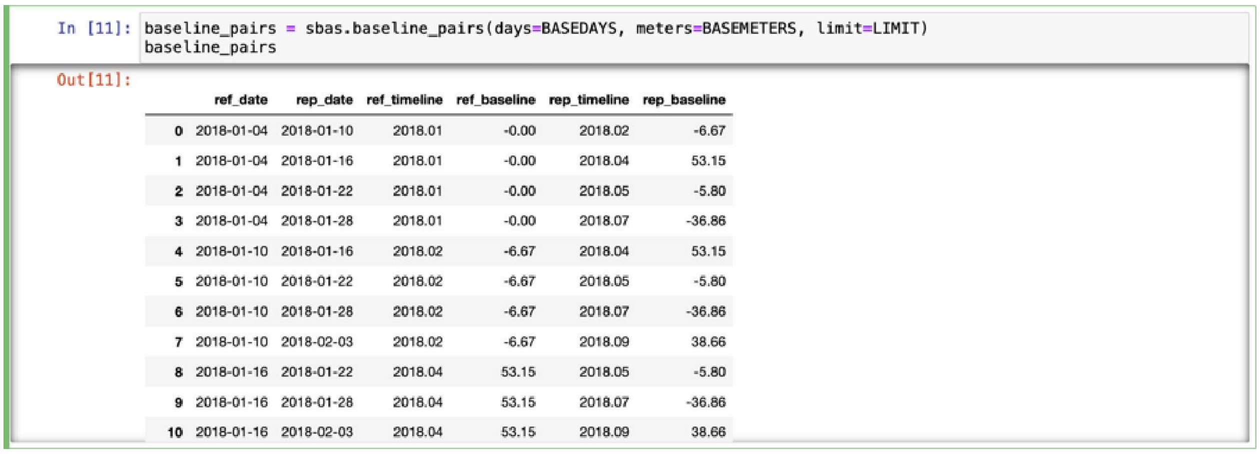
This allows you to continue from where you left off without losing previous computation results.

Creating SBAS Pairs

To create interferograms, we need to define the pairs of images (dates). Each pair represents two SAR acquisitions used to form and process interferograms. For two images, there are two possible pairs—direct and inverse. For three images, there are six possible pairs, and so on. You can define these pairs manually or by using built-in methods.

The SBAS object in PyGMTSAR offers a method to define Small Baseline Subset (SBAS) interferogram pairs. This is achieved by specifying parameters such as the maximum date interval (BASEDAYS), the perpendicular baseline (BASEMETERS), and a limit on the number of pairs beginning from a certain date (LIMIT). These pairs, defined by baseline, are critical for the interferogram processing functions. Here's how you can generate such pairs:

```
baseline_pairs = sbas.baseline_pairs(
    days=BASEDAYS,
    meters=BASEMETERS,
    limit=LIMIT
)
baseline_pairs
```



```
In [11]: baseline_pairs = sbas.baseline_pairs(days=BASEDAYS, meters=BASEMETERS, limit=LIMIT)
         baseline_pairs
```

Out[11]:

	ref_date	rep_date	ref_timeline	ref_baseline	rep_timeline	rep_baseline
0	2018-01-04	2018-01-10	2018.01	-0.00	2018.02	-6.67
1	2018-01-04	2018-01-16	2018.01	-0.00	2018.04	53.15
2	2018-01-04	2018-01-22	2018.01	-0.00	2018.05	-5.80
3	2018-01-04	2018-01-28	2018.01	-0.00	2018.07	-36.86
4	2018-01-10	2018-01-16	2018.02	-6.67	2018.04	53.15
5	2018-01-10	2018-01-22	2018.02	-6.67	2018.05	-5.80
6	2018-01-10	2018-01-28	2018.02	-6.67	2018.07	-36.86
7	2018-01-10	2018-02-03	2018.02	-6.67	2018.09	38.66
8	2018-01-16	2018-01-22	2018.04	53.15	2018.05	-5.80
9	2018-01-16	2018-01-28	2018.04	53.15	2018.07	-36.86
10	2018-01-16	2018-02-03	2018.04	53.15	2018.09	38.66

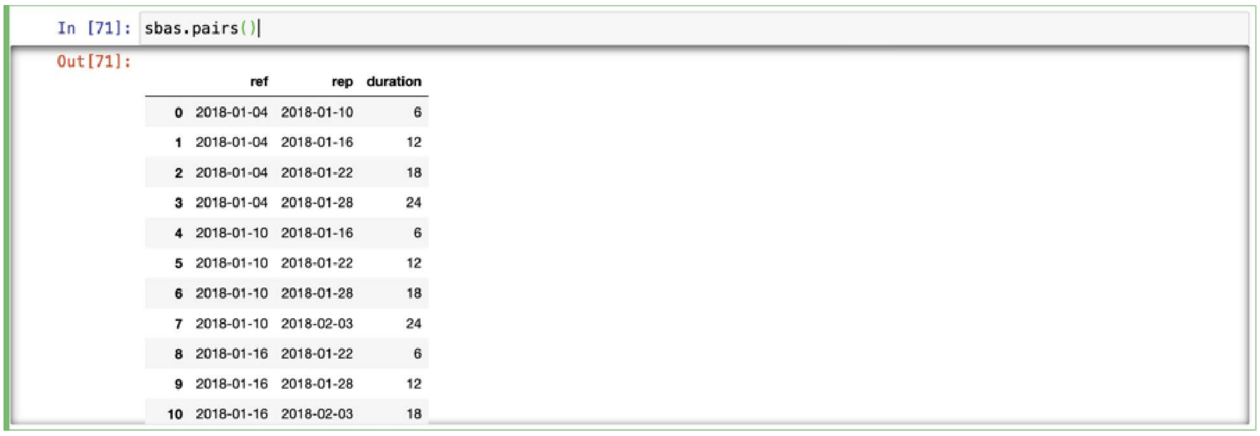
The output is a Pandas DataFrame that can be readily manipulated, allowing you to filter and plot the baseline pairs with ease.

The baseline pairs are used for interferogram creation and after that we can continue to use the pairs for the next processing step or read the existing SBAS pairs corresponding to the created interferograms.

Reading SBAS Pairs

PyGMTSAR provides a feature that allows you to identify all existing pairs from already processed grids. This is particularly useful when you want to proceed with other processing tasks, such as unwrapping, detrending, and more, in separate Jupyter notebooks or Python scripts. While the baseline_pairs() function creates only a list of SBAS pairs for future interferogram processing, the pairs() function reads from disk the interferograms that have already been processed.

```
sbas.pairs()
```



```
In [71]: sbas.pairs()
Out[71]:
            ref         rep  duration
 0   2018-01-04  2018-01-10         6
 1   2018-01-04  2018-01-16        12
 2   2018-01-04  2018-01-22        18
 3   2018-01-04  2018-01-28        24
 4   2018-01-10  2018-01-16         6
 5   2018-01-10  2018-01-22        12
 6   2018-01-10  2018-01-28        18
 7   2018-01-10  2018-02-03        24
 8   2018-01-16  2018-01-22         6
 9   2018-01-16  2018-01-28        12
10   2018-01-16  2018-02-03        18
```

This method returns a DataFrame with the pairs identified from the processed grids. You can then proceed with additional processing steps using these pairs in the same or another Jupyter notebook or Python script.

3.4. Data Reading and Writing

PyGMTSAR primarily operates with 2D NetCDF grids, using 3D grids for time-series analyses for more efficiency. For compatibility reasons, SBAS results are stored in 2D grids. 3D grids are not well for use on Google Colab while other platforms are better suited for larger time-series processing.

While PyGMTSAR supports importing and exporting GeoTIFF files for 2D GIS software processing (such as QGIS, GDAL, etc.), NetCDF is the recommended format. This recommendation is primarily due to NetCDF's superior performance in lazy processing, aligning with the PyGMTSAR paradigm. Exporting large GeoTIFF files is not as robust, and it requires careful handling. Therefore, GeoTIFF should only be used when necessary.

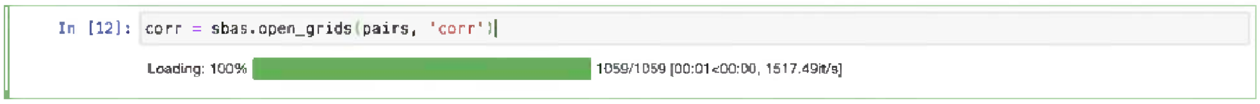
Due to performance reasons, PyGMTSAR does not provide user-level functions to write internal NetCDF files. Instead, it focuses on the data processing pipeline and effective data storage. PyGMTSAR processing functions handle these tasks well, and you can easily read the outputs and export them into different output formats as illustrated below.

Reading PyGMTSAR 2D Grids

2D NetCDF grids serve as the internal storage format for PyGMTSAR. These grids are chunked for lazy parallel access and compressed to optimize disk consumption and read-write performance.

The function pygmtsar.SBAS.open_grids performs the read operations inside the PyGMTSAR working directory. The code below illustrates how to open a set of 2D grids named 'corr':

```
sbas.open_grids(pairs, 'corr')
```



```
In [12]: corr = sbas.open_grids(pairs, 'corr')
         Loading: 100%  1059/1059 [00:01<00:00, 1517.49it/s]
```

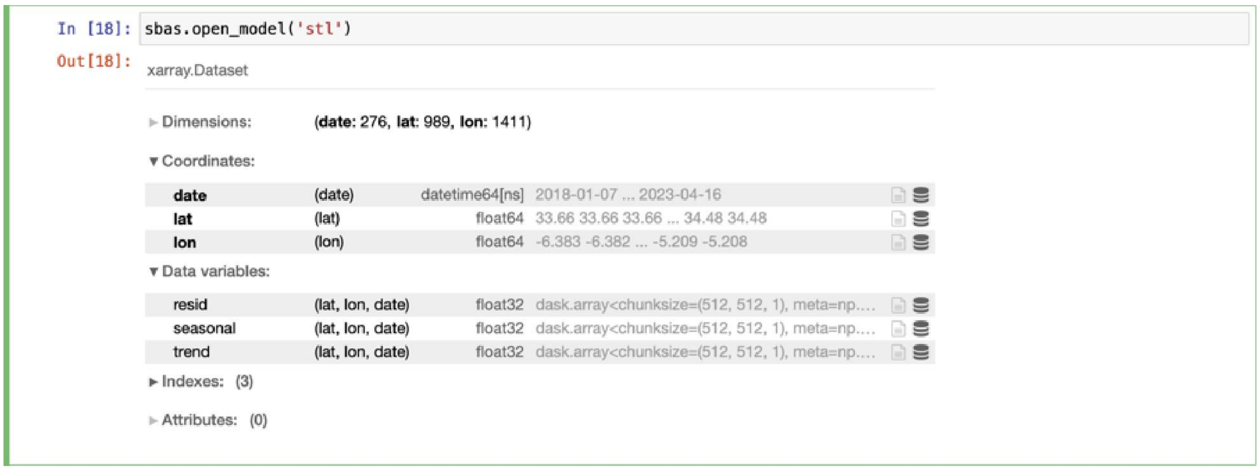
There are many helpful options within this function.

Reading PyGMTSAR 3D Grids

For trend detection results, we do not need the capability for incremental 2D grids processing, and the results can be saved to a single 3D dataset. For compatibility with Google Colab and other limited environments, SBAS output is currently saved to a set of 2D grids. Like 2D grids, the 3D grids are chunked for lazy parallel access and compressed to optimize disk consumption and read-write performance.

The function pygmtsar.SBAS.open_model performs read operations inside the PyGMTSAR working directory:

```
sbas.open_model('stl')
```



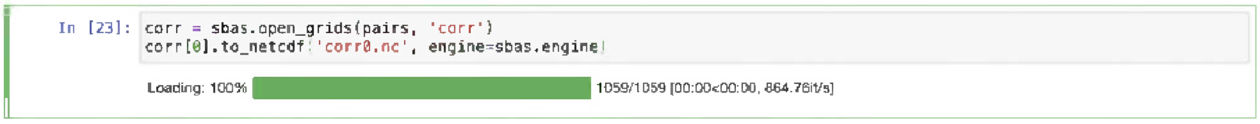
```
In [18]: sbas.open_model('stl')
Out[18]:
         xarray.Dataset

         ▸ Dimensions:      (date: 276, lat: 989, lon: 1411)
         ▾ Coordinates:
             date           (date)           datetime64[ns]  2018-01-07 ... 2023-04-16
             lat            (lat)            float64         33.66 33.66 33.66 ... 34.48 34.48
             lon            (lon)            float64         -6.383 -6.382 ... -5.209 -5.208
         ▾ Data variables:
             resid          (lat, lon, date) float32         dask.array<chunksize=(512, 512, 1), meta=np....
             seasonal       (lat, lon, date) float32         dask.array<chunksize=(512, 512, 1), meta=np....
             trend          (lat, lon, date) float32         dask.array<chunksize=(512, 512, 1), meta=np....
         ▸ Indexes:  (3)
         ▸ Attributes:  (0)
```

Reading and Writing NetCDF

You can open and save external NetCDF files outside the PyGMTSAR working directory using and xarray.to_netcdf functions.

To write a single 2D grid from the correlation stack, use the following command:

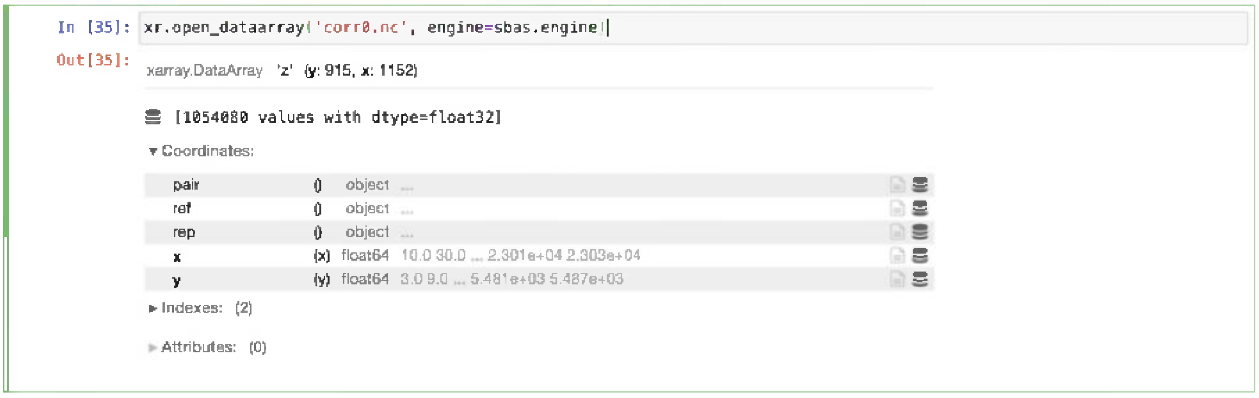
```
corr = sbas.open_grids(pairs, 'corr')
corr[0].to_netcdf(
    'corr0.nc',
    engine=sbas.engine
)
```



```
In [23]: corr = sbas.open_grids(pairs, 'corr')
         corr[0].to_netcdf('corr0.nc', engine=sbas.engine)

         Loading: 100% 1059/1059 [00:00<00:00, 864.76it/s]
```

To read an external NetCDF file or one created as described above, use:

```
xr.open_dataarray('corr0.nc', engine=sbas.engine)
```



```
In [35]: xr.open_dataarray('corr0.nc', engine=sbas.engine)
Out[35]:
xarray.DataArray  'z'  (y: 915, x: 1152)

[1054080 values with dtype=float32]

▼ Coordinates:
   pair      ()    object  ...
   ref       ()    object  ...
   rep       ()    object  ...
   x         (x)   float64 10.0 30.0 ... 2.301e+04 2.303e+04
   y         (y)   float64 3.0 9.0 ... 5.481e+03 5.487e+03
► Indexes:  (2)
► Attributes:  (0)
```

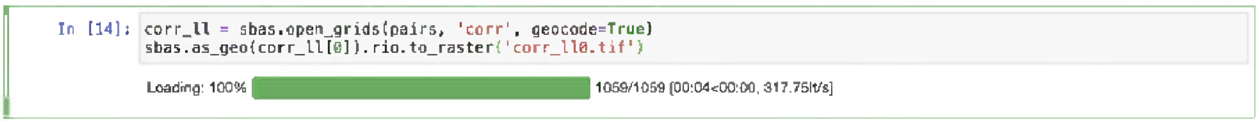
Numerous NetCDF handling libraries are available, but many are incompatible with Dask's lazy processing. Therefore, in PyGMTSAR, we specify the engine used via the designated option This ensures that the appropriate engine is utilized throughout the processing, promoting smooth and effective data handling.

Reading and Writing GeoTIFF

While GeoTIFF files are not optimal for lazy processing due to potential issues, they can still be imported and exported when needed. Well-prepared NetCDF files in geographic coordinates can be easily exported in GeoTIFF format using the pygmtsar.SBAS.as_geo function.

It’s worth noting that there’s no practical reason to export a GeoTIFF file in radar coordinates, as such files cannot be georeferenced. The example below illustrates how to save the first 2D grid from the geocoded correlation stack as a GeoTIFF file.

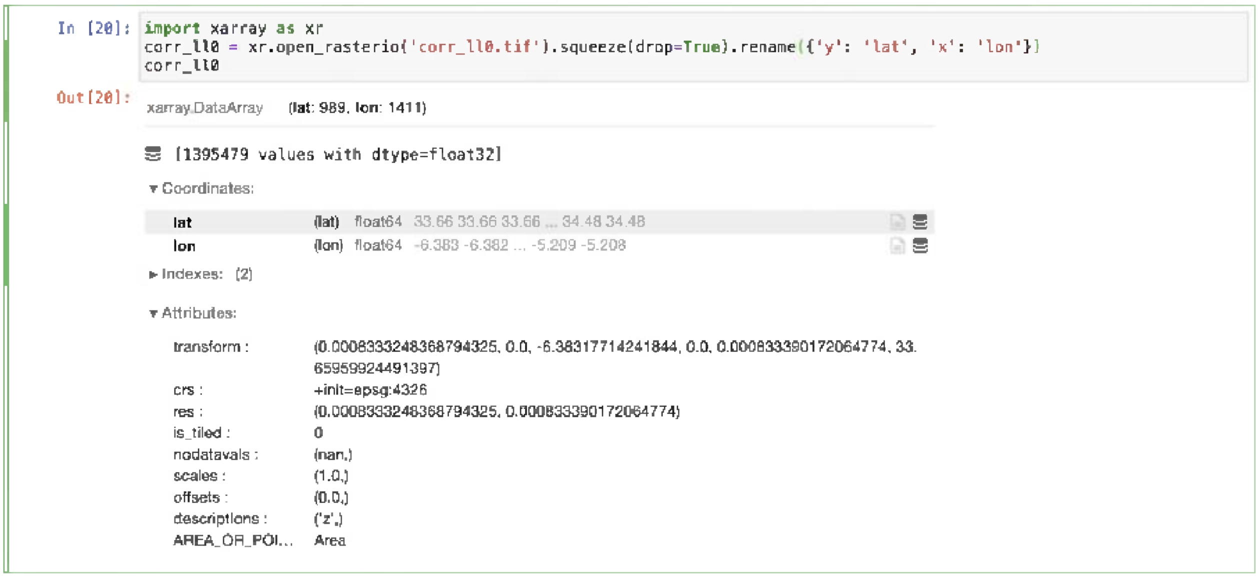
```
corr_ll = sbas.open_grids(
    pairs, 'corr', geocode=True
)
sbas.as_geo(corr_ll[0]).rio.to_raster(
    'corr_ll0.tif'
)
```



```
In [14]: corr_ll = sbas.open_grids(pairs, 'corr', geocode=True)
         sbas.as_geo(corr_ll[0]).rio.to_raster('corr_ll0.tif')

         Loading: 100%  1059/1059 [00:04<00:00, 317.75it/s]
```

To read an external GeoTIFF file or one created as described above, you can use the following code. Note that the data structure of an opened GeoTIFF file differs from that of a common NetCDF file, and as such, additional operations are required to convert it to a compatible format.

```
import xarray as xr
corr_ll0 = xr.open_rasterio('corr_ll0.tif') \
        .squeeze(drop=True) \
        .rename({'y': 'lat', 'x': 'lon'})
corr_ll0
```



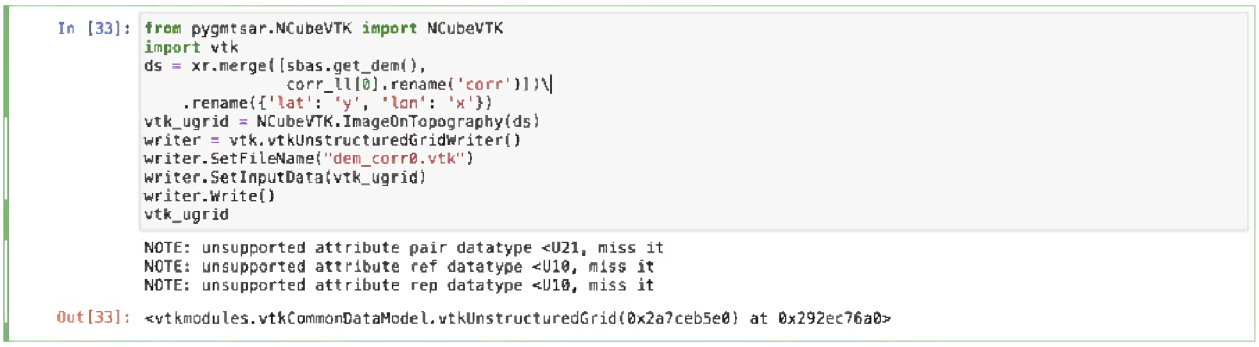
```
In [20]: import xarray as xr
         corr_ll0 = xr.open_rasterio('corr_ll0.tif').squeeze(drop=True).rename({'y': 'lat', 'x': 'lon'})
         corr_ll0

Out[20]: xarray.DataArray   (lat: 989, lon: 1411)

         [1395479 values with dtype=float32]
         ▾ Coordinates:
             lat              (lat)  float64  33.66 33.66 33.66 ... 34.48 34.48
             lon              (lon)  float64  -6.383 -6.382 ... -5.209 -5.208
         ▸ Indexes:  (2)
         ▾ Attributes:
             transform :      (0.0008333248368794325, 0.0, -6.38317714241844, 0.0, 0.0008333901720064774, 33.
                              65959924491397)
             crs :            +init=epsg:4326
             res :            (0.0008333248368794325, 0.0008333901720064774)
             is_tiled :       0
             nodatavals :     (nan,)
             scales :         (1.0,)
             offsets :        (0.0,)
             descriptions :   ('z',)
             AREA_OR_POI...   Area
```

Writing VTK

VTK format is excellent for 3D and 4D visualization and certain types of analysis in ParaView software. A single VTK file can include a DEM, a satellite image to be shown on top of DEM or on a plane, and 2D or 3D data grids. PyGMTSAR provides the pygmtsar.NCubeVTK.ImageOnTopography function for VTK data export. This function is defined in a separate class to allow for potential placement into a separate library since it is not directly related to PyGMTSAR. For now, it’s the only function available, and thus there’s no need to split the code.

We can use the vtk library itself, although it requires a few more lines of code:

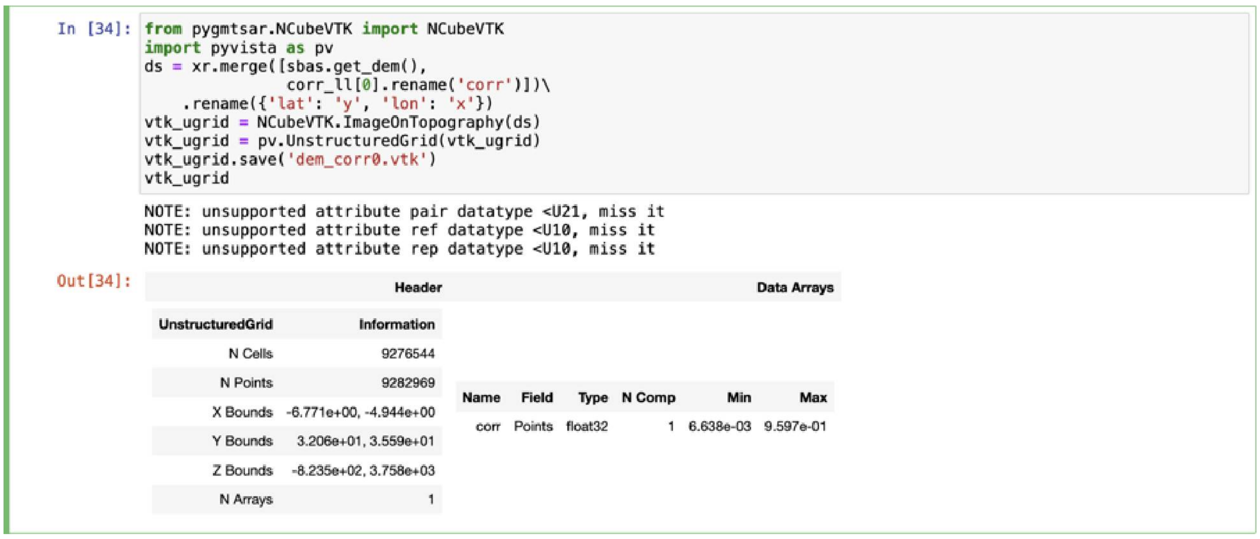
```
from pygmtsar.NCubeVTK import NCubeVTK
import vtk
ds = xr.merge(
   [sbas.get_dem(), corr_ll[0].rename('corr')]
).rename(
   {'lat': 'y', 'lon': 'x'}
)
vtk_ugrid = NCubeVTK.ImageOnTopography(ds)
writer = vtk.vtkUnstructuredGridWriter()
writer.SetFileName ("dem_corr0.vtk")
writer.SetInputData(vtk_ugrid)
writer.Write()
vtk_ugrid
```



```
In [33]: from pygmtsar.NCubeVTK import NCubeVTK
         import vtk
         ds = xr.merge([sbas.get_dem(),
                        corr_ll[0].rename('corr')])\
             .rename({'lat': 'y', 'lon': 'x'})
         vtk_ugrid = NCubeVTK.ImageOnTopography(ds)
         writer = vtk.vtkUnstructuredGridWriter()
         writer.SetFileName("dem_corr0.vtk")
         writer.SetInputData(vtk_ugrid)
         writer.Write()
         vtk_ugrid

         NOTE: unsupported attribute pair datatype <U21, miss it
         NOTE: unsupported attribute ref datatype <U10, miss it
         NOTE: unsupported attribute rep datatype <U10, miss it

Out[33]: <vtkmodules.vtkCommonDataModel.vtkUnstructuredGrid(0x2a7ceb5e0) at 0x292ec76a0>
```

With the helper library pyvista, the same code can be shorter, and the printed object structure is clearer:

```
from pygmtsar.NCubeVTK import NCubeVTK
import pyvista as pv
ds = xr.merge(
    [sbas.get_dem(), corr_ll[0].rename('corr')]
).rename(
    {'lat': 'y', 'lon': 'x'}
)
vtk_ugrid = NCubeVTK.ImageOnTopography(ds)
vtk_ugrid = pv.UnstructuredGrid(vtk_ugrid)
vtk_ugrid.save('dem_corr0.vtk')
vtk_ugrid
```



```
In [34]: from pygmtsar.NCubeVTK import NCubeVTK
         import pyvista as pv
         ds = xr.merge([sbas.get_dem(),
                        corr_ll[0].rename('corr')])\
             .rename({'lat': 'y', 'lon': 'x'})
         vtk_ugrid = NCubeVTK.ImageOnTopography(ds)
         vtk_ugrid = pv.UnstructuredGrid(vtk_ugrid)
         vtk_ugrid.save('dem_corr0.vtk')
         vtk_ugrid

         NOTE: unsupported attribute pair datatype <U21, miss it
         NOTE: unsupported attribute ref datatype <U10, miss it
         NOTE: unsupported attribute rep datatype <U10, miss it
Out[34]:
```

Header

UnstructuredGrid	Information
N Cells	9276544
N Points	9282969
X Bounds	-6.771e+00, -4.944e+00
Y Bounds	3.206e+01, 3.559e+01
Z Bounds	-8.235e+02, 3.758e+03
N Arrays	1

Data Arrays

Name	Field	Type	N Comp	Min	Max
corr	Points	float32	1	6.638e-03	9.597e-01

The output file can even be opened in a Jupyter notebook for interactive 3D and 4D visualizations.

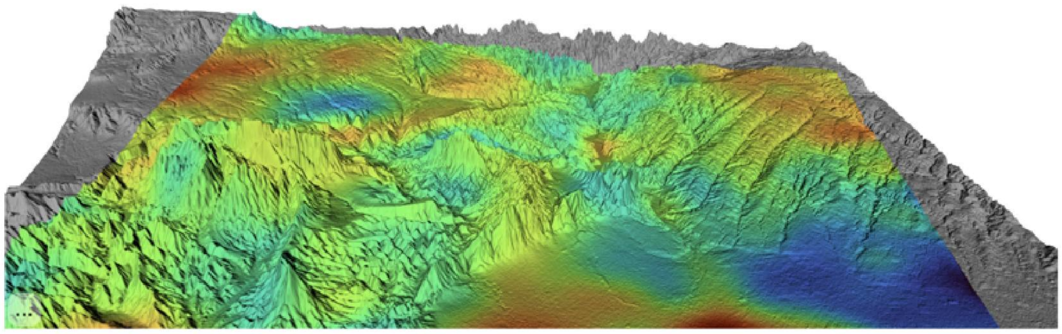
Note that there's a separate library, N-Cube ParaView plugin for 3D/4D GIS Data authored by the creator of PyGMTSAR and available on Github. This is designed for use within ParaView, and the codes can be easily adapted for other needs. As such, the modified code keeps the same authorship and can be used in PyGMTSAR without licensing issues.

The example below illustrates how an interactive 3D VTK map appears within a Jupyter notebook. The gray surface represents the topography, and the colored map overlaid on it displays the InSAR trend movement.

```
from pygmtsar.NCubeVTK import NCubeVTK
import pyvista as pv
import xarray as xr

pv.set_plot_theme("document")
```

```
stl = sbas.open_model('stl')
delta = stl.trend[...,0] - stl.trend[...,-1]
ds = xr.merge(
   [0.0001*dem, delta]
).rename(
   {'lat': 'y', 'lon': 'x'}
)
vtk_ugrid = NCubeVTK.ImageOnTopography(ds)

static_plotter = pv.Plotter(notebook=True)
static_plotter.add_mesh(
   vtk_ugrid,
   scalars='trend',
   cmap='turbo',
   clim=[-100, 100]
)



static_plotter.show(
   jupyter_backend="panel",
   return_viewer=True
)
```

```
In [52]: from pygmtsar.NCubeVTK import NCubeVTK
         import pyvista as pv
         import xarray as xr

         # magic trick for white background
         pv.set_plot_theme("document")

         stl = sbas.open_model('stl')
         delta = stl.trend[...,0] - stl.trend[...,-1]
         # scale for visualization
         ds = xr.merge([0.0001*dem, delta]).rename({'lat': 'y', 'lon': 'x'})
         vtk_ugrid = NCubeVTK.ImageOnTopography(ds)

         static_plotter = pv.Plotter(notebook=True)
         static_plotter.add_mesh(vtk_ugrid, scalars='trend', cmap='turbo', clim=[-100, 100])

         static_plotter.show(jupyter_backend="panel", return_viewer=True)

Out[52]:
```



3.5. InSAR Workflow Steps

PyGMTSAR implements the Small BAseline Subset (SBAS) InSAR time-series analysis for mapping ground deformation and offers a set of features beyond it. All the fundamental steps in InSAR processing are available, along with some advanced and modern machine learning algorithms to simplify your work on InSAR projects. Depending on the project, some steps can be omitted.

Although InSAR processing is quicker with just two images and a single interferogram, PyGMTSAR carries out the required steps in the same way as it would for SBAS processing with multiple interferograms. This approach offers numerous benefits, such as simplifying the process and providing unique features. First, it is more user-friendly to carry out the same procedure for either one or many interferograms. Additionally, it’s worth noting that technically, we can produce two interferograms from two images: one in the direct order of the dates and one in the inverse order. Having two interferograms derived from the same two images can be useful for comparing unwrapping accuracy - we would expect the same unwrapped results, up to a sign and aliquot 2π. Any discrepancies in this comparison can highlight potential unwrapping errors.

The complete workflow is outlined below. The most challenging projects can benefit significantly from having all processing steps readily available in one software, easily stackable to achieve the desired results.

Data This is the process of downloading two or more Sentinel-1 scenes and the related orbit files, along with downloading and preparing the topography covering the area.

Sentinel-1 SLC scenes can be obtained using known scene names from the Alaska Satellite Facility (ASF) data storage fast, or from the Sentinel usually slower. For both storages, user credentials are required. Typically, the Sentinel Hub is too slow for interactive notebooks, and for this reason, the ASF access point is used in the notebooks. The notebooks request credentials to download the data; these are not saved anywhere, so every user needs to register on ASF and enter their credentials to execute the notebooks. This authentication makes automated downloading impossible but allows you to replace the used scene names with your own and run the processing with ease.

For fully automated analysis, some of the example PyGMTSAR notebooks use pre-downloaded scenes shared on Apple's iCloud drive. While there are other well-known and user-friendly cloud storages like Google Drive or Dropbox, only iCloud provides direct links for automated downloading. Other storages require user interaction. These notebooks can be executed in a single click, but to replace the source scenes, you need to download the new ones on your computer, share them on iCloud drive, and copy the shared file link into the notebook.

Orbit files downloading is straightforward and automated. It's also possible to use pre-downloaded orbit files for offline processing. PyGMTSAR finds all existing orbit files in the same directory where the Sentinel-1 SLC scenes are located and downloads only missing ones, if any. Pay attention that orbit files may sometimes be updated, and it's better to download the recent ones than use old ones. If you still need to use outdated orbit files, they can produce a ramp in the produced interferograms — use PyGMTSAR detrending to remove it.

The function pygmtsar.SBAS.download_orbits is designed for the task of downloading orbit files.

DEM file can be automatically downloaded and pre-processed. If you use your own DEM file, be aware that common SRTM and other DEM data should be processed to remove the EGM96 geoid, making the heights relative to the WGS84 ellipsoid. For automatically downloaded DEMs, PyGMTSAR identifies the area covered by provided scenes and performs all the required pre-processing itself, converting the DEM to the defined resolution. Note: The output grid is exactly the same as the DEM grid.

The pygmtsar.SBAS.download_dem function streamlines the process of downloading the DEM.

Scene A set of sequential Sentinel-1 scenes in the same orbit can be stitched together. Conversely, a single scene comprising about 9 bursts can be cropped to one or more bursts for faster processing. This step can be performed before or after image co-registration. We generally do it before, to have the ability to export the stitched and cropped scenes identical to the original GeoTIFF files for reproducible analysis.

If you need to reframe scenes, the pygmtsar.SBAS.reframe_parallel function is at your service.

Image Co-Registration Images must be accurately co-registered, i.e., aligned with each other within a fraction of a pixel. Each SAR image has a unique radar coordinate system, defined by the satellite's path and position. For older satellites, alignment typically involves identifying common features in the two images using their amplitudes, and then aligning these features with each other. Modern Sentinel-1 satellites allow for pure geometrical co-registration using precise satellite orbits. This is achieved through inverse geocoding from a grid of reference points in geographic coordinates for the images, and determining the aligning transform between the master image and every repeating image using robust linear regression. The defined transform is then applied to convert all the images into the same master image radar coordinates, forming an image stack. This approach allows different orbits and polarizations to be analyzed separately and geocoded before the calculation of displacement components or polarimetric analysis.

The pygmtsar.SBAS.stack_parallel function is your primary tool for co-registering the image stack.

Note: While Sentinel-1 SLC scenes do provide a set of ground control points (GCP), their accuracy is quite low, approximately a couple of kilometers. Therefore, they are only suitable for mapping a scene boundary preview.

Radar Topography The conversion of a Digital Elevation Model (DEM) from a geodetic coordinate system (latitude/longitude) to a radar coordinate grid—a process known as radar coding or inverse geocoding—is an essential step in the processing pipeline. Initially, the DEM in the radar coordinate system can be used to simulate the radar phase for each pixel. This phase is related to the topography and viewing geometry of the radar system. This simulation forms a “flat earth” phase, which is subtracted from the interferogram to remove the topographic effect. As a result, it enables the study of deformation or other changes on the earth’s surface. Secondly, it allows the production of geocoding matrices to convert land masks and other geographic grids into radar coordinates, as well as converting InSAR results into geographic coordinates.

The pygmtsar.SBAS.topo_ra_parallel function is used to perform these radar topography calculations.

Interferogram The interferogram is formed by cross-multiplying the two complex SAR images and taking the phase of the result. This step requires the removal of the Doppler effect and topographic phase. The remaining phase difference at each point in the image is proportional to the change in travel time between the ground and the satellite along the line-of-sight direction. Under ideal circumstances, this travel time is linearly proportional when atmospheric conditions remain the same. However, real-world conditions such as atmospheric turbulence and clouds can affect it. Large movements are easiest to detect as they result in many so-called fringes. The application of a Gaussian filter with a default cut-off wavelength of 200m and modified adaptive Goldstein filters with a coherence-driven alpha parameter in a default window size of 32 pixels aids in better fringe shape detection. Be aware, though, that these significantly affect image resolution, and small objects and changes can be completely missed or biased. To reduce the output interferogram size, a user-defined function can be applied. The example notebooks use an averaging function that realizes noise-reducing multi-looking.

The pygmtsar.SBAS.intf_parallel function is available to facilitate your interferogram formation tasks.

Subswaths The separate processing of multiple Sentinel-1 subswaths should be followed by their merging into one grid after the interferogram formation. This step is not necessary for single subswath processing and can be omitted in such cases. In addition to merging subswaths, a merged radar topography grid is also produced.

For all your subswath merging needs, turn to the pygmtsar.SBAS.merge_parallel function.

Direct and Inverse Direct geocoding refers to the transformation of an image from radar coordinates to geographic coordinates, while inverse geocoding involves transforming a grid from geographic coordinates to radar coordinates. PyGMTSAR enables fast on-the-fly geocoding, relying on two geocoding matrices (direct and inverse) for simple pixel permutations. This allows conversion of a geographic coordinate grid like a land mask to the radar coordinate grid for unwrapping, and vice versa for export.

For creating geocoding matrices, you can use the pygmtsar.SBAS.geocode_parallel function.

Note: It’s recommended to use the same resolutions for the interferogram and DEM for best accuracy because the conversion is a one-to-one process, where every radar coordinate pixel corresponds to a single geographic pixel. In cases where every DEM grid pixel includes a set of radar grid pixels, only one pixel is used, which effectively results in unwanted decimation. Conversely, when multiple DEM pixels correspond to a single radar coordinate pixel, the DEM spacing is too precise and could be reduced without loss of accuracy.

Phase The interferogram is a “wrapped” phase image, in the sense that phase values are cyclic with a period of 2π. Unwrapping this phase is the process of recovering the true phase values. This is achieved with the use of the SNAPHU binary tool. PyGMTSAR offers a user-friendly Python wrapper to define custom SNAPHU parameters, validate results, and process output messages. Both single-tile and tiled unwrapping methods are supported.

Use the pygmtsar.SBAS.unwrap_parallel function for the unwrapping process.

Phase The unwrapped phase is a complex mixture of ground deformations, atmospheric effects, and linear ramps due to orbit error, among other things. Due to the lack of information about the absolute phases on the source wrapped interferogram, the unwrapped phase is offset by an arbitrary number of 2π shifts. This means that the unwrapped phases, in their raw form, are not suitable for displacement calculations. The process of detrending is required to remove the unwanted components from the phase to isolate ground deformation. PyGMTSAR offers two ways to perform detrending: in the spatial domain and the spatial frequency domain. The well-established spatial domain detrending is based on linear regression to fit and remove coordinates and topography and the product of coordinates and topography. Frequency domain detrending employs band-pass Gaussian filtering to achieve similar goals. Detrending can be omitted when observing significant surface movement and atmospheric effects are relatively small.

Use the pygmtsar.SBAS.detrend_parallel function for your detrending requirements.

Line-of-Sight (LOS) Displacement The unwrapped phase can be converted into a displacement map, which demonstrates the ground deformation occurring between the two radar image captures. This transformation is usually expressed in terms of the change in distance along the satellite's line-of-sight (LOS). This conversion can be applied to both radar coordinates and geographic coordinate grids.

Use the pygmtsar.SBAS.los_displacement_mm function to calculate LOS displacement in millimeters from the continuous phase given in radians.

Displacement The LOS displacements calculated are tied to satellite positions and are not comparable between two orbits. To compute vertical and east-west displacement components, the incidence angle grid is computed and known conversion formulas are used. It is important to note that PyGMTSAR uses the complete grid, not just the average incidence angle. Due to the geometry of the Sentinel-1 satellites' orbits, the calculation of the north-south component is significantly less accurate and therefore not provided. By default, PyGMTSAR calculates the projections in geographic coordinates, suggesting that these steps should be performed at the end of an InSAR analysis.

Make use of the pygmtsar.SBAS.vertical_displacement_mm and pygmtsar.SBAS.eastwest_displacement_mm functions to compute projections in millimeters from continuous phase in radians.

Time-Series When more than two radar images are available for the same location, interferograms can be generated for many image pairs (by date) to form overlapping displacement maps. To determine the most probable deformation time series, PyGMTSAR supports both correlation-weighted and non-weighted least squares calculations. The result is a displacement map for each image (by date), derived from the displacements of the interferogram pairs (by date pairs). This processing greatly minimizes atmospheric delay effects, which are neutralized in sequential pairs of interferograms.

For detecting displacement time series, utilize the pygmtsar.SBAS.sbas_parallel or pygmtsar.SBAS.lstsq_parallel functions.

Trend Decomposing seasonal and long-term trends enables the detection of phenomena like seasonal aquifer movements, long-term changes, and more. PyGMTSAR employs the widely-used Seasonal-Trend decomposition using LOESS (STL) technique, renowned for its robustness in detecting trends amidst noisy data. Even when detrending and SBAS analyses combined are unable to produce clean displacement time series, STL decomposition is typically successful. Note that this feature is not demonstrated in the example Jupyter notebooks on Google Colab.

Rely on the pygmtsar.SBAS.stl_parallel function for robust trend decomposition.

NetCDF and VTK Upon completion, interferograms, displacement maps, and time series can be exported to NetCDF format. This format is compatible with common GIS software such as QGIS and GDAL. It’s worth noting that NetCDF effectively replaces GeoTIFF files in terms of processing efficiency, rendering the generation of GeoTIFF files unnecessary, although it remains an option. Additionally, VTK export is supported to create files for 3D and 4D analysis and visualizations. However, this feature isn't demonstrated in the example Jupyter notebooks on Google Colab.

To expedite your export process, utilize the xarray.to_netcdf and pygmtsar.NCubeVTK.ImageOnTopography functions. For displaying progress indicators during NetCDF exporting, consider using the

The steps listed above serve as key functional units in constructing a comprehensive InSAR processing pipeline. Each step is designed to be highly customizable, allowing you to seamlessly integrate your unique code as necessary to tailor the process to your specific project requirements. The results generated can be exported in widely compatible formats such as NetCDF and VTK, enabling post-processing and review in GIS software like QGIS. Moreover, dynamic 3D visualization of the results can be achieved using tools like ParaView. This streamlined and adaptable process ensures that PyGMTSAR is widely applicable across diverse InSAR projects.

Troubleshooting and FAQs

In this chapter, we'll address some common issues and questions users might encounter while using PyGMTSAR. This section is not exhaustive and primarily serves as a starting point for troubleshooting. For more in-depth inquiries, the project's GitHub Issues page is a valuable resource, particularly the resolved (closed) tickets. PyGMTSAR is complex software as it implements a multitude of modern and well-established algorithms using advanced numerical calculations and machine learning approaches across various environments.

Most of the time, PyGMTSAR works flawlessly—almost like magic. However, on rare occasions, the black magic of technical bugs can affect us. These issues can be related to platform-specific changes, such as backward-incompatible upgrades on Google Colab, as experienced in early 2023. In such instances, some time may be required to identify and resolve platform-related issues. This is all part of the ongoing commitment to maintaining and improving the functionality and usability of PyGMTSAR.

Q: I’m having trouble running PyGMTSAR on Google Colab. What should I do?

A: First, ensure you select the default Google Colab environment. GPU-powered environments can be limited in RAM and disk space. Ensure your internet connection is stable, as interruptions can cause issues.

For more specific problems, the GitHub Issues page is a useful resource.

Q: I’m having trouble running PyGMTSAR in a Docker container. What should I do?

A: Check if your Docker environment meets all the prerequisites and that you’ve followed the Docker installation and setup instructions correctly. Ensure your Docker container has adequate resources (RAM, CPU, storage) to run PyGMTSAR. If the problem persists, consider seeking assistance from the GitHub Issues page.

Q: I’m having trouble running PyGMTSAR installed on my computer. What should I do?

A: Ensure you have installed and properly configured the latest versions of GMTSAR, SNAPHU, and PyGMTSAR software. Verify that all dependencies are installed correctly and are compatible with PyGMTSAR.

Up-to-date installation scripts are available in the project’s GitHub continuous integration (CI) tests directory for MacOS and Linux Ubuntu. Additionally, the Dockerfiles directory contains the latest installation scripts for Linux Ubuntu.

If you continue to experience problems, consider seeking help on the project’s GitHub Issues page.

Q: I’m experiencing errors during processing. What should I do?

A: PyGMTSAR generates error messages to help diagnose issues. Read any error messages you receive during processing carefully. They often include information that can aid in identifying the problem. Also, PyGMTSAR’s parallel functions come with non-parallel counterparts and arguments for enabling debug output. If you can’t resolve the issue, search for the error message on the project’s GitHub Issues page to see if others have encountered and solved the same issue.

Q: What should I do if I find a bug in PyGMTSAR?

A: If you believe you’ve found a bug in PyGMTSAR, please report it via the project’s GitHub Issues page. Be sure to provide a detailed description of the issue, steps to reproduce it, and any error messages you received. This will help the maintainer and community address the issue more effectively.

Q: How can I stay updated with changes or new features in PyGMTSAR?

A: Google Colab examples install the latest PyGMTSAR version for every run, ensuring you always have access to the most recent features. Docker images, on the other hand, provide a well-tested and stable environment, so there might not be any new features available, and the existing examples work exactly the same for every run. You can modify this pre-defined behavior if needed. Set the exact PyGMTSAR library version to be installed in Google Colab notebooks for a fixed and stable environment, or upgrade the pre-installed PyGMTSAR version in Docker containers to access the most recent features.

Q: Where can I find the most recent information about PyGMTSAR?

A: PyGMTSAR maintains various channels for updates, catering to different interests:

For Developers: If you're interested in code updates or wish to contribute, visit the PyGMTSAR project on This is the hub for development and issue tracking, providing the latest updates on code modifications and project enhancements.
For Real-World Applications: If you're curious about how PyGMTSAR is used in real-world applications, follow the author on The posts often contain updates on PyGMTSAR applications and related satellite interferometry developments.
For Exclusive Insights: If you're looking for detailed insights and case studies on PyGMTSAR and satellite interferometry, consider subscribing to the authors' It provides in-depth content not yet widely published, granting you early access to PyGMTSAR insights, case studies, and more.

Remember to check back frequently, as the information in these sources is updated regularly.

Q: Can I use PyGMTSAR for my commercial projects?

A: Yes, PyGMTSAR can be used for commercial projects. However, always ensure you comply with the terms of the license under which PyGMTSAR is distributed.

Q: Can I embed PyGMTSAR in my commercial software?

A: Yes, you can. The PyGMTSAR Python library is licensed under the BSD License (BSD-3-Clause), which is a permissive license. This allows you to use the library in commercial software provided that you adhere to the conditions of the BSD-3-Clause license. Please consult the terms of this license for more details.

Q: Where can I find commercial support and order additional features development?

A: You can reach out to the author of PyGMTSAR on Upwork for commercial support and additional feature development. This platform provides a structured environment for hiring professionals and managing projects.

Q: Is the PyGMTSAR code free of any potential conflicts of interest?

A: Yes, the PyGMTSAR Python library is developed by a single developer and does not incorporate any third-party code. Initially, the PyGMTSAR Github repository was cloned from the open-source GMTSAR This step was necessitated by modifications the author made to the GMTSAR project code to enable its integration with PyGMTSAR. As it stands, these changes have all been merged into the GMTSAR repository, allowing PyGMTSAR to work seamlessly with the upstream GMTSAR tools. For the sake of reliability, PyGMTSAR continues to maintain the modified GMTSAR sources in a separate branch.

Moreover, GMTSAR incorporates open-source SNAPHU project meaning that an installation of GMTSAR also provides access to SNAPHU tools. In its operations, PyGMTSAR calls GMTSAR and SNAPHU binaries as separate processes, in line with the respective project licenses.

PyGMTSAR relies on numerous Python numerical processing and scientific libraries. These libraries have permissive licenses that are compatible with PyGMTSAR's license and allow for this type of usage. As is standard practice, all the external libraries used are listed in PyGMTSAR's setup file.

To sum up, the source code of PyGMTSAR directly originates from the author. The repository's commit history validates that every line of code in PyGMTSAR has been written by the developer. As soon as PyGMTSAR becomes self-sufficient with all necessary replacements for GMTSAR tools, it will be moved to an independent repository.

This guide details PyGMTSAR internals, offering theoretical knowledge and practical tips for mastering InSAR principles. It uncovers the underlying patterns of core InSAR algorithms and their implementations, aiding you in extracting valuable information even from the most challenging projects. It serves as your bridge between academic sciences, high-performance computing, and real-world InSAR cases. [Planned]

About the Author

My name is Aleksei (Alexey) Pechnikov, a data scientist and software engineer with a Master's degree in Radio Physics and Electronics. I have specialized in forward and inverse modeling for non-linear optics, interferometry, and holography. My work in these areas earned me the first prize in the All-Russian Physics competition in 2004.

With over 20 years of experience, I have engaged in various projects for government agencies, universities, and multinational corporations like LG Corp and Google Inc. I also have teaching experience at the university level, including postgraduate students.

For many years now, I have been living in Thailand with my family, enjoying the country while pursuing my professional endeavors.

I publish articles and posts on LinkedIn, Medium, and Patreon. These platforms allow me to connect with readers and fellow professionals for discussions on concepts, findings, and collaboration opportunities.

For over 9 years, I have been successfully completing projects on Upwork, a renowned freelance platform, working on a variety of challenging scientific and industrial projects. I have also provided long-term support for some of them. My work is characterized by its complexity, efficiency, excellent communication, and effective time management.

Connect with me on the following platforms:

YouTube
GitHub
DockerHub

LinkedIn
Medium
Patreon
Upwork