PyGMT

Release v0.5.0

unknown

Nov 03, 2021

GETTING STARTED

A beautiful map is worth a thousand words. To truly understand how powerful PyGMT is, play with it online on Binder! But if you need some convincing first, watch this 1 hour introduction to PyGMT!

Afterwards, feel free to look at our Tutorials or visit the PyGMT Gallery.

PyGMT is a Python mapping and plotting tool.

It's powered by the Generic Mapping Tools (**GMT**) — these are *command-line tools*.

CHAPTER

ONE

ABOUT

PyGMT is a library for processing geospatial and geophysical data and making publication quality maps and figures. It provides a Pythonic interface for the Generic Mapping Tools (GMT), a command-line program widely used in the Earth Sciences.

We rely heavily on new features that have been implemented in GMT 6.0. In particular, a new *modern execution mode* that greatly simplifies figure creation. **These features are not available in the 5.4 version of GMT**.

CHAPTER TWO

PROJECT GOALS

- Make GMT more accessible to new users.
- Build a Pythonic API for GMT.
- Interface with the GMT C API directly using ctypes (no system calls).
- Support for rich display in the Jupyter notebook.
- Integration with the PyData ecosystem: `numpy.ndarray` or `pandas.DataFrame` for data tables and `xarray.DataArray` for grids.

CHAPTER

THREE

CONTACTING US

- Most discussion happens on GitHub. Feel free to open an issue or comment on any open issue or pull request.
- We have a Discourse forum where you can ask questions and leave comments.

CHAPTER

FOUR

CONTRIBUTING

4.1 Code of conduct

Please note that this project is released with a Contributor Code of Conduct. By participating in this project you agree to abide by its terms.

4.2 Contributing Guidelines

Please read our Contributing Guide to see how you can help and give feedback.

4.3 Imposter syndrome disclaimer

We want your help. No, really.

There may be a little voice inside your head that is telling you that you're not ready to be an open source contributor; that your skills aren't nearly good enough to contribute. What could you possibly offer?

We assure you that the little voice in your head is wrong.

Being a contributor doesn't just mean writing code. Equally important contributions include: writing or proof-reading documentation, suggesting or implementing tests, or even giving feedback about the project (including giving feedback about the contribution process). If you're coming to the project with fresh eyes, you might see the errors and assumptions that seasoned contributors have glossed over. If you can write any code at all, you can contribute code to open source. We are constantly trying out new skills, making mistakes, and learning from those mistakes. That's how we all improve and we are happy to help others learn.

This disclaimer was adapted from the MetPy project.

CITING PYGMT

PyGMT is a community developed project. See the AUTHORS.md file on GitHub for a list of the people involved and a definition of the term "PyGMT Developers". Feel free to cite our work in your research using the following BibTeX:

```
@software{pygmt_2021_5607255,
  author       = {Uieda, Leonardo and
                  Tian, Dongdong and
                  Leong, Wei Ji and
                  Jones, Meghan and
                  Schlitzer, William and
                  Toney, Liam and
                  Grund, Michael and
                  Yao, Jiayuan and
                  Magen, Yohai and
                  Materna, Kathryn and
                  Newton, Tyler and
                  Anant, Abhishek and
                  Ziebarth, Malte and
                  Quinn, Jamie aand
                  Wessel, Paul},
  title        = {{PyGMT: A Python interface for the Generic Mapping Tools}},
  month        = oct,
  year         = 2021,
  publisher    = {Zenodo},
  version      = {v0.5.0},
  doi          = {10.5281/zenodo.5607255},
  url          = {https://doi.org/10.5281/zenodo.5607255}
}
```

To cite a specific version of PyGMT, go to our Zenodo page at https://doi.org/10.5281/zenodo.3781524 and use the "Export to BibTeX" function there. It is also strongly recommended to cite the GMT6 paper (which PyGMT wraps around). Note that some modules like `surface` and `x2sys` also have their dedicated citation. Further information for all these can be found at https://www.generic-mapping-tools.org/cite.

CHAPTER

SIX

LICENSE

PyGMT is free software: you can redistribute it and/or modify it under the terms of the **BSD 3-clause License**. A copy of this license is provided in LICENSE.txt.

CHAPTER SEVEN

SUPPORT

The development of PyGMT has been supported by NSF grants OCE-1558403 and EAR-1948603.

CHAPTER

EIGHT

RELATED PROJECTS

- GMT.jl: A Julia wrapper for GMT.
- gmtmex: A Matlab/Octave wrapper for GMT.

Other Python wrappers for GMT:

- gmtpy by Sebastian Heimann
- pygmt by Ian Rose
- PyGMT by Magnus Hagdorn

CHAPTER

NINE

COMPATIBILITY WITH GMT/PYTHON/NUMPY VERSIONS

PyGMT Version	Documentation	GMT	Python	Numpy
Dev (upcoming release)	Dev Documentation (reflects main branch)	>=6.2.0	>=3.7	>=1.18
v0.5.0 (latest release)	v0.5.0 Documentation	>=6.2.0	>=3.7	>=1.18
v0.4.1	v0.4.1 Documentation	>=6.2.0	>=3.7	>=1.17
v0.4.0	v0.4.0 Documentation	>=6.2.0	>=3.7	>=1.17
v0.3.1	v0.3.1 Documentation	>=6.1.1	>=3.7	
v0.3.0	v0.3.0 Documentation	>=6.1.1	>=3.7	
v0.2.1	v0.2.1 Documentation	>=6.1.1	>=3.6	
v0.2.0	v0.2.0 Documentation	>=6.1.1	3.6 - 3.8	
v0.1.2	v0.1.2 Documentation	>=6.0.0	3.6 - 3.8	
v0.1.1	v0.1.1 Documentation	>=6.0.0	3.6 - 3.8	
v0.1.0	v0.1.0 Documentation	>=6.0.0	3.6 - 3.8	

9.1 Overview

9.1.1 About

PyGMT is a Python wrapper for the Generic Mapping Tools (GMT), a command-line program widely used in the Earth Sciences. It provides capabilities for processing spatial data (gridding, filtering, masking, FFTs, etc) and making high quality plots and maps.

PyGMT is different from Python libraries like Bokeh and Matplotlib, which have a larger focus on interactivity and allowing different backends. GMT uses the PostScript format to generate high quality (static) vector graphics for publications, posters, talks, etc. It is memory efficient and very fast. The PostScript figures can be converted to other formats like PDF, PNG, and JPG for use on the web and elsewhere. In fact, PyGMT users will usually not have any contact with the original PostScript files and get only the more convenient formats like PDF and PNG.

The project was started in 2017 by Leonardo Uieda and Paul Wessel (the co-creator and main developer of GMT) at the University of Hawaii at Manoa. The development of PyGMT has been supported by NSF grants OCE-1558403 and EAR-1948603.

9.1.2 Presentations

These are conference presentations about the development of PyGMT (previously "GMT/Python"):

- "Remote Online Sessions for Emerging Seismologists (ROSES): Unit 8 - PyGMT". 2020. Liam Toney. Presented at *ROSES 2020*. url: https://www.iris.edu/hq/inclass/lesson/728

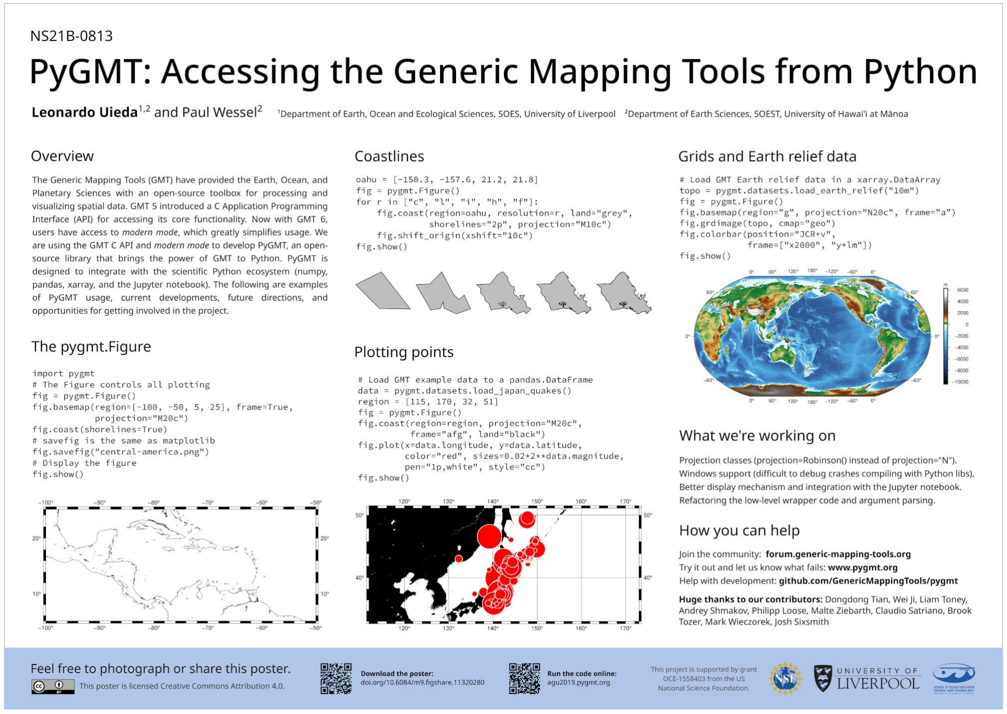
ROSES Unit 8:
PyGMT

Liam Toney

Geophysical Institute,
University of Alaska Fairbanks

11 August 2020

- "PyGMT: Accessing the Generic Mapping Tools from Python". 2019. Leonardo Uieda and Paul Wessel. Presented at *AGU 2019*. doi:10.6084/m9.figshare.11320280



NS21B-0813

PyGMT: Accessing the Generic Mapping Tools from Python

Leonardo Uieda[1,2] and Paul Wessel[2] [1]Department of Earth, Ocean and Ecological Sciences, SOES, University of Liverpool [2]Department of Earth Sciences, SOEST, University of Hawai'i at Mānoa

Overview

The Generic Mapping Tools (GMT) have provided the Earth, Ocean, and Planetary Sciences with an open-source toolbox for processing and visualizing spatial data. GMT 5 introduced a C Application Programming Interface (API) for accessing its core functionality. Now with GMT 6, users have access to *modern mode*, which greatly simplifies usage. We are using the GMT C API and *modern mode* to develop PyGMT, an open-source library that brings the power of GMT to Python. PyGMT is designed to integrate with the scientific Python ecosystem (numpy, pandas, xarray, and the Jupyter notebook). The following are examples of PyGMT usage, current developments, future directions, and opportunities for getting involved in the project.

The pygmt.Figure

```
import pygmt
# The Figure controls all plotting
fig = pygmt.Figure()
fig.basemap(region=[-100, -50, 5, 25], frame=True,
            projection="M20c")
fig.coast(shorelines=True)
# savefig is the same as matplotlib
fig.savefig("central-america.png")
# Display the figure
fig.show()
```

Coastlines

```
oahu = [-158.3, -157.6, 21.2, 21.8]
fig = pygmt.Figure()
for r in ["c", "l", "i", "h", "f"]:
    fig.coast(region=oahu, resolution=r, land="grey",
              shorelines="2p", projection="M10c")
    fig.shift_origin(xshift="10c")
fig.show()
```

Plotting points

```
# Load GMT example data to a pandas.DataFrame
data = pygmt.datasets.load_japan_quakes()
region = [115, 170, 32, 51]
fig = pygmt.Figure()
fig.coast(region=region, projection="M20c",
          frame="afg", land="black")
fig.plot(x=data.longitude, y=data.latitude,
         color="red", sizes=0.02*2**data.magnitude,
         pen="1p,white", style="cc")
fig.show()
```

Grids and Earth relief data

```
# Load GMT Earth relief data in a xarray.DataArray
topo = pygmt.datasets.load_earth_relief("10m")
fig = pygmt.Figure()
fig.basemap(region="g", projection="N20c", frame="a")
fig.grdimage(topo, cmap="geo")
fig.colorbar(position="JCR+v",
             frame=["x2000", "y+lm"])
fig.show()
```

What we're working on

Projection classes (projection=Robinson() instead of projection="N").
Windows support (difficult to debug crashes compiling with Python libs).
Better display mechanism and integration with the Jupyter notebook.
Refactoring the low-level wrapper code and argument parsing.

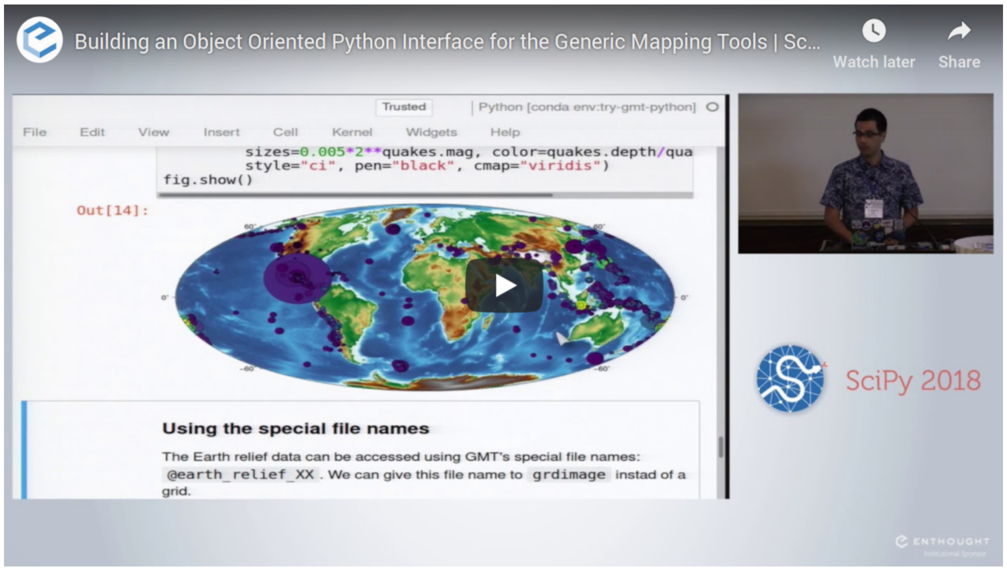
How you can help

Join the community: **forum.generic-mapping-tools.org**
Try it out and let us know what fails: **www.pygmt.org**
Help with development: **github.com/GenericMappingTools/pygmt**

Huge thanks to our contributors: Dongdong Tian, Wei Ji, Liam Toney, Andrey Shmakov, Philipp Loose, Malte Ziebarth, Claudio Satriano, Brook Tozer, Mark Wieczorek, Josh Sixsmith

Feel free to photograph or share this poster.
This poster is licensed Creative Commons Attribution 4.0.
Download the poster: doi.org/10.6084/m9.figshare.11320280
Run the code online: agu2019.pygmt.org
This project is supported by grant OCE-1558403 from the US National Science Foundation.

- "Building an object-oriented Python interface for the Generic Mapping Tools". 2018. Leonardo Uieda and Paul Wessel. Presented at *SciPy 2018*. doi:10.6084/m9.figshare.6814052


Building an Object Oriented Python Interface for the Generic Mapping Tools | Sc...
Watch later
Share
Trusted
Python [conda env:try-gmt-python]
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
SciPy 2018
ENTHOUGHT

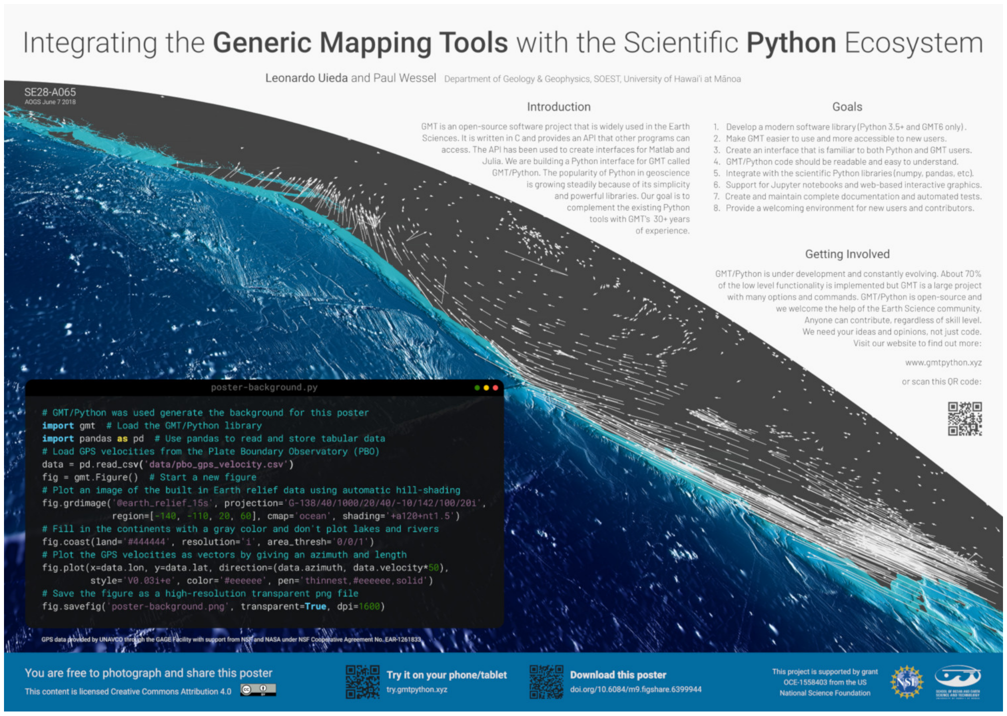
```
    sizes=0.005*2**quakes.mag, color=quakes.depth/qua
    style="ci", pen="black", cmap="viridis")
fig.show()
```

Out[14]:

Using the special file names

The Earth relief data can be accessed using GMT's special file names: `@earth_relief_XX`. We can give this file name to `grdimage` instad of a grid.

- “Integrating the Generic Mapping Tools with the Scientific Python Ecosystem”. 2018. Leonardo Uieda and Paul Wessel. Presented at *AOGS Annual Meeting 2018*. doi:10.6084/m9.figshare.6399944


SE28-A065
AOGS June 7 2018

Integrating the **Generic Mapping Tools** with the Scientific **Python** Ecosystem

Leonardo Uieda and Paul Wessel Department of Geology & Geophysics, SOEST, University of Hawai'i at Mānoa

Introduction

GMT is an open-source software project that is widely used in the Earth Sciences. It is written in C and provides an API that other programs can access. The API has been used to create interfaces for Matlab and Julia. We are building a Python interface for GMT called GMT/Python. The popularity of Python in geoscience is growing steadily because of its simplicity and powerful libraries. Our goal is to complement the existing Python tools with GMT's 30+ years of experience.

Goals

1. Develop a modern software library (Python 3.5+ and GMT6 only).
2. Make GMT easier to use and more accessible to new users.
3. Create an interface that is familiar to both Python and GMT users.
4. GMT/Python code should be readable and easy to understand.
5. Integrate with the scientific Python libraries (numpy, pandas, etc).
6. Support for Jupyter notebooks and web-based interactive graphics.
7. Create and maintain complete documentation and automated tests.
8. Provide a welcoming environment for new users and contributors.

Getting Involved

GMT/Python is under development and constantly evolving. About 70% of the low level functionality is implemented but GMT is a large project with many options and commands. GMT/Python is open-source and we welcome the help of the Earth Science community. Anyone can contribute, regardless of skill level. We need your ideas and opinions, not just code. Visit our website to find out more:

www.gmtpython.xyz

or scan this QR code:

poster-background.py

```
# GMT/Python was used generate the background for this poster
import gmt  # Load the GMT/Python library
import pandas as pd  # Use pandas to read and store tabular data
# Load GPS velocities from the Plate Boundary Observatory (PBO)
data = pd.read_csv('data/pbo_gps_velocity.csv')
fig = gmt.Figure()  # Start a new figure
# Plot an image of the built in Earth relief data using automatic hill-shading
fig.grdimage('@earth_relief_15s', projection='G-138/40/1000/20/40/-10/142/100/20i',
             region=[-140, -110, 20, 60], cmap='ocean', shading='+a120+nt1.5')
# Fill in the continents with a gray color and don't plot lakes and rivers
fig.coast(land='#444444', resolution='i', area_thresh='0/0/1')
# Plot the GPS velocities as vectors by giving an azimuth and length
fig.plot(x=data.lon, y=data.lat, direction=(data.azimuth, data.velocity*50),
         style='V0.03i+e', color='#eeeeee', pen='thinnest,#eeeeee,solid')
# Save the figure as a high-resolution transparent png file
fig.savefig('poster-background.png', transparent=True, dpi=1600)
```

GPS data provided by UNAVCO through the GAGE Facility with support from NSF and NASA under NSF Cooperative Agreement No. EAR-1261833

You are free to photograph and share this poster
This content is licensed Creative Commons Attribution 4.0

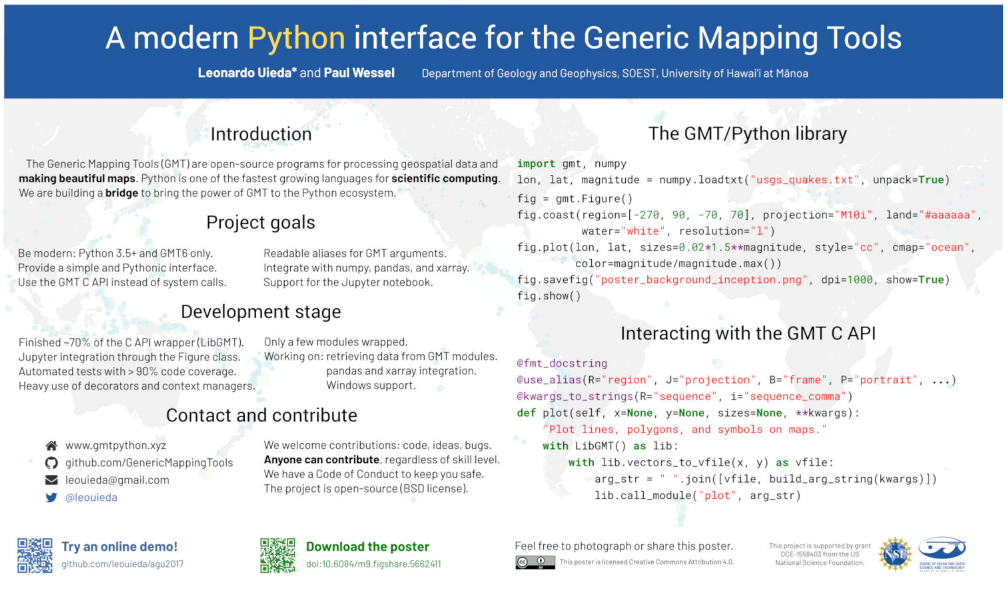
Try it on your phone/tablet
try.gmtpython.xyz

Download this poster
doi.org/10.6084/m9.figshare.6399944

This project is supported by grant OCE-1558403 from the US National Science Foundation

- “Bringing the Generic Mapping Tools to Python”. 2017. Leonardo Uieda and Paul Wessel. Presented at *SciPy 2017*. doi:10.6084/m9.figshare.7635833
- “A modern Python interface for the Generic Mapping Tools”. 2017. Leonardo Uieda and Paul Wessel. Presented at *AGU 2017*. doi:10.6084/m9.figshare.5662411

Bringing the Generic Mapping Tools to Python | SciPy 2017 | Leonardo Uieda & ...
Watch later
Share
Aloha
WELCOME TO HAWAII
SciPy 2017



A modern Python interface for the Generic Mapping Tools

Leonardo Uieda* and **Paul Wessel** Department of Geology and Geophysics, SOEST, University of Hawai'i at Mānoa

Introduction

The Generic Mapping Tools (GMT) are open-source programs for processing geospatial data and **making beautiful maps**. Python is one of the fastest growing languages for **scientific computing**. We are building a **bridge** to bring the power of GMT to the Python ecosystem.

Project goals

Be modern: Python 3.5+ and GMT6 only.
Provide a simple and Pythonic interface.
Use the GMT C API instead of system calls.

Readable aliases for GMT arguments.
Integrate with numpy, pandas, and xarray.
Support for the Jupyter notebook.

Development stage

Finished ~70% of the C API wrapper (LibGMT).
Jupyter integration through the Figure class.
Automated tests with > 90% code coverage.
Heavy use of decorators and context managers.

Only a few modules wrapped.
Working on: retrieving data from GMT modules.
pandas and xarray integration.
Windows support.

Contact and contribute

www.gmtpython.xyz
github.com/GenericMappingTools
leouieda@gmail.com
@leouieda

We welcome contributions: code, ideas, bugs.
Anyone can contribute, regardless of skill level.
We have a Code of Conduct to keep you safe.
The project is open-source (BSD license).

The GMT/Python library

```
import gmt, numpy
lon, lat, magnitude = numpy.loadtxt("usgs_quakes.txt", unpack=True)
fig = gmt.Figure()
fig.coast(region=[-270, 90, -70, 70], projection="M10i", land="#aaaaaa",
          water="white", resolution="l")
fig.plot(lon, lat, sizes=0.02*1.5**magnitude, style="cc", cmap="ocean",
         color=magnitude/magnitude.max())
fig.savefig("poster_background_inception.png", dpi=1000, show=True)
fig.show()
```

Interacting with the GMT C API

```
@fmt_docstring
@use_alias(R="region", J="projection", B="frame", P="portrait", ...)
@kwargs_to_strings(R="sequence", i="sequence_comma")
def plot(self, x=None, y=None, sizes=None, **kwargs):
    "Plot lines, polygons, and symbols on maps."
    with LibGMT() as lib:
        with lib.vectors_to_vfile(x, y) as vfile:
            arg_str = " ".join([vfile, build_arg_string(kwargs)])
            lib.call_module("plot", arg_str)
```

Try an online demo!
github.com/leouieda/agu2017

Download the poster
doi:10.6084/m9.figshare.5662411

Feel free to photograph or share this poster.
This poster is licensed Creative Commons Attribution 4.0.

This project is supported by grant OCE-1558403 from the US National Science Foundation.

9.2 Installing

Note: This package is in the early stages of design and implementation.

We welcome any feedback and ideas! Let us know by submitting issues on GitHub or by posting on our Discourse forum.

9.2.1 Quickstart

The fastest way to install PyGMT is with the conda package manager which takes care of setting up a virtual environment, as well as the installation of GMT and all the dependencies PyGMT depends on:

```
conda create --name pygmt --channel conda-forge pygmt
```

To activate the virtual environment, you can do:

```
conda activate pygmt
```

After this, check that everything works by running the following in a Python interpreter (e.g., in a Jupyter notebook):

```
import pygmt
pygmt.show_versions()
```

You are now ready to make you first figure! Start by looking at the tutorials on our sidebar, good luck!

Note: The sections below provide more detailed, step by step instructions to install and test PyGMT for those who may have a slightly different setup or want to install the latest development version.

9.2.2 Which Python?

PyGMT is tested to run on **Python 3.7 or greater**.

We recommend using the Anaconda Python distribution to ensure you have all dependencies installed and the conda package manager is available. Installing Anaconda does not require administrative rights to your computer and doesn't interfere with any other Python installations on your system.

9.2.3 Which GMT?

PyGMT requires Generic Mapping Tools (GMT) version 6 as a minimum, which is the latest released version that can be found at the GMT official site. We need the latest GMT (>=6.2.0) since there are many changes being made to GMT itself in response to the development of PyGMT, mainly the new modern execution mode.

Compiled conda packages of GMT for Linux, macOS and Windows are provided through conda-forge. Advanced users can also build GMT from source instead, which is not so recommended but we would love to get feedback from anyone who tries.

We recommend following the instructions further on to install GMT 6.

9.2.4 Dependencies

PyGMT requires the following libraries to be installed:

- numpy (>= 1.18)
- pandas
- xarray
- netCDF4
- packaging

The following are optional dependencies:

- IPython: For embedding the figures in Jupyter notebooks (recommended).
- GeoPandas: For using and plotting GeoDataFrame objects.

9.2.5 Installing GMT and other dependencies

Before installing PyGMT, we must install GMT itself along with the other dependencies. The easiest way to do this is via the `conda` package manager. We recommend working in an isolated conda environment to avoid issues with conflicting versions of dependencies.

First, we must configure conda to get packages from the conda-forge channel:

```
conda config --prepend channels conda-forge
```

Now we can create a new conda environment with Python and all our dependencies installed (we'll call it `pygmt` but feel free to change it to whatever you want):

```
conda create --name pygmt python=3.9 numpy pandas xarray netcdf4 packaging gmt
```

Activate the environment by running the following (**do not forget this step!**):

```
conda activate pygmt
```

From now on, all commands will take place inside the conda virtual environment called `pygmt` and won't affect your default `base` installation.

9.2.6 Installing PyGMT

Now that you have GMT installed and your conda virtual environment activated, you can install PyGMT using any of the following methods:

Using conda (recommended)

This installs the latest stable release of PyGMT from conda-forge:

```
conda install pygmt
```

This upgrades the installed PyGMT version to be the latest stable release:

```
conda update pygmt
```

Using pip

This installs the latest stable release from PyPI:

```
pip install pygmt
```

Alternatively, you can install the latest development version from TestPyPI:

```
pip install --pre --index-url https://test.pypi.org/simple/ --extra-index-url https://
↪pypi.org/simple pygmt
```

To upgrade the installed stable release or development version to be the latest one, just add `--upgrade` to the corresponding command above.

Any of the above methods (conda/pip) should allow you to use the PyGMT package from Python.

9.2.7 Testing your install

To ensure that PyGMT and its dependencies are installed correctly, run the following in your Python interpreter:

```
import pygmt
pygmt.show_versions()

fig = pygmt.Figure()
fig.coast(region="g", frame=True, shorelines=1)
fig.show()
```

If you see a global map with shorelines, then you're all set.

9.2.8 Finding the GMT shared library

Sometimes, PyGMT will be unable to find the correct version of the GMT shared library (`libgmt`). This can happen if you have multiple versions of GMT installed.

You can tell PyGMT exactly where to look for `libgmt` by setting the `GMT_LIBRARY_PATH` environment variable. This should be set to the directory where `libgmt.so`, `libgmt.dylib` or `gmt.dll` can be found for Linux, macOS and Windows, respectively. e.g., on a command line, run:

```
# Linux/macOS
export GMT_LIBRARY_PATH=$HOME/anaconda3/envs/pygmt/lib
# Windows
set "GMT_LIBRARY_PATH=C:\Users\USERNAME\Anaconda3\envs\pygmt\Library\bin\"
```

9.3 Making your first figure

Welcome to PyGMT! Here we'll cover some of basic concepts, like creating simple figures and naming conventions.

9.3.1 Loading the library

All modules and figure generation is accessible from the *pygmt* top level package:

```
import pygmt
```

9.3.2 Creating figures

All figure generation in PyGMT is handled by the *pygmt.Figure* class. Start a new figure by creating an instance of this class:

```
fig = pygmt.Figure()
```

Add elements to the figure using its methods. For example, let's use *pygmt.Figure.basemap* to start the creation of a map. We'll use the `region` parameter to provide the longitude and latitude bounds, the `projection` parameter to set the projection to Mercator (**M**) and the map width to 15 cm, and the `frame` parameter to generate a frame with automatic tick and annotation spacings.

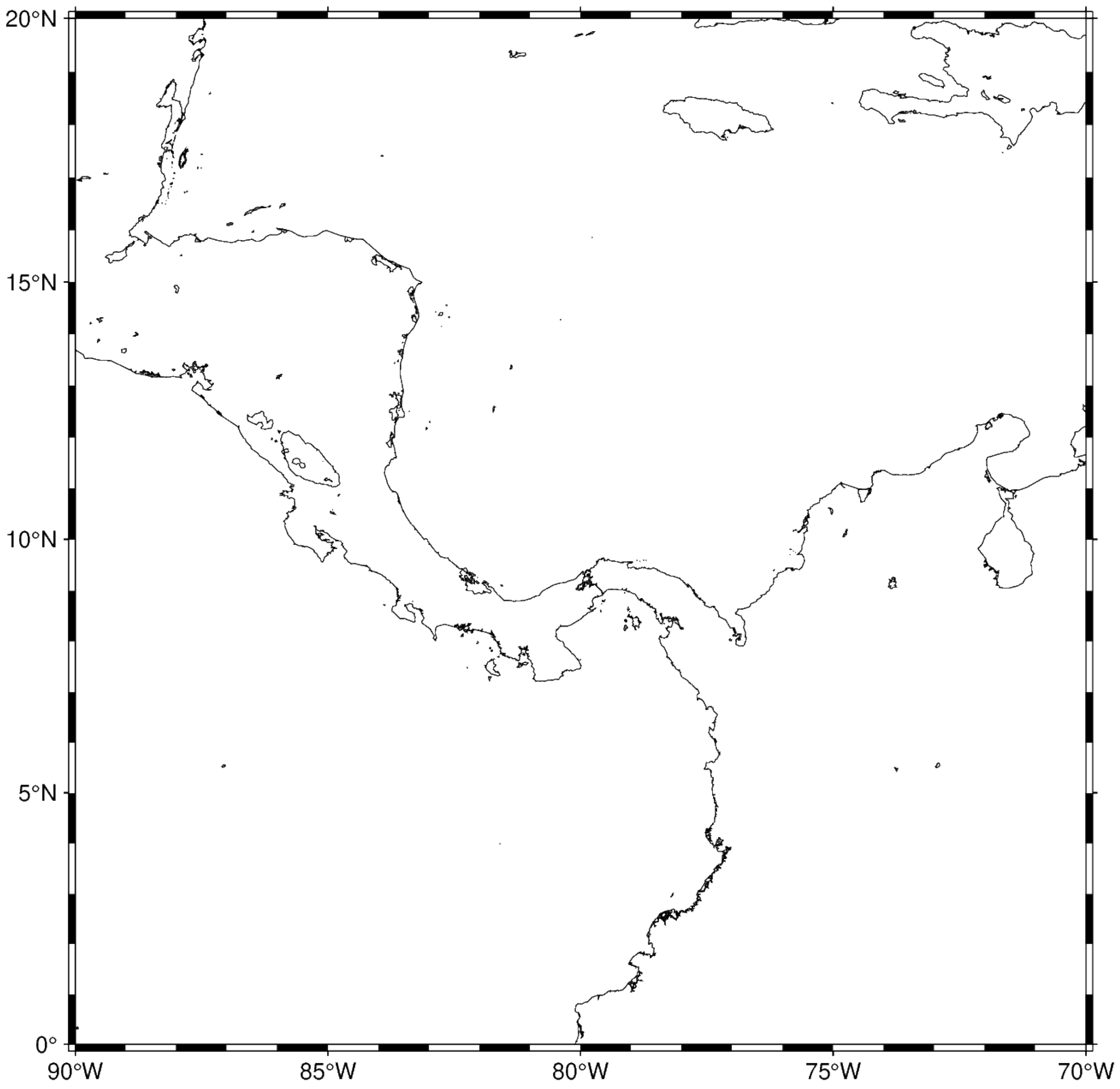
```
fig.basemap(region=[-90, -70, 0, 20], projection="M15c", frame=True)
```

Now we can add coastlines using *pygmt.Figure.coast* to this map using the default resolution, line width, and color:

```
fig.coast(shorelines=True)
```

To see the figure, call *pygmt.Figure.show*:

```
fig.show()
```

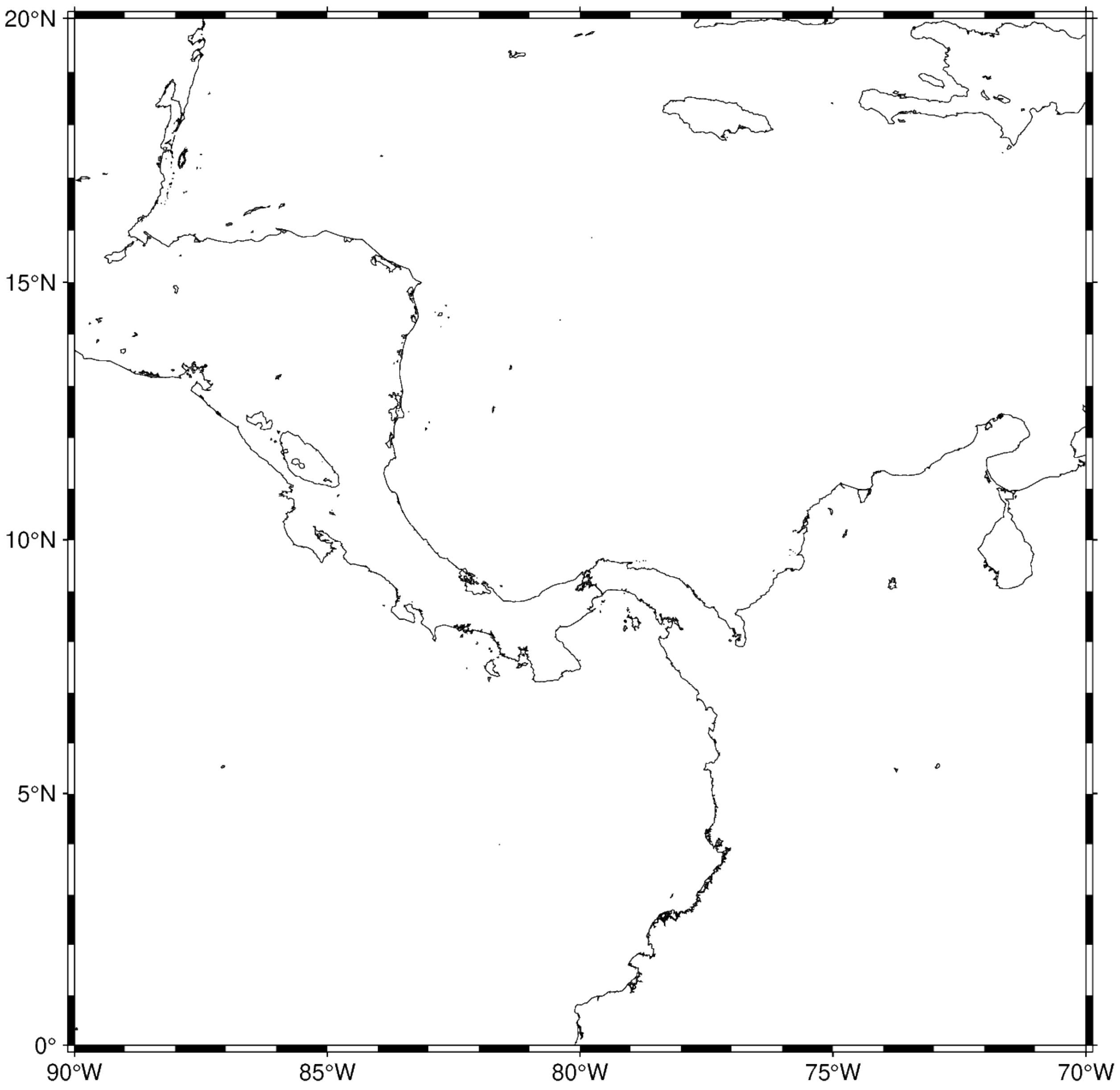


Out:

```
<IPython.core.display.Image object>
```

You can also set the map region, projection, and frame type directly in other methods without calling `gmt.Figure.basemap`:

```
fig = pygmt.Figure()
fig.coast(shorelines=True, region=[-90, -70, 0, 20], projection="M15c", frame=True)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

9.3.3 Saving figures

Use the method *pygmt.Figure.savefig* to save your figure to a file. The figure format is inferred from the extension.

```
fig.savefig("central-america-shorelines.png")
```

9.3.4 Note for experienced GMT users

You have probably noticed several things that are different from classic command-line GMT. Many of these changes reflect the new GMT modern execution mode that is part of GMT 6.

1. As a general rule, the `ps` prefix has been removed from all `ps*` modules (PyGMT methods). For example, the name of the GMT 5 module `pscoast` is `coast` in GMT 6 and PyGMT. The exceptions are: `psxy` which is now `plot`, `psxyz` which is now `plot3d`, and `psscale` which is now `colorbar`.
2. More details can be found in the GMT cookbook introduction to modern mode.

A few are PyGMT exclusive (like the `savefig` method).

1. The PyGMT parameters (called options or arguments in GMT) don't use the GMT 1-letter syntax (**R**, **J**, **B**, etc). We use longer aliases for these parameters and have some Python exclusive names. The mapping between the GMT parameters and their PyGMT aliases should be straightforward. For some modules, these aliases are still being developed.
2. Parameters like `region` can take `lists` as well as strings like `1/2/3/4`.
3. If a GMT option has no arguments (like `-B` instead of `-Baf`), use a `True` in Python. An empty string would also be acceptable. For repeated parameters, such as `-B+Loleron -Bxaf -By+lm`, provide a `list`: `frame=["+Loleron", "xaf", "y+lm"]`.
4. There is no output redirecting to a PostScript file. The figure is generated in the background and will only be shown or saved when you ask for it.

Total running time of the script: (0 minutes 4.768 seconds)

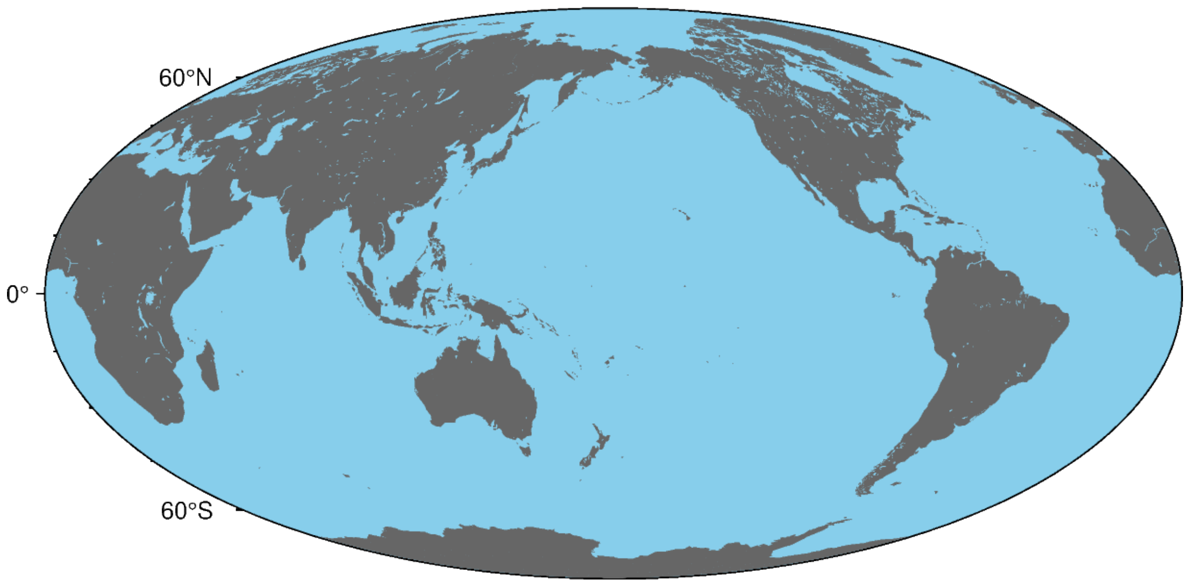
9.4 Gallery

This gallery contains examples of what PyGMT can do. Click on any example to see the code used to generate it.

9.4.1 Maps and map elements

Color land and water

The `land` and `water` parameters of *pygmt.Figure.coast* specify a color to fill in the land and water masses, respectively. There are many color codes in GMT, including standard GMT color names (like `skyblue`), R/G/B levels (like `0/0/255`), a hex value (like `#333333`), and a graylevel (like `50`).



```python
import pygmt

fig = pygmt.Figure()
# Make a global Mollweide map with automatic ticks
fig.basemap(region="g", projection="W15c", frame=True)
# Plot the land as light gray, and the water as sky blue
fig.coast(land="#666666", water="skyblue")
fig.show()
```

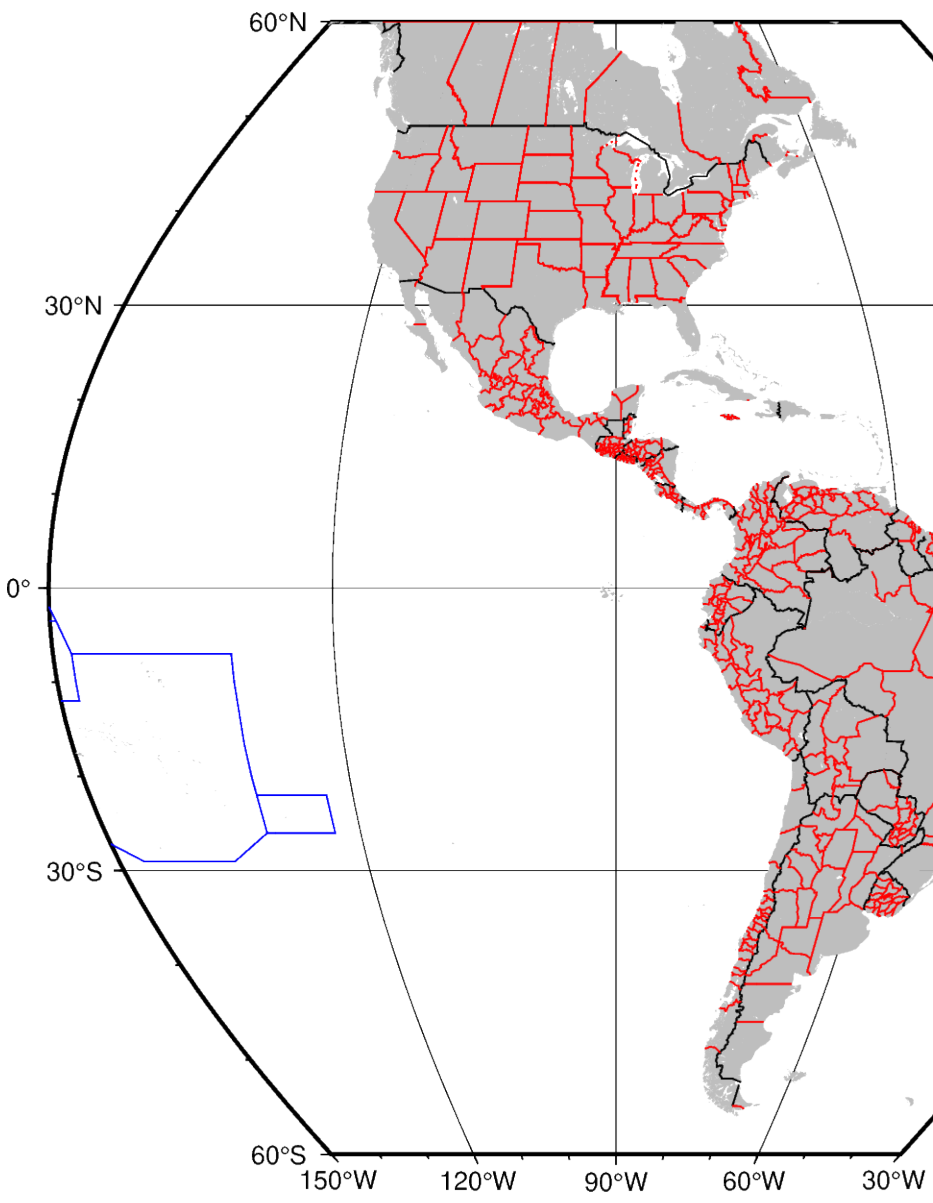
Total running time of the script: (0 minutes 3.687 seconds)

Political Boundaries

The `borders` parameter of *`pygmt.Figure.coast`* specifies levels of political boundaries to plot and the pen used to draw them. Choose from the list of boundaries below:

- **1** = National boundaries
- **2** = State boundaries within the Americas
- **3** = Marine boundaries
- **a** = All boundaries (1-3)

For example, to draw national boundaries with a line thickness of 1p and black line color use `borders="1/1p,black"`. You can draw multiple boundaries by passing in a list to `borders`.

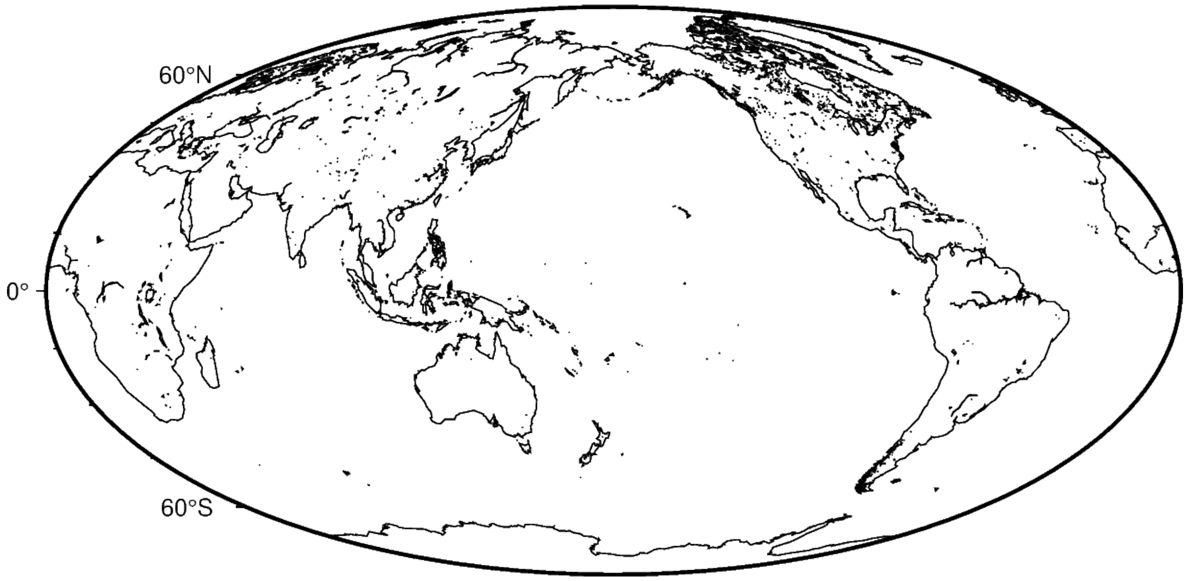


```python
import pygmt

fig = pygmt.Figure()
# Make a Sinusoidal projection map of the Americas with automatic annotations,
# ticks and gridlines
fig.basemap(region=[-150, -30, -60, 60], projection="I-90/15c", frame="afg")
# Plot each level of the boundaries dataset with a different color.
fig.coast(borders=["1/0.5p,black", "2/0.5p,red", "3/0.5p,blue"], land="gray")
fig.show()
```

Total running time of the script: (0 minutes 4.883 seconds)

Shorelines

Use *pygmt.Figure.coast* to display shorelines as black lines.



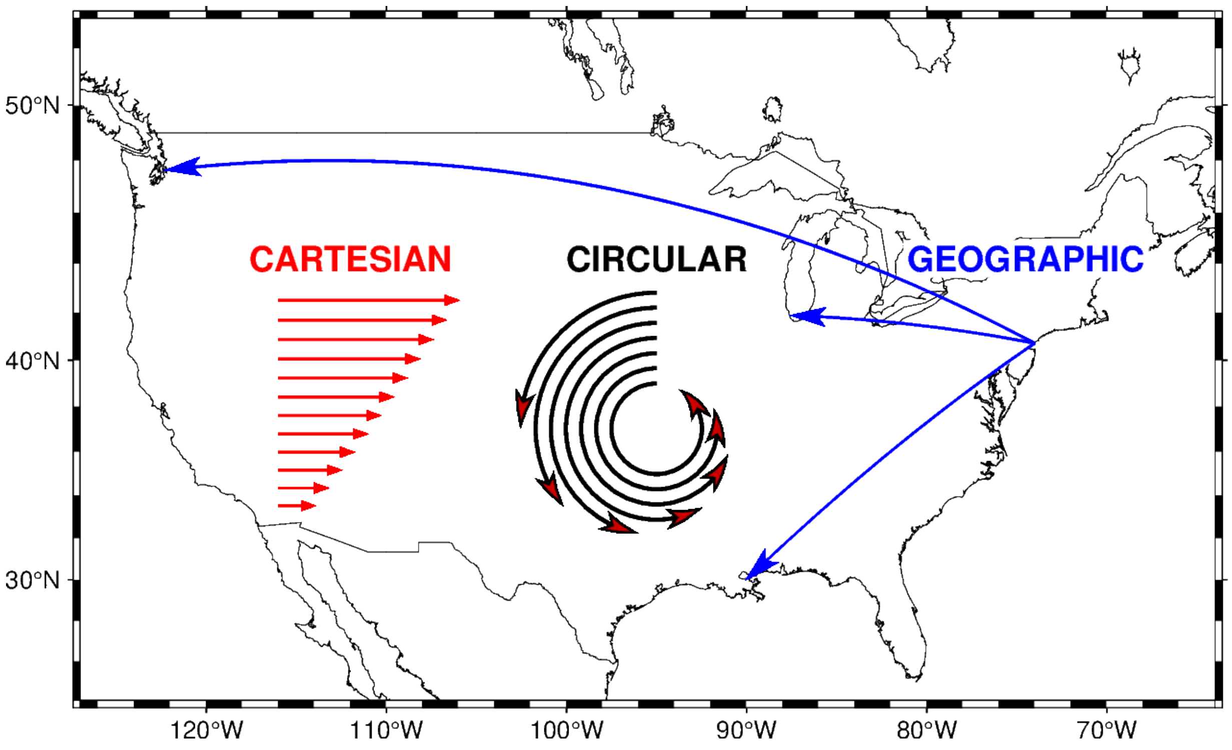
```
import pygmt

fig = pygmt.Figure()
# Make a global Mollweide map with automatic ticks
fig.basemap(region="g", projection="W15c", frame=True)
# Display the shorelines as black lines with 0.5 point thickness
fig.coast(shorelines="0.5p,black")
fig.show()
```

Total running time of the script: (0 minutes 3.529 seconds)

9.4.2 Lines and vectors

Cartesian, circular, and geographic vectors

The *pygmt.Figure.plot* method can plot Cartesian, circular, and geographic vectors. The `style` parameter controls vector attributes. See also *Vector attributes documentation*.



Out:

```
<IPython.core.display.Image object>
```

```
import numpy as np
import pygmt

# create a plot with coast, Mercator projection (M) over the continental US
fig = pygmt.Figure()
fig.coast(
    region=[-127, -64, 24, 53],
    projection="M15c",
    frame=True,
    borders=1,
    area_thresh=4000,
    shorelines="0.25p,black",
)


# Left: plot 12 Cartesian vectors with different lengths
x = np.linspace(-116, -116, 12)  # x vector coordinates
y = np.linspace(33.5, 42.5, 12)  # y vector coordinates
direction = np.zeros(x.shape)  # direction of vectors
length = np.linspace(0.5, 2.4, 12)  # length of vectors
# Cartesian vectors (v) with red pen and fill (+g, +p), vector head at
```

(continues on next page)

(continued from previous page)

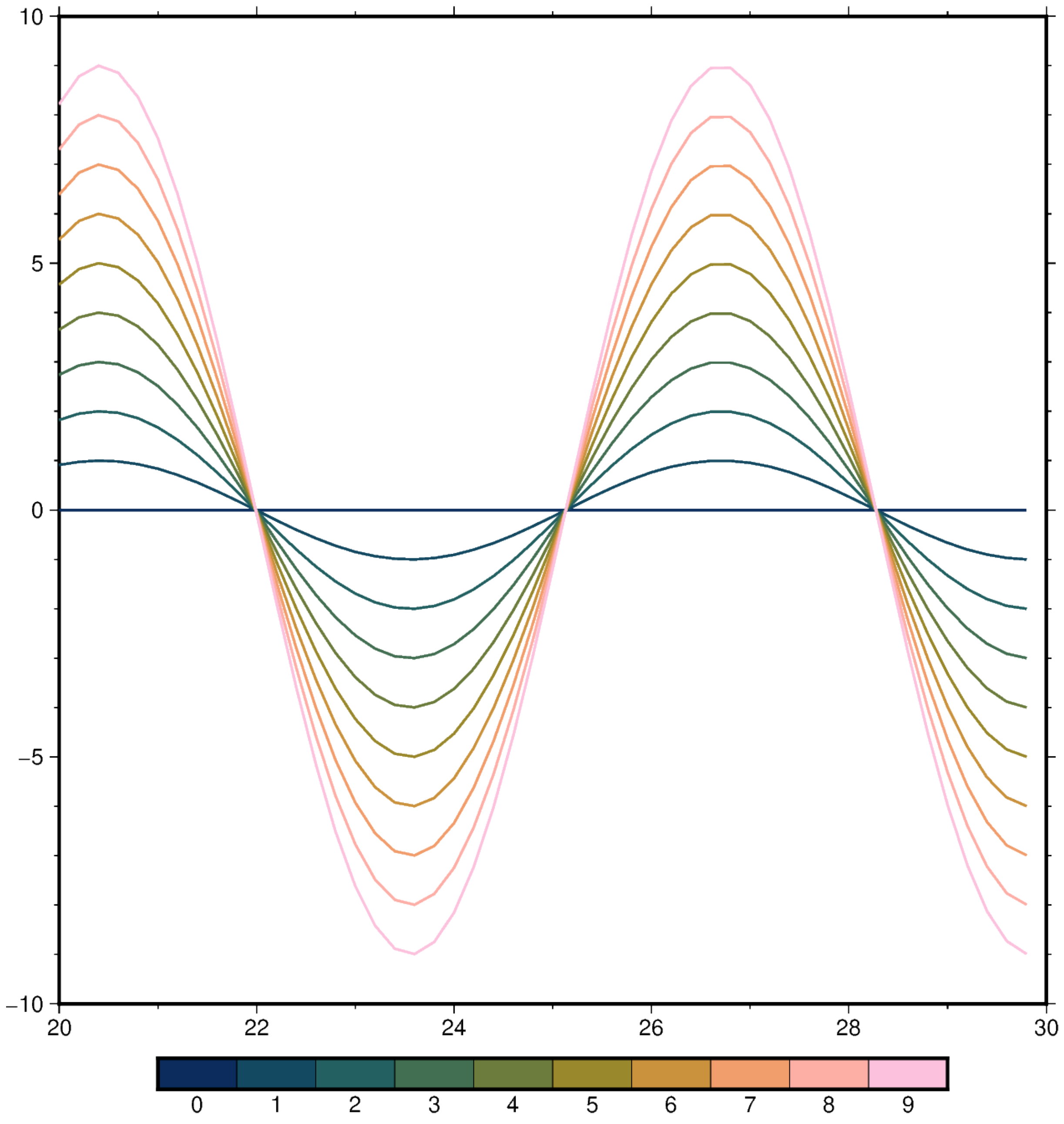
```
# end (+e), and 40 degree angle (+a) with no indentation for vector head (+h)
style = "v0.2c+e+a40+gred+h0+p1p,red"
fig.plot(x=x, y=y, style=style, pen="1p,red", direction=[direction, length])
fig.text(text="CARTESIAN", x=-112, y=44.2, font="13p,Helvetica-Bold,red", fill="white")


# Middle: plot 7 math angle arcs with different radii
num = 7
x = np.full(num, -95)  # x coordinates of the center
y = np.full(num, 37)  # y coordinates of the center
radius = 1.8 - 0.2 * np.arange(0, num)  # radius
startdir = np.full(num, 90)  # start direction in degrees
stopdir = 180 + 40 * np.arange(0, num)  # stop direction in degrees
# data for circular vectors
data = np.column_stack([x, y, radius, startdir, stopdir])
arcstyle = "m0.5c+ea"  # Circular vector (m) with an arrow at end
fig.plot(data=data, style=arcstyle, color="red3", pen="1.5p,black")
fig.text(text="CIRCULAR", x=-95, y=44.2, font="13p,Helvetica-Bold,black", fill="white")


# Right: plot geographic vectors using endpoints
NYC = [-74.0060, 40.7128]
CHI = [-87.6298, 41.8781]
SEA = [-122.3321, 47.6062]
NO = [-90.0715, 29.9511]
# `=` means geographic vectors.
# With the modifier '+s', the input data should contain coordinates of start
# and end points
style = "=0.5c+s+e+a30+gblue+h0.5+p1p,blue"
data = np.array([NYC + CHI, NYC + SEA, NYC + NO])
fig.plot(data=data, style=style, pen="1.0p,blue")
fig.text(
    text="GEOGRAPHIC", x=-74.5, y=44.2, font="13p,Helvetica-Bold,blue", fill="white"
)
fig.show()
```

Total running time of the script: (0 minutes 3.833 seconds)

Line colors with a custom CPT

The color of the lines made by `pygmt.Figure.plot` can be set according to a custom CPT and assigned with the `pen` parameter.

The custom CPT can be used by setting the plot command's `cmap` parameter to `True`. The `zvalue` parameter sets the z-value (color) to be used from the custom CPT, and the line color is set as the z-value by using **+z** when setting the `pen` color.



Out:

```
<IPython.core.display.Image object>
```

```python
import numpy as np
import pygmt

# Create a list of values between 20 and 30 with 0.2 intervals
```

(continues on next page)

(continued from previous page)

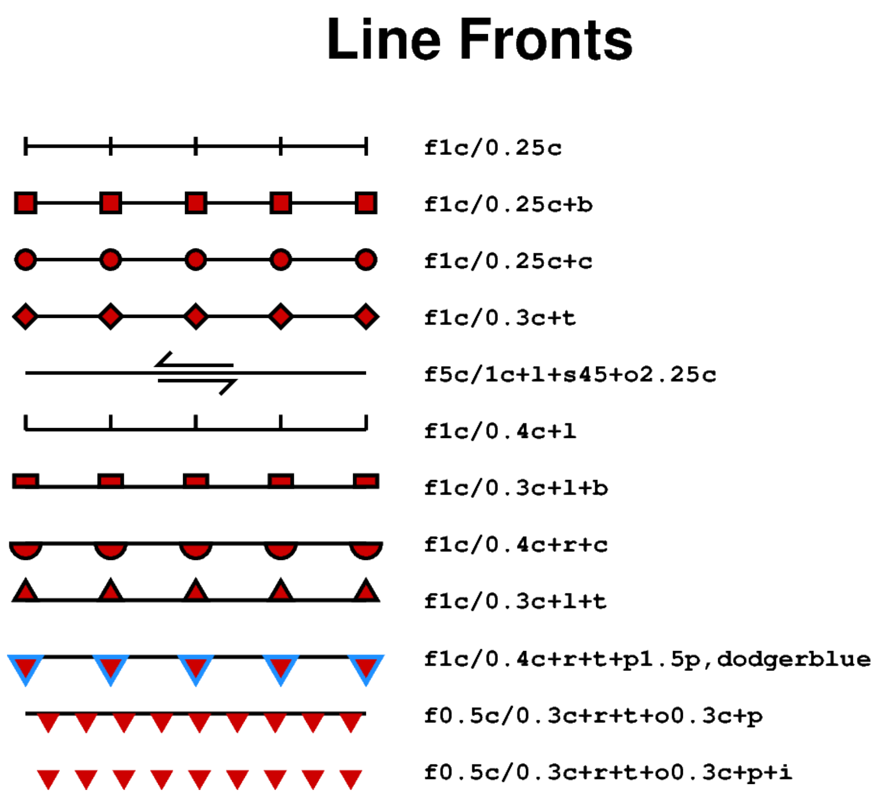
```python
x = np.arange(start=20, stop=30, step=0.2)

fig = pygmt.Figure()
fig.basemap(frame=["WSne", "af"], region=[20, 30, -10, 10])

# Create a custom CPT with the batlow CPT and 10 discrete z-values (colors),
# use color_model="+c0-9" to write the color palette in categorical format and
# add labels (0) to (9) for the colorbar legend
pygmt.makecpt(cmap="batlow", series=[0, 9, 1], color_model="+c0-9")

# Plot 10 lines and set a different z-value for each line
for zvalue in range(0, 10):
    y = zvalue * np.sin(x)
    fig.plot(x=x, y=y, cmap=True, zvalue=zvalue, pen="thick,+z,-")

# Color bar to show the custom CPT and the associated z-values
fig.colorbar()
fig.show()
```

Total running time of the script: (0 minutes 5.616 seconds)

Line fronts

Using the `pygmt.Figure.plot` method you can draw a so-called *front* which allows to plot specific symbols distributed along a line or curve. Typical use cases are weather fronts, fault lines, subduction zones, and more.

A front can be drawn by passing **f**[±]*gap*[/*size*] to the `style` parameter where *gap* defines the distance gap between the symbols and *size* the symbol size. If *gap* is negative, it is interpreted to mean the number of symbols along the front instead. If *gap* has a leading + then we use the value exactly as given [Default will start and end each line with a symbol, hence the *gap* is adjusted to fit]. If *size* is missing it is set to 30% of the *gap*, except when *gap* is negative and *size* is thus required. Append **+l** or **+r** to plot symbols on the left or right side of the front [Default is centered]. Append **+***type* to specify which symbol to plot: **b**ox, **c**ircle, **f**ault (default), **s**lip, or **t**riangle. Slip means left-lateral or right-lateral strike-slip arrows (centered is not an option). The **+s** modifier optionally accepts the angle used to draw the vector (default is 20). Alternatively, use **+S** which draws arcuate arrow heads. Append **+o***offset* to offset the first symbol from the beginning of the front by that amount (default is 0). The chosen symbol is drawn with the same pen as set for the line (i.e., via the `pen` parameter). To use an alternate pen, append **+p***pen*. To skip the outline, just use **+p** with no argument. To make the main front line invisible, add **+i**.



Out:

```
<IPython.core.display.Image object>
```

```python
import numpy as np
import pygmt

# Generate a two-point line for plotting
x = np.array([1, 4])
y = np.array([20, 20])

fig = pygmt.Figure()
fig.basemap(region=[0, 10, 0, 20], projection="X15c/15c", frame='+t"Line Fronts"')

# Plot the line using different front styles
for frontstyle in [
    # line with "faults" front style, same as +f (default)
    "f1c/0.25c",
    # line with box front style
    "f1c/0.25c+b",
    # line with circle front style
    "f1c/0.25c+c",
    # line with triangle front style
    "f1c/0.3c+t",
```

(continues on next page)

(continued from previous page)

```
    # line with left-lateral ("+l") slip ("+s") front style, angle is set to 45
    # and offset to 2.25 cm
    "f5c/1c+l+s45+o2.25c",
    # line with "faults" front style, symbols are plotted on the left side of
    # the front
    "f1c/0.4c+l",
    # line with box front style, symbols are plotted on the left side of the
    # front
    "f1c/0.3c+l+b",
    # line with circle front style, symbols are plotted on the right side of
    # the front
    "f1c/0.4c+r+c",
    # line with triangle front style, symbols are plotted on the left side of
    # the front
    "f1c/0.3c+l+t",
    # line with triangle front style, symbols are plotted on the right side of
    # the front, use other pen for the outline of the symbol
    "f1c/0.4c+r+t+p1.5p,dodgerblue",
    # line with triangle front style, symbols are plotted on the right side of
    # the front and offset is set to 0.3 cm, skip the outline
    "f0.5c/0.3c+r+t+o0.3c+p",
    # line with triangle front style, symbols are plotted on the right side of
    # the front and offset is set to 0.3 cm, skip the outline and make the main
    # front line invisible
    "f0.5c/0.3c+r+t+o0.3c+p+i",
]:
    y -= 1  # move the current line down
    fig.plot(x=x, y=y, pen="1.25p", style=frontstyle, color="red3")
    fig.text(
        x=x[-1],
        y=y[-1],
        text=frontstyle,
        font="Courier-Bold",
        justify="ML",
        offset="0.75c/0c",
    )

fig.show()
```

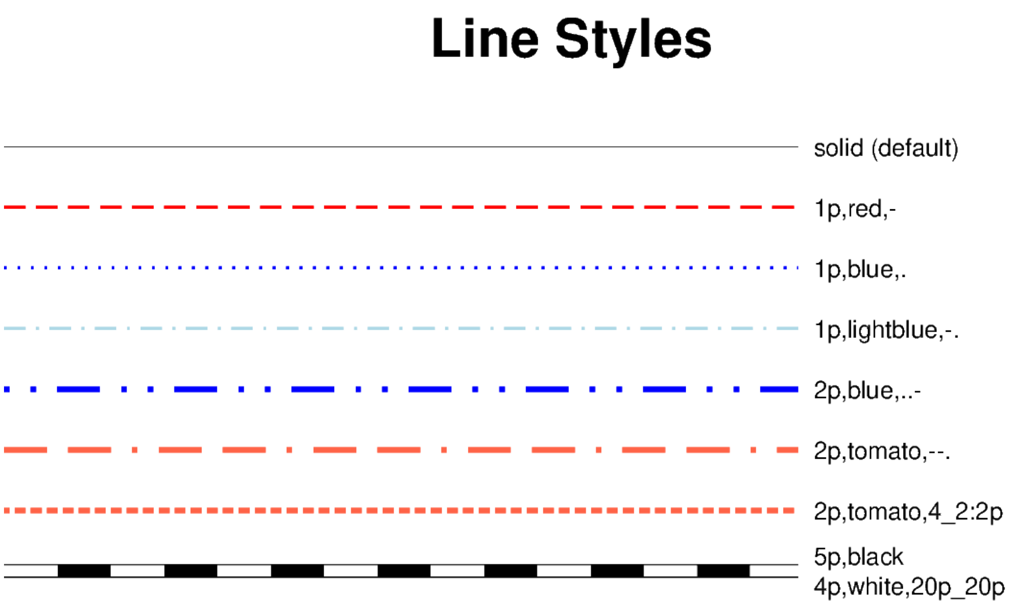
Total running time of the script: (0 minutes 7.829 seconds)

Line styles

The *pygmt.Figure.plot* method can plot lines in different styles. The default line style is a 0.25-point wide, black, solid line, and can be customized with the `pen` parameter.

A *pen* in GMT has three attributes: *width*, *color*, and *style*. The *style* attribute controls the appearance of the line. Giving "dotted" or "." yields a dotted line, whereas a dashed pen is requested with "dashed" or "-". Also combinations of dots and dashes, like ".-" for a dot-dashed line, are allowed.

For more advanced *pen* attributes, see the GMT cookbook https://docs.generic-mapping-tools.org/latest/cookbook/features.html#wpen-attrib.



Out:

```
<IPython.core.display.Image object>
```

```
import numpy as np
import pygmt

# Generate a two-point line for plotting
x = np.array([0, 7])
y = np.array([9, 9])

fig = pygmt.Figure()
fig.basemap(region=[0, 10, 0, 10], projection="X15c/8c", frame='+t"Line Styles"')

# Plot the line using the default line style
fig.plot(x=x, y=y)
fig.text(x=x[-1], y=y[-1], text="solid (default)", justify="ML", offset="0.2c/0c")

# Plot the line using different line styles
for linestyle in [
    "1p,red,-",  # dashed line
    "1p,blue,.",  # dotted line
    "1p,lightblue,-.",  # dash-dotted line
    "2p,blue,..-",  # dot-dot-dashed line
    "2p,tomato,--.",  # dash-dash-dotted line
    # A pattern of 4-point-long line segments and 2-point-long gaps between
    # segments, with pattern offset by 2 points from the origin
    "2p,tomato,4_2:2p",
]:
    y -= 1  # Move the current line down
```

(continues on next page)

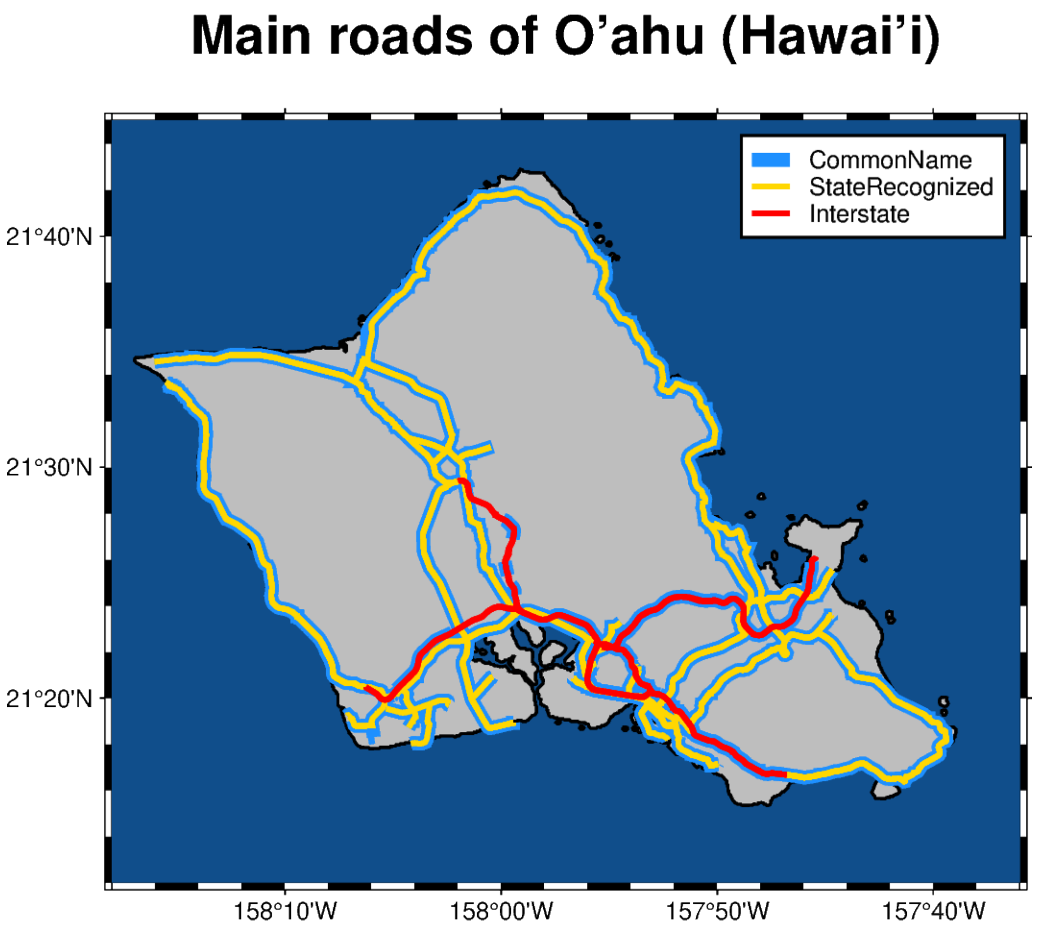
(continued from previous page)

```
    fig.plot(x=x, y=y, pen=linestyle)
    fig.text(x=x[-1], y=y[-1], text=linestyle, justify="ML", offset="0.2c/0c")

# Plot the line like a railway track (black/white).
# The trick here is plotting the same line twice but with different line styles
y -= 1  # move the current line down
fig.plot(x=x, y=y, pen="5p,black")
fig.plot(x=x, y=y, pen="4p,white,20p_20p")
fig.text(x=x[-1], y=y[-1], text="5p,black", justify="ML", offset="0.2c/0.2c")
fig.text(x=x[-1], y=y[-1], text="4p,white,20p_20p", justify="ML", offset="0.2c/-0.2c")

fig.show()
```

Total running time of the script: (0 minutes 6.247 seconds)

Roads

The *pygmt.Figure.plot* method allows us to plot geographical data such as lines which are stored in a geopandas.GeoDataFrame object. Use geopandas.read_file to load data from any supported OGR format such as a shapefile (.shp), GeoJSON (.geojson), geopackage (.gpkg), etc. Then, pass the geopandas.GeoDataFrame as an argument to the data parameter in *pygmt.Figure.plot*, and style the geometry using the pen parameter.



Out:

```
<IPython.core.display.Image object>
```

```
import geopandas as gpd
import pygmt

# Read shapefile data using geopandas
gdf = gpd.read_file(
    "http://www2.census.gov/geo/tiger/TIGER2015/PRISECROADS/tl_2015_15_prisecroads.zip"
)
# The dataset contains different road types listed in the RTTYP column,
# here we select the following ones to plot:
roads_common = gdf[gdf.RTTYP == "M"]  # Common name roads
roads_state = gdf[gdf.RTTYP == "S"]  # State recognized roads
roads_interstate = gdf[gdf.RTTYP == "I"]  # Interstate roads

fig = pygmt.Figure()

# Define target region around O'ahu (Hawai'i)
region = [-158.3, -157.6, 21.2, 21.75]  # xmin, xmax, ymin, ymax

title = r"Main roads of O\047ahu (Hawai\047i)"  # \047 is octal code for '
fig.basemap(region=region, projection="M12c", frame=["af", f'WSne+t"{title}"'])
fig.coast(land="gray", water="dodgerblue4", shorelines="1p,black")

# Plot the individual road types with different pen settings and assign labels
# which are displayed in the legend
fig.plot(data=roads_common, pen="5p,dodgerblue", label="CommonName")
fig.plot(data=roads_state, pen="2p,gold", label="StateRecognized")
fig.plot(data=roads_interstate, pen="2p,red", label="Interstate")

# Add legend
fig.legend()

fig.show()
```

Total running time of the script: (0 minutes 11.598 seconds)

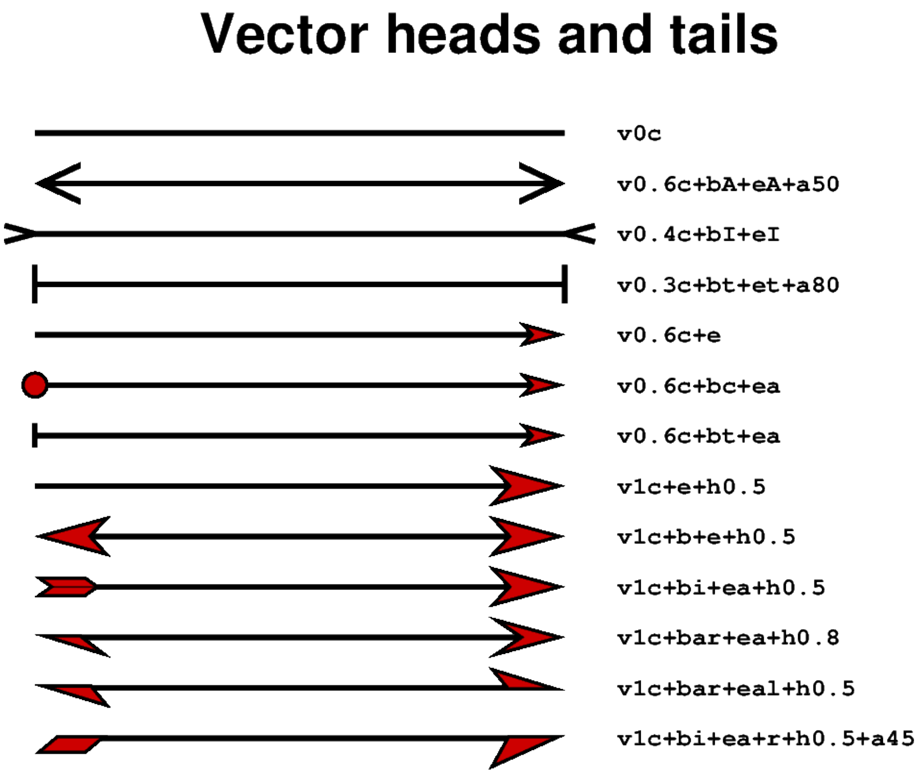
Vector heads and tails

Many modules in PyGMT allow plotting vectors with individual heads and tails. For this purpose, several modifiers may be appended to the corresponding vector-producing parameters for specifying the placement of vector heads and tails, their shapes, and the justification of the vector.

To place a vector head at the beginning of the vector path simply append **+b** to the vector-producing option (use **+e** to place one at the end). Optionally, append **t** for a terminal line, **c** for a circle, **a** for arrow (default), **i** for tail, **A** for plain open arrow, and **I** for plain open tail. Further append **l** or **r** (e.g. `+bar`) to only draw the left or right half-sides of the selected head/tail (default is both sides) or use **+l** or **+r** to apply simultaneously to both sides. In this context left and right refer to the side of the vector line when viewed from the beginning point to the end point of a line segment. The angle of the vector head apex can be set using **+a***angle* (default is 30). The shape of the vector head can be adjusted using **+h***shape* (e.g. +h0.5).

For further modifiers see the *Vector Attributes* subsection of the corresponding module.

In the following we use the `pygmt.Figure.plot` method to plot vectors with individual heads and tails. We must specify the modifiers (together with the vector type, here `v`, see also *Vector types documentation*) by passing the corresponding shortcuts to the `style` parameter.



Out:

```
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()
fig.basemap(
    region=[0, 10, 0, 15], projection="X15c/10c", frame='+t"Vector heads and tails"'
)

x = 1
y = 14
angle = 0  # in degrees, measured counter-clockwise from horizontal
length = 7

for vecstyle in [
```

(continues on next page)

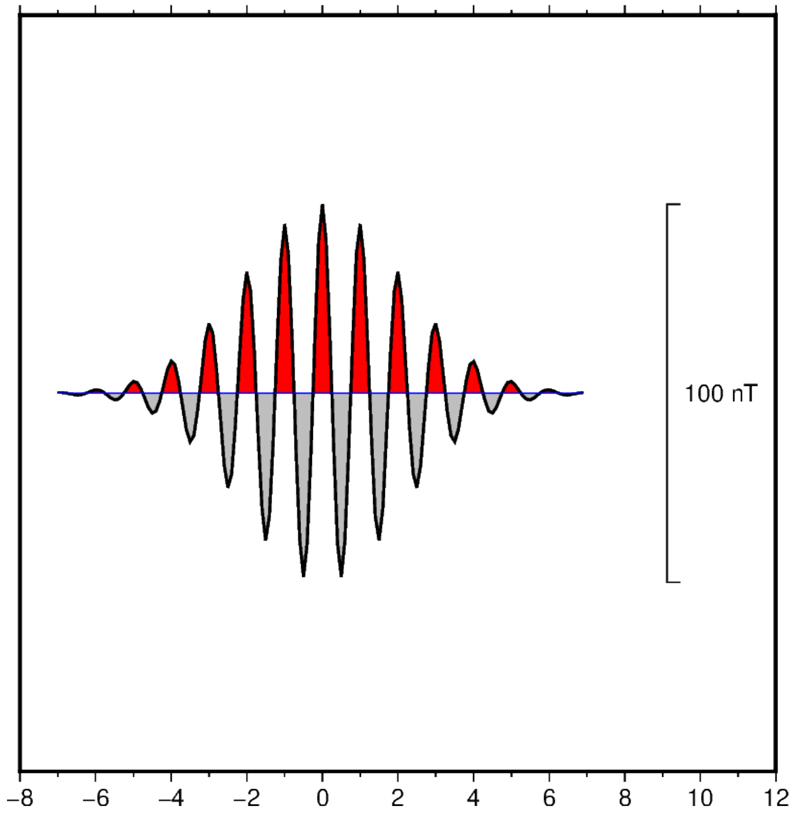
(continued from previous page)

```
    # vector without head and tail (line)
    "v0c",
    # plain open arrow at beginning and end, angle of the vector head apex is
    # set to 50
    "v0.6c+bA+eA+a50",
    # plain open tail at beginning and end
    "v0.4c+bI+eI",
    # terminal line at beginning and end, angle of vector head apex is set
    # to 80
    "v0.3c+bt+et+a80",
    # arrow head at end
    "v0.6c+e",
    # circle at beginning and arrow head at end
    "v0.6c+bc+ea",
    # terminal line at beginning and arrow head at end
    "v0.6c+bt+ea",
    # arrow head at end, shape of vector head is set to 0.5
    "v1c+e+h0.5",
    # modified arrow heads at beginning and end
    "v1c+b+e+h0.5",
    # tail at beginning and arrow with modified vector head at end
    "v1c+bi+ea+h0.5",
    # half-sided arrow head (right side) at beginning and arrow at the end
    "v1c+bar+ea+h0.8",
    # half-sided arrow heads at beginning (right side) and end (left side)
    "v1c+bar+eal+h0.5",
    # half-sided tail at beginning and arrow at end (right side for both)
    "v1c+bi+ea+r+h0.5+a45",
]:
    fig.plot(
        x=x, y=y, style=vecstyle, direction=([angle], [length]), pen="2p", color="red3"
    )
    fig.text(
        x=6, y=y, text=vecstyle, font="Courier-Bold", justify="ML", offset="0.2c/0c"
    )
    y -= 1  # move the next vector down

fig.show()
```

Total running time of the script: (0 minutes 8.503 seconds)

Wiggle along tracks

The *pygmt.Figure.wiggle* method can plot z = f(x,y) anomalies along tracks. `x`, `y`, `z` can be specified as 1d arrays or within a specified file. The `scale` parameter can be used to set the scale of the anomaly in data/distance units. The positive and/or negative areas can be filled with color by setting the `color` parameter.



Out:

```
<IPython.core.display.Image object>
```

```
import numpy as np
import pygmt

# Create (x, y, z) triplets
x = np.arange(-7, 7, 0.1)
y = np.zeros(x.size)
z = 50 * np.exp(-((x / 3) ** 2)) * np.cos(2 * np.pi * x)

fig = pygmt.Figure()
fig.basemap(region=[-8, 12, -1, 1], projection="X10c", frame=["Snlr", "xa2f1"])
fig.wiggle(
    x=x,
    y=y,
```

(continues on next page)

(continued from previous page)

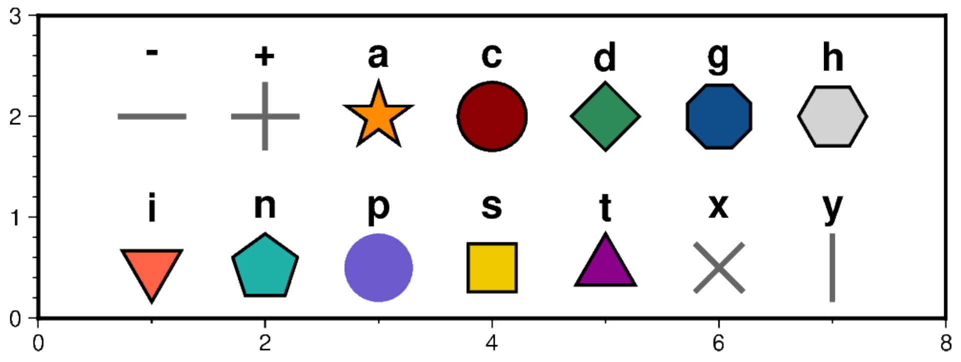
```
    z=z,
    # Set anomaly scale to "20c"
    scale="20c",
    # Fill positive and negative areas red and gray, respectively
    color=["red+p", "gray+n"],
    # Set the outline width to "1.0p"
    pen="1.0p",
    # Draw a blue track with a width of 0.5 points
    track="0.5p,blue",
    # Plot a vertical scale bar at the right middle. The bar length is 100 in
    # data (z) units. Set the z unit label to "nT".
    position="jRM+w100+lnT",
)
fig.show()
```

Total running time of the script: (0 minutes 2.019 seconds)

9.4.3 Symbols and markers

Basic geometric symbols

The *pygmt.Figure.plot* method can plot individual geometric symbols by passing the corresponding shortcuts to the `style` parameter. The 14 basic geometric symbols are shown underneath their corresponding shortcut codes. Four symbols (**-**, **+**, **x** and **y**) are line-symbols only for which we can adjust the linewidth via the `pen` parameter. The point symbol (**p**) only takes a color fill which we can define via the `color` parameter. For the remaining symbols we may define a linewidth as well as a color fill.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()
fig.basemap(region=[0, 8, 0, 3], projection="X12c/4c", frame=True)
```

(continues on next page)

(continued from previous page)

```
# define fontstlye for annotations
font = "15p,Helvetica-Bold"

# upper row
y = 2

# use a dash in x direction (-) with a size of 0.9 cm,
# linewidth is set to 2p and the linecolor to "gray40"
fig.plot(x=1, y=y, style="-0.9c", pen="2p,gray40")
fig.text(x=1, y=y + 0.6, text="-", font=font)

# use a plus (+) with a size of 0.9 cm,
# linewidth is set to 2p and the linecolor to "gray40"
fig.plot(x=2, y=y, style="+0.9c", pen="2p,gray40")
fig.text(x=2, y=y + 0.6, text="+", font=font)

# use a star (a) with a size of 0.9 cm,
# linewidth is set to 1p, the linecolor to "black" (default) and the
# color fill to "darkorange"
fig.plot(x=3, y=y, style="a0.9c", pen="1p,black", color="darkorange")
fig.text(x=3, y=y + 0.6, text="a", font=font)

# use a circle (c) with a size of 0.9 cm,
# linewidth is set to 1p, the linecolor to "black" and the
# color fill to "darkred"
fig.plot(x=4, y=y, style="c0.9c", pen="1p,black", color="darkred")
fig.text(x=4, y=y + 0.6, text="c", font=font)

# use a diamond (d) with a size of 0.9 cm,
# linewidth is set to 1p, the linecolor to "black" and the
# color fill to "seagreen"
fig.plot(x=5, y=y, style="d0.9c", pen="1p,black", color="seagreen")
fig.text(x=5, y=y + 0.6, text="d", font=font)

# use a octagon (g) with a size of 0.9 cm,
# linewidth is set to 1p, the linecolor to "black" and the
# color fill to "dodgerblue4"
fig.plot(x=6, y=y, style="g0.9c", pen="1p,black", color="dodgerblue4")
fig.text(x=6, y=y + 0.6, text="g", font=font)

# use a hexagon (h) with a size of 0.9 cm,
# linewidth is set to 1p, the linecolor to "black" and the
# color fill to "lightgray"
fig.plot(x=7, y=y, style="h0.9c", pen="1p,black", color="lightgray")
fig.text(x=7, y=y + 0.6, text="h", font=font)

# lower row
y = 0.5

# use an inverted triangle (i) with a size of 0.9 cm,
# linewidth is set to 1p, the linecolor to "black" and the
```

(continues on next page)

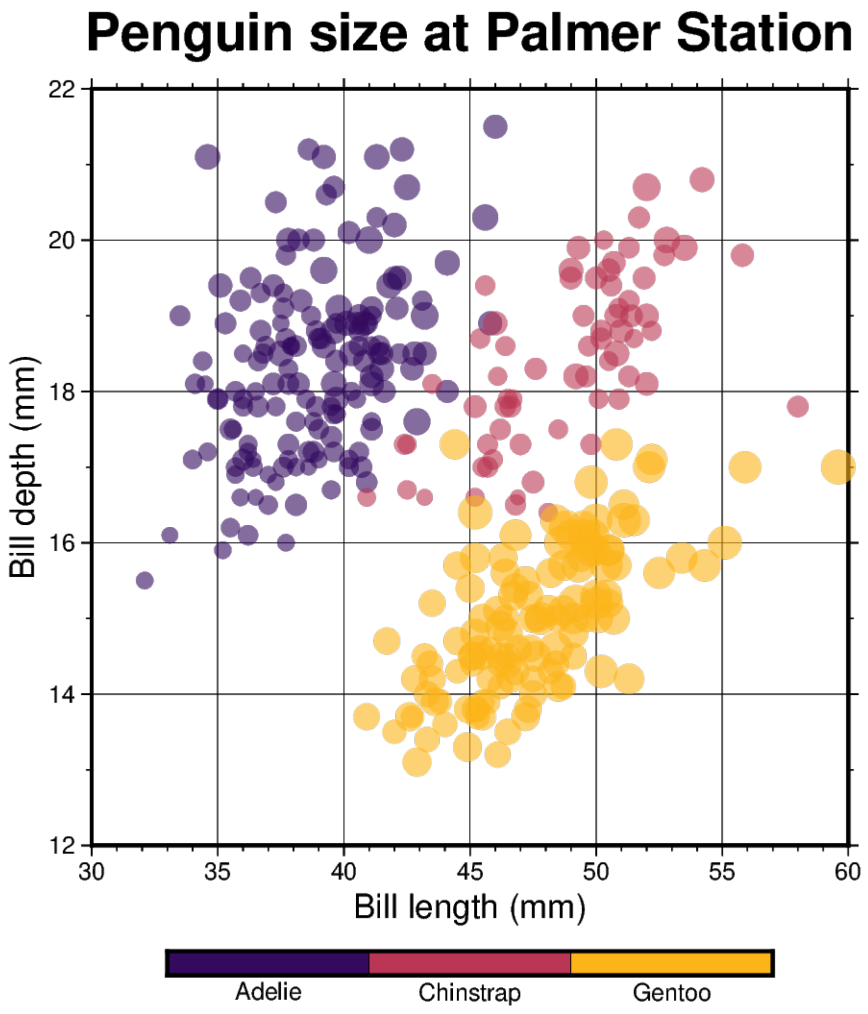
(continued from previous page)

```python
# color fill to "tomato"
fig.plot(x=1, y=y, style="i0.9c", pen="1p,black", color="tomato")
fig.text(x=1, y=y + 0.6, text="i", font=font)

# use pentagon (n) with a size of 0.9 cm,
# linewidth is set to 1p, the linecolor to "black" and the
# color fill to "lightseagreen"
fig.plot(x=2, y=y, style="n0.9c", pen="1p,black", color="lightseagreen")
fig.text(x=2, y=y + 0.6, text="n", font=font)

# use a point (p) with a size of 0.9 cm,
# color fill is set to "lightseagreen"
fig.plot(x=3, y=y, style="p0.9c", color="slateblue")
fig.text(x=3, y=y + 0.6, text="p", font=font)

# use square (s) with a size of 0.9 cm,
# linewidth is set to 1p, the linecolor to "black" and the
# color fill to "gold2"
fig.plot(x=4, y=y, style="s0.9c", pen="1p,black", color="gold2")
fig.text(x=4, y=y + 0.6, text="s", font=font)

# use triangle (t) with a size of 0.9 cm,
# linewidth is set to 1p, the linecolor to "black" and the
# color fill to "magenta4"
fig.plot(x=5, y=y, style="t0.9c", pen="1p,black", color="magenta4")
fig.text(x=5, y=y + 0.6, text="t", font=font)

# use cross (x) with a size of 0.9 cm,
# linewidth is set to 2p and the linecolor to "gray40"
fig.plot(x=6, y=y, style="x0.9c", pen="2p,gray40")
fig.text(x=6, y=y + 0.6, text="x", font=font)

# use a dash in y direction (y) with a size of 0.9 cm,
# linewidth is set to 2p and the linecolor to "gray40"
fig.plot(x=7, y=y, style="y0.9c", pen="2p,gray40")
fig.text(x=7, y=y + 0.6, text="y", font=font)

fig.show()
```

Total running time of the script: (0 minutes 8.792 seconds)

Color points by categories

The *pygmt.Figure.plot* method can be used to plot symbols which are color-coded by categories. In the example below, we show how the Palmer Penguins dataset can be visualized. Here, we can pass the individual categories included in the "species" column directly to the `color` parameter via `color=df.species.cat.codes.astype(int)`. Additionally, we have to set `cmap=True`. A desired colormap can be selected via the *pygmt.makecpt* method.



Out:

```
<IPython.core.display.Image object>
```

```python
import pandas as pd
import pygmt

# Load sample penguins data and convert 'species' column to categorical dtype
df = pd.read_csv("https://github.com/mwaskom/seaborn-data/raw/master/penguins.csv")
df.species = df.species.astype(dtype="category")
```

(continues on next page)

(continued from previous page)

```
# Use pygmt.info to get region bounds (xmin, xmax, ymin, ymax)
# The below example will return a numpy array like [30.0, 60.0, 12.0, 22.0]
region = pygmt.info(
    data=df[["bill_length_mm", "bill_depth_mm"]],  # x and y columns
    per_column=True,  # report the min/max values per column as a numpy array
    # round the min/max values of the first two columns to the nearest multiple
    # of 3 and 2, respectively
    spacing=(3, 2),
)

# Make a 2D categorical scatter plot, coloring each of the 3 species
# differently
fig = pygmt.Figure()

# Generate a basemap of 10 cm x 10 cm size
fig.basemap(
    region=region,
    projection="X10c/10c",
    frame=[
        'xafg+l"Bill length (mm)"',
        'yafg+l"Bill depth (mm)"',
        'WSen+t"Penguin size at Palmer Station"',
    ],
)

# Define a colormap to be used for three categories, define the range of the
# new discrete CPT using series=(lowest_value, highest_value, interval),
# use color_model="+cAdelie,Chinstrap,Gentoo" to write the discrete color
# palette "inferno" in categorical format and add the species names as
# annotations for the colorbar
pygmt.makecpt(cmap="inferno", series=(0, 2, 1), color_model="+cAdelie,Chinstrap,Gentoo")

fig.plot(
    # Use bill length and bill depth as x and y data input, respectively
    x=df.bill_length_mm,
    y=df.bill_depth_mm,
    # Vary each symbol size according to another feature (body mass,
    # scaled by 7.5*10e-5)
    size=df.body_mass_g * 7.5e-5,
    # Points colored by categorical number code
    color=df.species.cat.codes.astype(int),
    # Use colormap created by makecpt
    cmap=True,
    # Do not clip symbols that fall close to the map bounds
    no_clip=True,
    # Use circles as symbols with size in centimeter units
    style="cc",
    # Set transparency level for all symbols to deal with overplotting
    transparency=40,
)
```

(continues on next page)

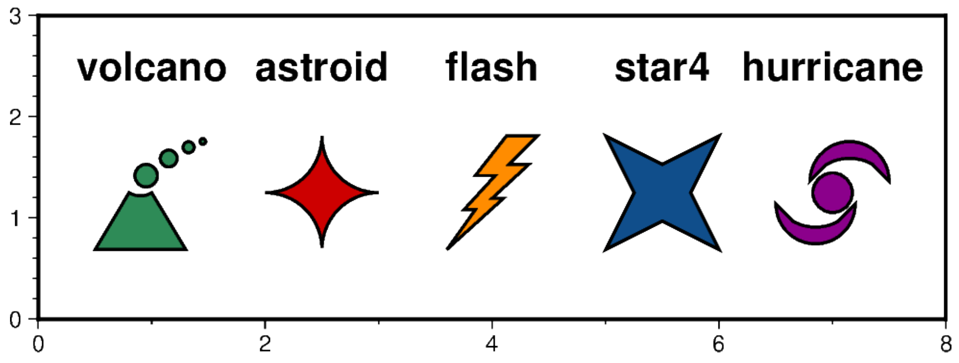
(continued from previous page)

```
# Add colorbar legend
fig.colorbar()

fig.show()
```

Total running time of the script: (0 minutes 4.186 seconds)

Custom symbols

The *pygmt.Figure.plot* method can plot individual custom symbols by passing the corresponding symbol name together with the **k** shortcut to the `style` parameter. In total 41 custom symbols are already included of which the following plot shows five exemplary ones. The symbols are shown underneath their corresponding names. For the remaining symbols see the GMT cookbook https://docs.generic-mapping-tools.org/latest/cookbook/custom-symbols.html.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()
fig.basemap(region=[0, 8, 0, 3], projection="X12c/4c", frame=True)

# define pen and fontstlye for annotations
pen = "1p,black"
font = "15p,Helvetica-Bold"

# use the volcano symbol with a size of 1.5c,
# fill color is set to "seagreen"
fig.plot(x=1, y=1.25, style="kvolcano/1.5c", pen=pen, color="seagreen")
fig.text(x=1, y=2.5, text="volcano", font=font)

# use the astroid symbol with a size of 1.5c,
# fill color is set to "red3"
fig.plot(x=2.5, y=1.25, style="kastroid/1.5c", pen=pen, color="red3")
```

(continues on next page)

(continued from previous page)

```
fig.text(x=2.5, y=2.5, text="astroid", font=font)

# use the flash symbol with a size of 1.5c,
# fill color is set to "darkorange"
fig.plot(x=4, y=1.25, style="kflash/1.5c", pen=pen, color="darkorange")
fig.text(x=4, y=2.5, text="flash", font=font)

# use the star4 symbol with a size of 1.5c,
# fill color is set to "dodgerblue4"
fig.plot(x=5.5, y=1.25, style="kstar4/1.5c", pen=pen, color="dodgerblue4")
fig.text(x=5.5, y=2.5, text="star4", font=font)

# use the hurricane symbol with a size of 1.5c,
# fill color is set to "magenta4"
fig.plot(x=7, y=1.25, style="khurricane/1.5c", pen=pen, color="magenta4")
fig.text(x=7, y=2.5, text="hurricane", font=font)

fig.show()
```

Total running time of the script: (0 minutes 4.232 seconds)

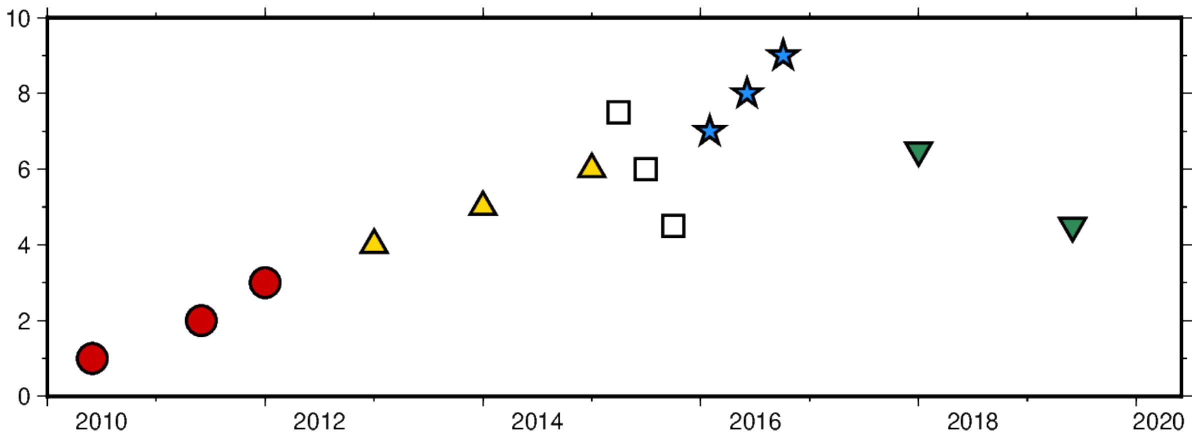
Datetime inputs

Datetime inputs of the following types are supported in PyGMT:

- `numpy.datetime64`
- `pandas.DatetimeIndex`
- `xarray.DataArray`: datetimes included in a *xarray.DataArray*
- raw datetime strings in ISO 8601 format (e.g. `"YYYY-MM-DD"`, `"YYYY-MM-DDTHH"`, and `"YYYY-MM-DDTHH:MM:SS"`)
- Python built-in `datetime.datetime` and `datetime.date`

We can pass datetime inputs based on one of the types listed above directly to the `x` and `y` parameters of e.g. the *`pygmt.Figure.plot`* method.

The `region` parameter has to include the x and y axis limits in the form [*date_min*, *date_max*, *ymin*, *ymax*]. Here *date_min* and *date_max* can be directly defined as datetime input.



Out:

```
<IPython.core.display.Image object>
```

```python
import datetime

import numpy as np
import pandas as pd
import pygmt
import xarray as xr

fig = pygmt.Figure()

# create a basemap with limits of 2010-01-01 to 2020-06-01 on the x axis and
# 0 to 10 on the y axis
fig.basemap(
    projection="X15c/5c",
    region=[datetime.date(2010, 1, 1), datetime.date(2020, 6, 1), 0, 10],
    frame=["WSen", "af"],
)

# numpy.datetime64 types
x = np.array(["2010-06-01", "2011-06-01T12", "2012-01-01T12:34:56"], dtype="datetime64")
y = [1, 2, 3]
fig.plot(x=x, y=y, style="c0.4c", pen="1p", color="red3")

# pandas.DatetimeIndex
x = pd.date_range("2013", periods=3, freq="YS")
y = [4, 5, 6]
fig.plot(x=x, y=y, style="t0.4c", pen="1p", color="gold")

# xarray.DataArray
x = xr.DataArray(data=pd.date_range(start="2015-03", periods=3, freq="QS"))
y = [7.5, 6, 4.5]
fig.plot(x=x, y=y, style="s0.4c", pen="1p")

# raw datetime strings
x = ["2016-02-01", "2016-06-04T14", "2016-10-04T00:00:15"]
y = [7, 8, 9]
fig.plot(x=x, y=y, style="a0.4c", pen="1p", color="dodgerblue")

# the Python built-in datetime and date
x = [datetime.date(2018, 1, 1), datetime.datetime(2019, 6, 1, 20, 5, 45)]
y = [6.5, 4.5]
fig.plot(x=x, y=y, style="i0.4c", pen="1p", color="seagreen")

fig.show()
```

Total running time of the script: (0 minutes 2.977 seconds)

Multi-parameter symbols

The *pygmt.Figure.plot* method can plot individual multi-parameter symbols by passing the corresponding shortcuts (**e**, **j**, **r**, **R**, **w**) to the `style` parameter:

- **e**: ellipse
- **j**: rotated rectangle
- **r**: rectangle
- **R**: rounded rectangle
- **w**: pie wedge

```
import pygmt
```

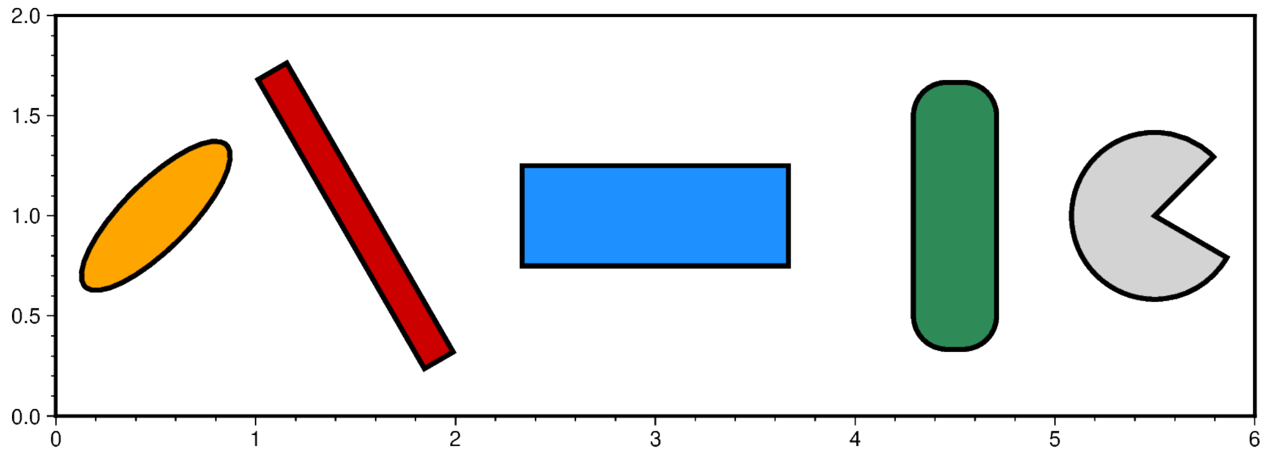
We can plot multi-parameter symbols using the same symbol style. We need to define locations (lon, lat) via the `x` and `y` parameters (scalar for a single symbol or 1d list for several ones) and two or three symbol parameters after those shortcuts via the `style` parameter.

The multi-parameter symbols in the `style` parameter are defined as:

- **e**: ellipse, `direction/major_axis/minor_axis`
- **j**: rotated rectangle, `direction/width/height`
- **r**: rectangle, `width/height`
- **R**: rounded rectangle, `width/height/radius`
- **w**: pie wedge, `radius/startdir/stopdir`, the last two arguments are directions given in degrees counter-clockwise from horizontal

Upper-case versions **E**, **J**, and **W** are similar to **e**, **j** and **w** but expect geographic azimuths and distances.

```
fig = pygmt.Figure()
fig.basemap(region=[0, 6, 0, 2], projection="x3c", frame=True)

# Ellipse
fig.plot(x=0.5, y=1, style="e45/3/1", color="orange", pen="2p,black")
# Rotated rectangle
fig.plot(x=1.5, y=1, style="j120/5/0.5", color="red3", pen="2p,black")
# Rectangle
fig.plot(x=3, y=1, style="r4/1.5", color="dodgerblue", pen="2p,black")
# Rounded rectangle
fig.plot(x=4.5, y=1, style="R1.25/4/0.5", color="seagreen", pen="2p,black")
# Pie wedge
fig.plot(x=5.5, y=1, style="w2.5/45/330", color="lightgray", pen="2p,black")

fig.show()
```



Out:

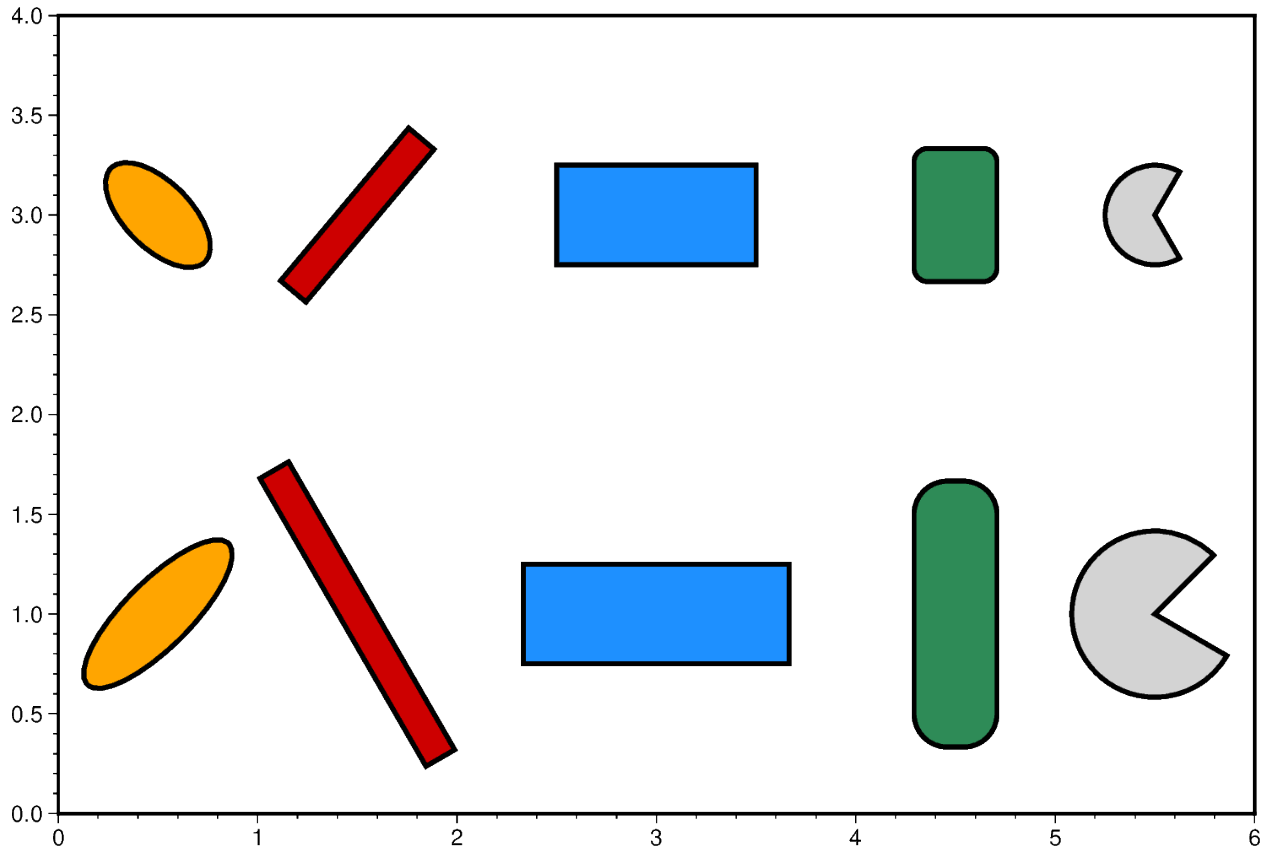
```
<IPython.core.display.Image object>
```

We can also plot symbols with varying parameters via defining those values in a 2d list or numpy array (`[[parameters]]` for a single symbol or `[[parameters_1],[parameters_2],[parameters_i]]` for several ones) or using an appropriately formatted input file and passing it to `data`.

The symbol parameters in the 2d list or numpy array are defined as:

- **e**: ellipse, `[[lon, lat, direction, major_axis, minor_axis]]`
- **j**: rotated rectangle, `[[lon, lat, direction, width, height]]`
- **r**: rectangle, `[[lon, lat, width, height]]`
- **R**: rounded rectangle, `[[lon, lat, width, height, radius]]`
- **w**: pie wedge, `[[lon, lat, radius, startdir, stopdir]]`, the last two arguments are directions given in degrees counter-clockwise from horizontal

```python
fig = pygmt.Figure()
fig.basemap(region=[0, 6, 0, 4], projection="x3c", frame=["xa1f0.2", "ya0.5f0.1"])

# Ellipse
data = [[0.5, 1, 45, 3, 1], [0.5, 3, 135, 2, 1]]
fig.plot(data=data, style="e", color="orange", pen="2p,black")
# Rotated rectangle
data = [[1.5, 1, 120, 5, 0.5], [1.5, 3, 50, 3, 0.5]]
fig.plot(data=data, style="j", color="red3", pen="2p,black")
# Rectangle
data = [[3, 1, 4, 1.5], [3, 3, 3, 1.5]]
fig.plot(data=data, style="r", color="dodgerblue", pen="2p,black")
# Rounded rectangle
data = [[4.5, 1, 1.25, 4, 0.5], [4.5, 3, 1.25, 2.0, 0.2]]
fig.plot(data=data, style="R", color="seagreen", pen="2p,black")
# Pie wedge
data = [[5.5, 1, 2.5, 45, 330], [5.5, 3, 1.5, 60, 300]]
fig.plot(data=data, style="w", color="lightgray", pen="2p,black")

fig.show()
```



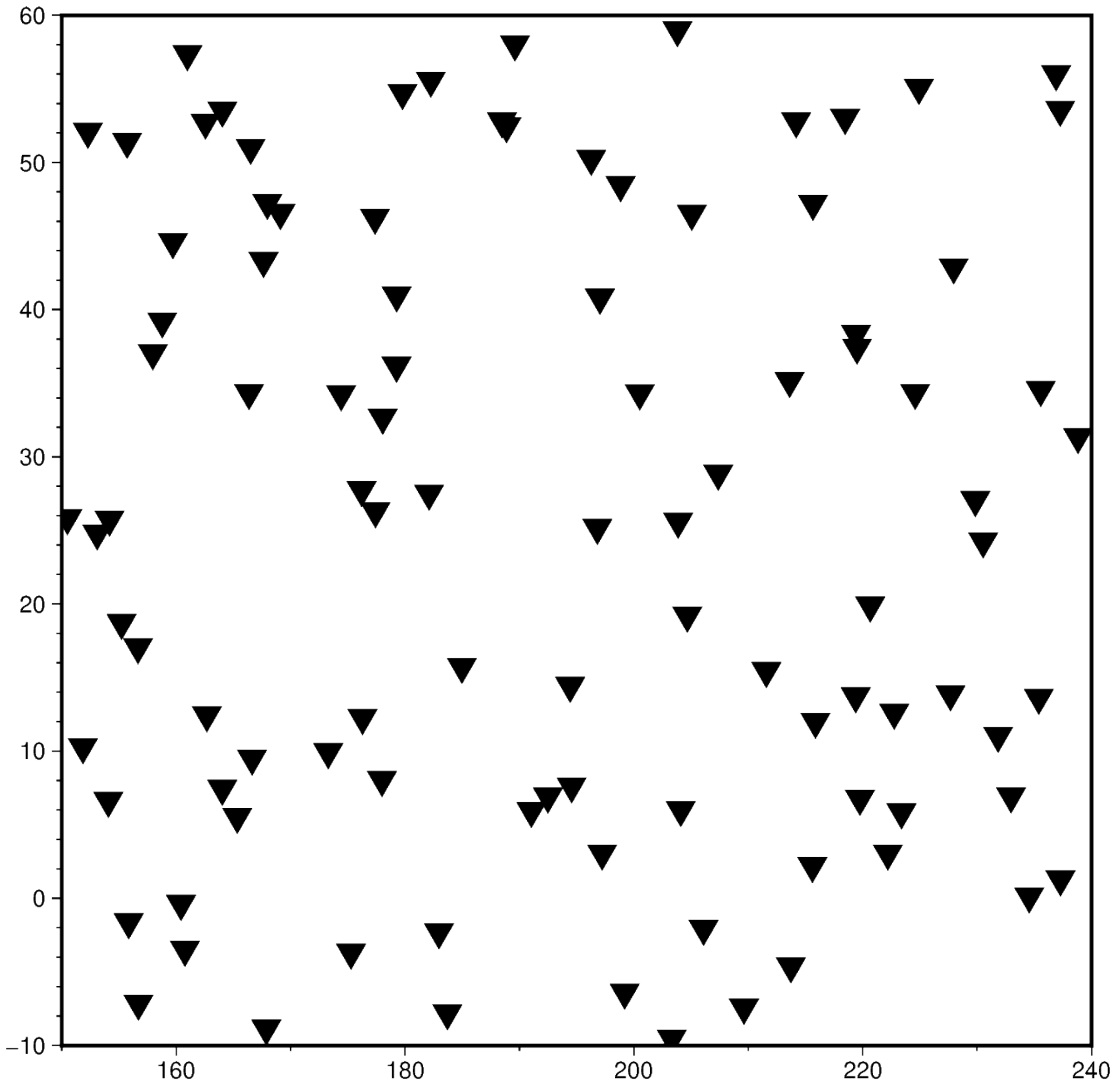
Out:

```
<IPython.core.display.Image object>
```

Total running time of the script: (0 minutes 6.515 seconds)

Points

The *pygmt.Figure.plot* method can plot points. The plot symbol and size is set with the `style` parameter.



Out:

```
<IPython.core.display.Image object>
```

```
import numpy as np
import pygmt

# Generate a random set of points to plot
np.random.seed(42)
region = [150, 240, -10, 60]
x = np.random.uniform(region[0], region[1], 100)
```

(continues on next page)

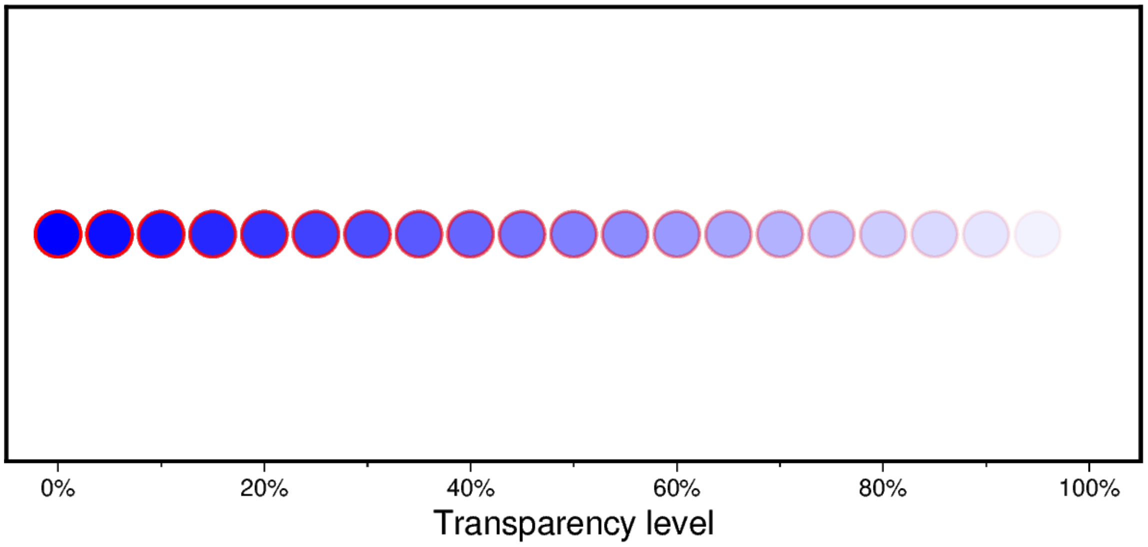
(continued from previous page)

```
y = np.random.uniform(region[2], region[3], 100)

fig = pygmt.Figure()
# Create a 15 cm x 15 cm basemap with a Cartesian projection (X) using the
# data region
fig.basemap(region=region, projection="X15c", frame=True)
# Plot using inverted triangles (i) of 0.5 cm size
fig.plot(x=x, y=y, style="i0.5c", color="black")
fig.show()
```

Total running time of the script: (0 minutes 2.247 seconds)

Points with varying transparency

Points can be plotted with different transparency levels by passing in an array argument to the `transparency` parameter of *pygmt.Figure.plot*.



Out:

```
<IPython.core.display.Image object>
```

```
import numpy as np
import pygmt

# prepare the input x and y data
x = np.arange(0, 105, 5)
y = np.ones(x.size)
# transparency level in percentage from 0 (i.e., opaque) to 100
transparency = x
```

(continues on next page)

(continued from previous page)

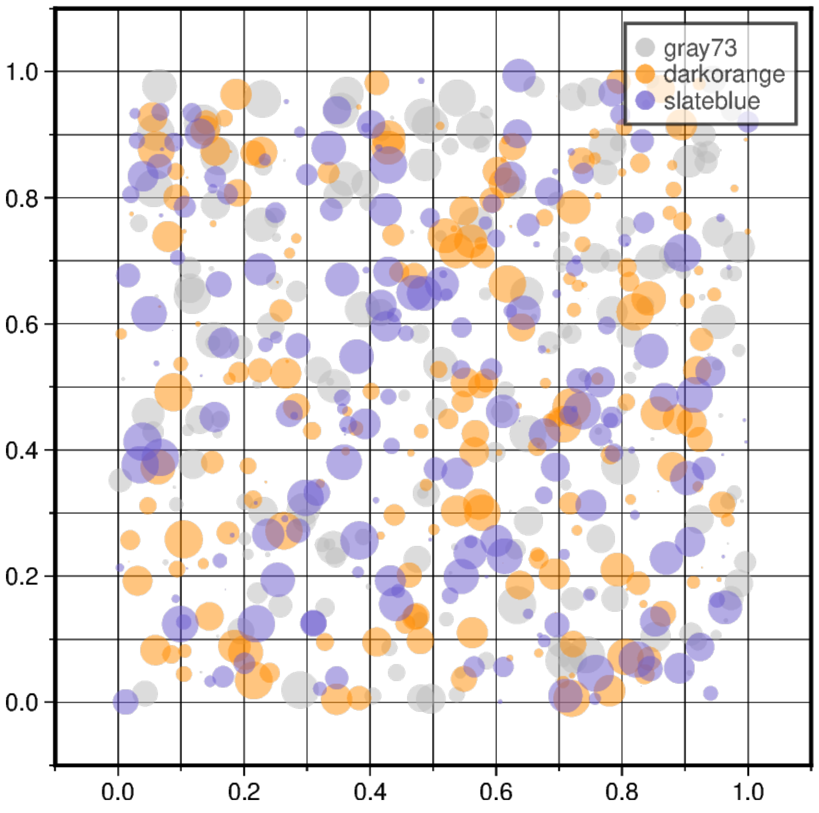
```
fig = pygmt.Figure()
fig.basemap(
    region=[-5, 105, 0, 2],
    frame=['xaf+l"Transparency level"+u%', "WSrt"],
    projection="X15c/6c",
)
fig.plot(x=x, y=y, style="c0.6c", color="blue", pen="1p,red", transparency=transparency)
fig.show()
```

Total running time of the script: (0 minutes 2.328 seconds)

Scatter plots with a legend

To create a scatter plot with a legend one may use a loop and create one scatter plot per item to appear in the legend and set the label accordingly.

Modified from the matplotlib example: https://matplotlib.org/gallery/lines_bars_and_markers/scatter_with_legend.html



Out:

```
<IPython.core.display.Image object>
```

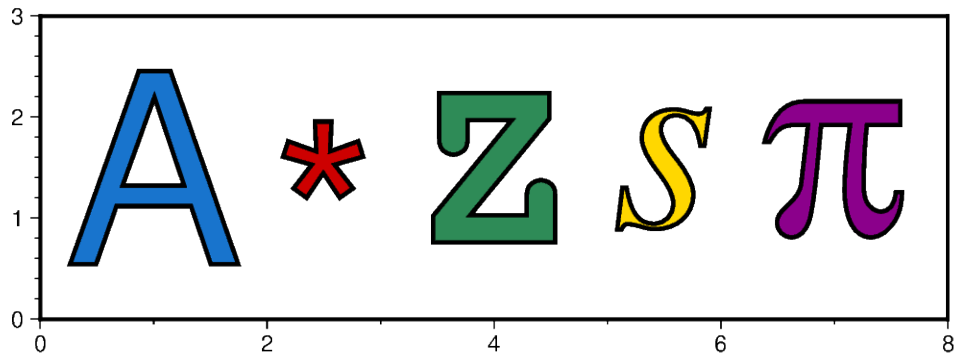
```
import numpy as np
import pygmt

np.random.seed(19680801)
n = 200  # number of random data points

fig = pygmt.Figure()
fig.basemap(
    region=[-0.1, 1.1, -0.1, 1.1],
    projection="X10c/10c",
    frame=["xa0.2fg", "ya0.2fg", "WSrt"],
)
for color in ["gray73", "darkorange", "slateblue"]:
    x, y = np.random.rand(2, n)  # random X and Y data in [0,1]
    size = np.random.rand(n) * 0.5  # random size [0,0.5], in cm
    # plot data points as circles (style="c"), with different sizes
    fig.plot(
        x=x,
        y=y,
        style="c",
        size=size,
        color=color,
        # Set the legend label,
        # and set the symbol size to be 0.25 cm (+S0.25c) in legend
        label=f"{color}+S0.25c",
        transparency=50,  # set transparency level for all symbols
    )

fig.legend(transparency=30)  # set transparency level for legends
fig.show()
```

Total running time of the script: (0 minutes 3.554 seconds)

Text symbols

The *pygmt.Figure.plot* method allows to plot text symbols. Text is normally placed with the *pygmt.Figure.text* method but there are times we wish to treat a character or even a string as a plottable symbol. A text symbol can be drawn by passing **l***size***+t***string* to the `style` parameter where *size* defines the size of the text symbol (note: the size is only approximate; no individual scaling is done for different characters) and *string* can be a letter or a text string (less than 256 characters). Optionally, you can append **+f***font* to select a particular font [Default is FONT_ANNOT_PRIMARY] as well as **+j***justify* to change the justification [Default is CM]. Outline and fill color of the text symbols can be customized via the `pen` and `color` parameters, respectively.

For all supported octal codes and fonts see the GMT cookbook https://docs.generic-mapping-tools.org/latest/cookbook/octal-codes.html and https://docs.generic-mapping-tools.org/latest/cookbook/postscript-fonts.html.



Out:

```
<IPython.core.display.Image object>
```

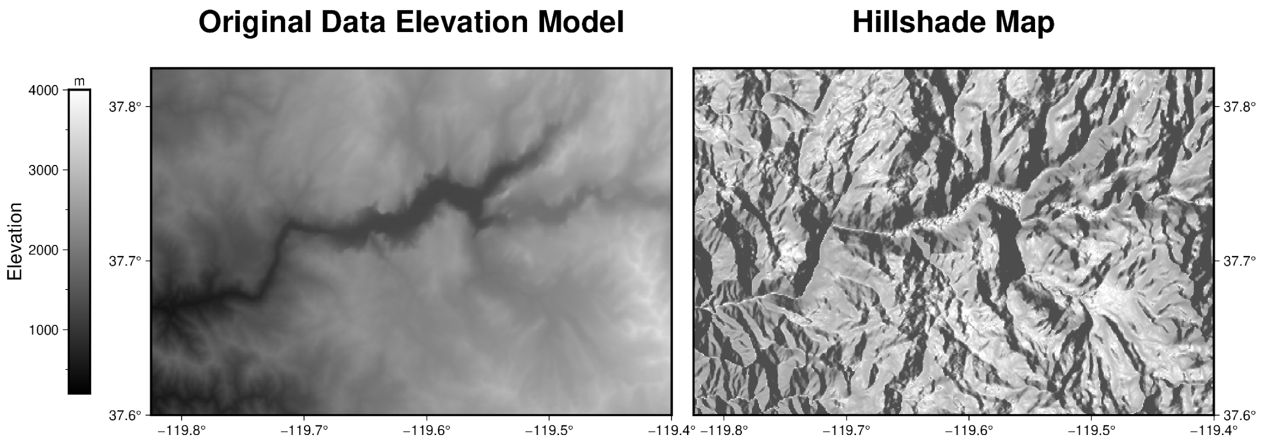
```python
import pygmt

fig = pygmt.Figure()

fig.basemap(region=[0, 8, 0, 3], projection="X12c/4c", frame=True)

pen = "1.5p"
# plot an uppercase "A" of size 3.5c, color fill is set to "dodgerblue3"
fig.plot(x=1, y=1.5, style="l3.5c+tA", color="dodgerblue3", pen=pen)
# plot an "asterisk" of size 3.5c, color fill is set to "red3"
fig.plot(x=2.5, y=1, style="l3.5c+t*", color="red3", pen=pen)
# plot an uppercase "Z" of size 3.5c and use the "Courier-Bold" font,
# color fill is set to "seagreen"
fig.plot(x=4, y=1.5, style="l3.5c+tZ+fCourier-Bold", color="seagreen", pen=pen)
# plot a lowercase "s" of size 3.5c and use the "Times-Italic" font,
# color fill is set to "gold"
fig.plot(x=5.5, y=1.5, style="l3.5c+ts+fTimes-Italic", color="gold", pen=pen)
# plot the pi symbol (\160 is octal code for pi) of size 3.5c, for this use
# the "Symbol" font, color fill is set to "magenta4"
fig.plot(x=7, y=1.5, style="l3.5c+t\160+fSymbol", color="magenta4", pen=pen)

fig.show()
```

Total running time of the script: (0 minutes 2.977 seconds)

9.4.4 Images, contours, and fields

Calculating grid gradient and radiance

The *pygmt.grdgradient* method calculates the gradient of a grid file. In the example shown below we will see how to calculate a hillshade map based on a Data Elevation Model (DEM). As input *pygmt.grdgradient* gets a `xarray.DataArray` object or a path string to a grid file, calculates the respective gradient and returns it as an `xarray.DataArray` object. We will use the `radiance` parameter in order to set the illumination source direction and altitude.



Out:

```
grdblend [NOTICE]: Remote data courtesy of GMT data server oceania [http://oceania.
↪generic-mapping-tools.org]
grdblend [NOTICE]: Earth Relief at 15x15 arc seconds provided by SRTM15+V2.1 [Tozer et␣
↪al., 2019].
grdblend [NOTICE]:   -> Download 10x10 degree grid tile (earth_relief_15s_p): N30W120
<IPython.core.display.Image object>
```

```python
import pygmt

# Define region of interest around Yosemite valley
region = [-119.825, -119.4, 37.6, 37.825]

# Load sample grid (3 arc second global relief) in target area
grid = pygmt.datasets.load_earth_relief(resolution="03s", region=region)

# calculate the reflection of a light source projecting from west to east
# (azimuth of 270 degrees) and at a latitude of 30 degrees from the horizon
dgrid = pygmt.grdgradient(grid=grid, radiance=[270, 30])

fig = pygmt.Figure()
# define figure configuration
pygmt.config(FORMAT_GEO_MAP="ddd.x", MAP_FRAME_TYPE="plain")

# --------------- plotting the original Data Elevation Model -----------
```

(continues on next page)

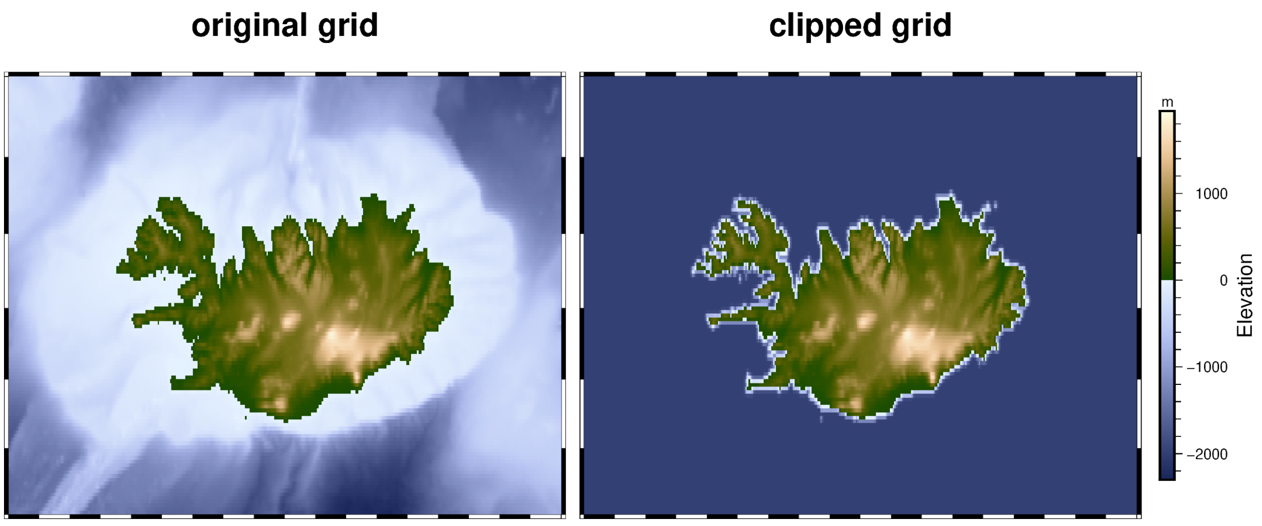
(continued from previous page)

```
pygmt.makecpt(cmap="gray", series=[200, 4000, 10])
fig.grdimage(
    grid=grid,
    projection="M12c",
    frame=['WSrt+t"Original Data Elevation Model"', "xa0.1", "ya0.1"],
    cmap=True,
)

fig.colorbar(position="JML+o1.4c/0c+w7c/0.5c", frame=["xa1000f500+lElevation", "y+lm"])

# --------------- plotting the hillshade map -----------

# Shift plot origin of the second map by 12.5 cm in x direction
fig.shift_origin(xshift="12.5c")

pygmt.makecpt(cmap="gray", series=[-1.5, 0.3, 0.01])
fig.grdimage(
    grid=dgrid,
    projection="M12c",
    frame=['lSEt+t"Hillshade Map"', "xa0.1", "ya0.1"],
    cmap=True,
)

fig.show()
```

Total running time of the script: (0 minutes 21.889 seconds)

Clipping grid values

The *pygmt.grdclip* method allows to clip defined ranges of grid values. In the example shown below we set all elevation values (grid points) smaller than 0 m (in general the bathymetric part of the grid) to a common value of -2000 m via the `below` parameter.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()

# Define region of interest around Iceland
region = [-28, -10, 62, 68]

# Load sample grid (3 arc minute global relief) in target area
grid = pygmt.datasets.load_earth_relief(resolution="03m", region=region)

# Plot original grid
fig.basemap(region=region, projection="M12c", frame=["f", '+t"original grid"'])
fig.grdimage(grid=grid, cmap="oleron")

# Shift plot origin of the second map by "width of the first map + 0.5 cm"
# in x direction
fig.shift_origin(xshift="w+0.5c")

# Set all grid points < 0 m to a value of -2000 m.
grid = pygmt.grdclip(grid, below=[0, -2000])

# Plot clipped grid
fig.basemap(region=region, projection="M12c", frame=["f", '+t"clipped grid"'])
fig.grdimage(grid=grid)
fig.colorbar(frame=["x+lElevation", "y+lm"], position="JMR+o0.5c/0c+w8c")

fig.show()
```

Total running time of the script: (0 minutes 4.435 seconds)

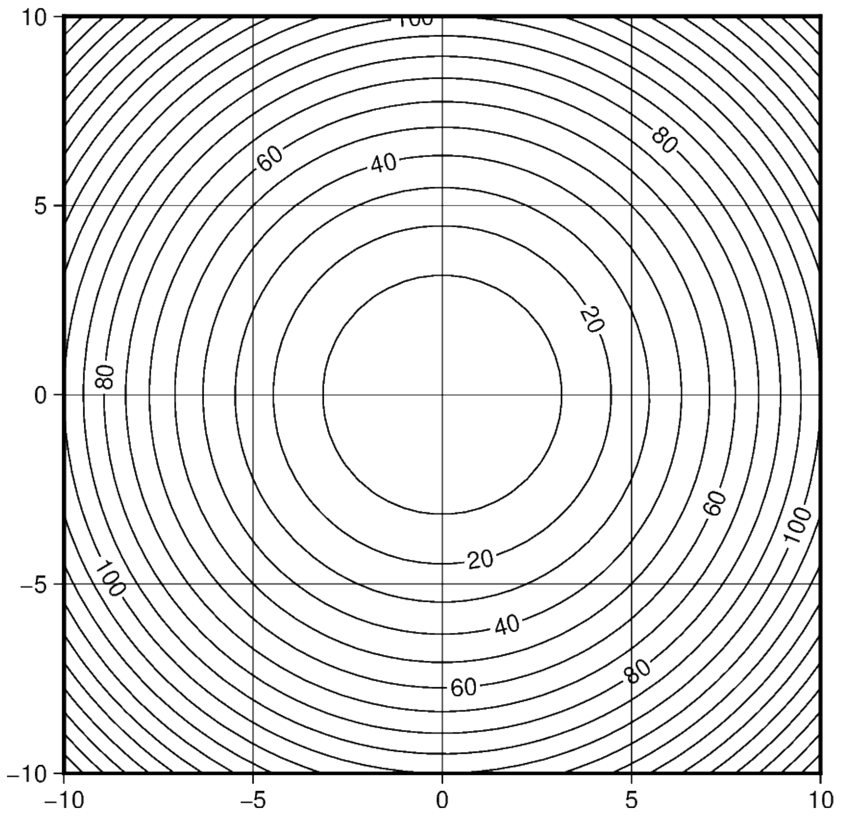
Contours

The *pygmt.Figure.contour* method can plot contour lines from a table of points by direct triangulation. The data for the triangulation can be provided using one of three methods:

1. x, y, z 1d numpy.ndarray data columns.
2. data 2d numpy.ndarray data matrix with 3 columns corresponding to x, y, z.
3. data path string to a file containing the x, y, z in a tabular format.

The parameters levels and annotation set the intervals of the contours and the annotation on the contours respectively.

In this example we supply the data as 1d numpy.ndarray with the x, y, and z parameters and draw the contours using a 0.5p pen with contours every 10 z values and annotations every 20 z values.



Out:

```
<IPython.core.display.Image object>
```

```
import numpy as np
import pygmt

# build the contours underlying data with the function z = x^2 + y^2
X, Y = np.meshgrid(np.linspace(-10, 10, 50), np.linspace(-10, 10, 50))
Z = X ** 2 + Y ** 2
x, y, z = X.flatten(), Y.flatten(), Z.flatten()


fig = pygmt.Figure()
fig.contour(
    region=[-10, 10, -10, 10],
    projection="X10c/10c",
    frame="ag",
    pen="0.5p",
    # pass the data as 3 1d data columns
    x=x,
    y=y,
    z=z,
```

(continues on next page)

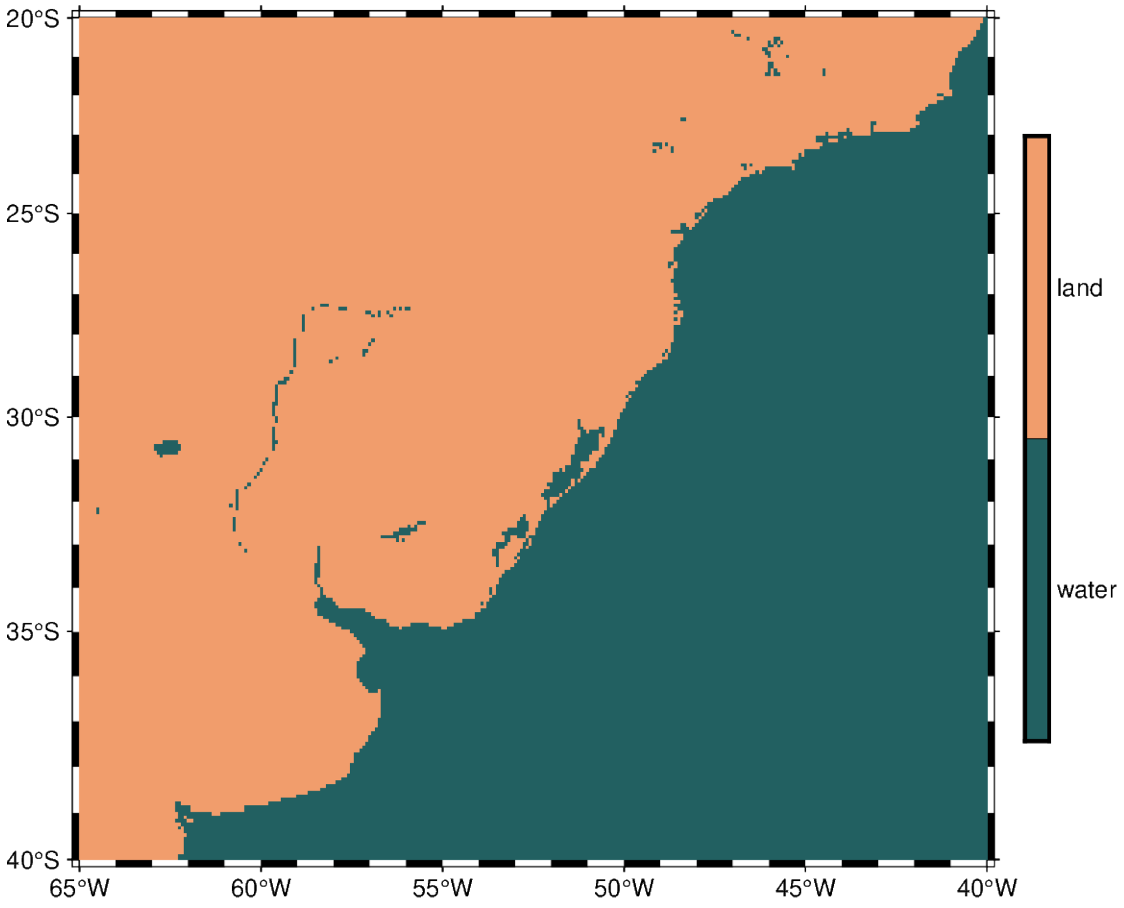
(continued from previous page)

```
    # set the contours z values intervals to 10
    levels=10,
    # set the contours annotation intervals to 20
    annotation=20,
)
fig.show()
```

Total running time of the script: (0 minutes 2.215 seconds)

Create 'wet-dry' mask grid

The *pygmt.grdlandmask* method allows setting all nodes on land or water to a specified value using the `maskvalues` parameter.



Out:

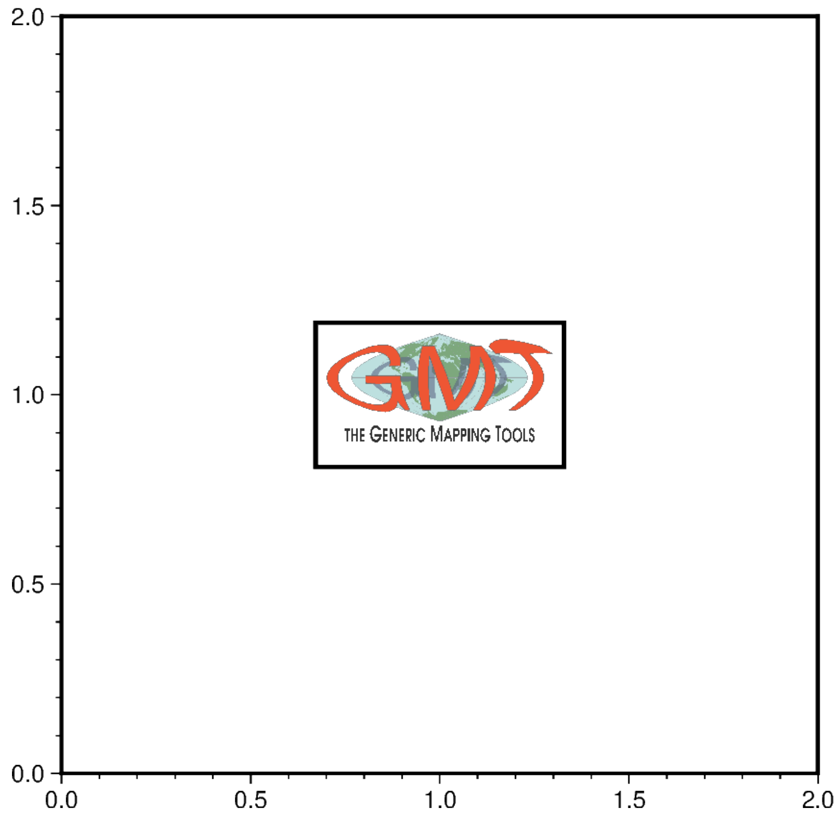
```
grdimage [WARNING]: Your CPT is categorical. Enabling -nn+a to avoid interpolation␣
↪across categories.
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()

# Define region of interest
region = [-65, -40, -40, -20]

# Assign a value of 0 for all water masses and a value of 1 for all land
# masses.
# Use shoreline data with (l)ow resolution and set the grid spacing to
# 5 arc minute in x and y direction.
grid = pygmt.grdlandmask(region=region, spacing="5m", maskvalues=[0, 1], resolution="l")

# Plot clipped grid
fig.basemap(region=region, projection="M12c", frame=True)

# Define a colormap to be used for two categories, define the range of the
# new discrete CPT using series=(lowest_value, highest_value, interval),
# use color_model="+cwater,land" to write the discrete color palette
# "batlow" in categorical format and add water/land as annotations for the
# colorbar.
pygmt.makecpt(cmap="batlow", series=(0, 1, 1), color_model="+cwater,land")

fig.grdimage(grid=grid, cmap=True)
fig.colorbar(position="JMR+o0.5c/0c+w8c")

fig.show()
```

Total running time of the script: (0 minutes 2.939 seconds)

Images on figures

The *pygmt.Figure.image* method can be used to read and place an image file in many formats (e.g., png, jpg, eps, pdf) on a figure. We must specify the filename via the `imagefile` parameter or simply use the filename as the first argument. You can also use a full URL pointing to your desired image. The `position` parameter allows us to set a reference point on the map for the image.



Out:

```
<IPython.core.display.Image object>
```

```python
import os

import pygmt

fig = pygmt.Figure()
fig.basemap(region=[0, 2, 0, 2], projection="X10c", frame=True)

# place and center the GMT logo from the GMT website to the position 1/1
# on a basemap, scaled up to be 3 cm wide and draw a rectangular border
# around the image
fig.image(
    imagefile="https://www.generic-mapping-tools.org/_static/gmt-logo.png",
    position="g1/1+w3c+jCM",
    box=True,
)

# clean up the downloaded image in the current directory
os.remove("gmt-logo.png")
```

(continues on next page)

(continued from previous page)

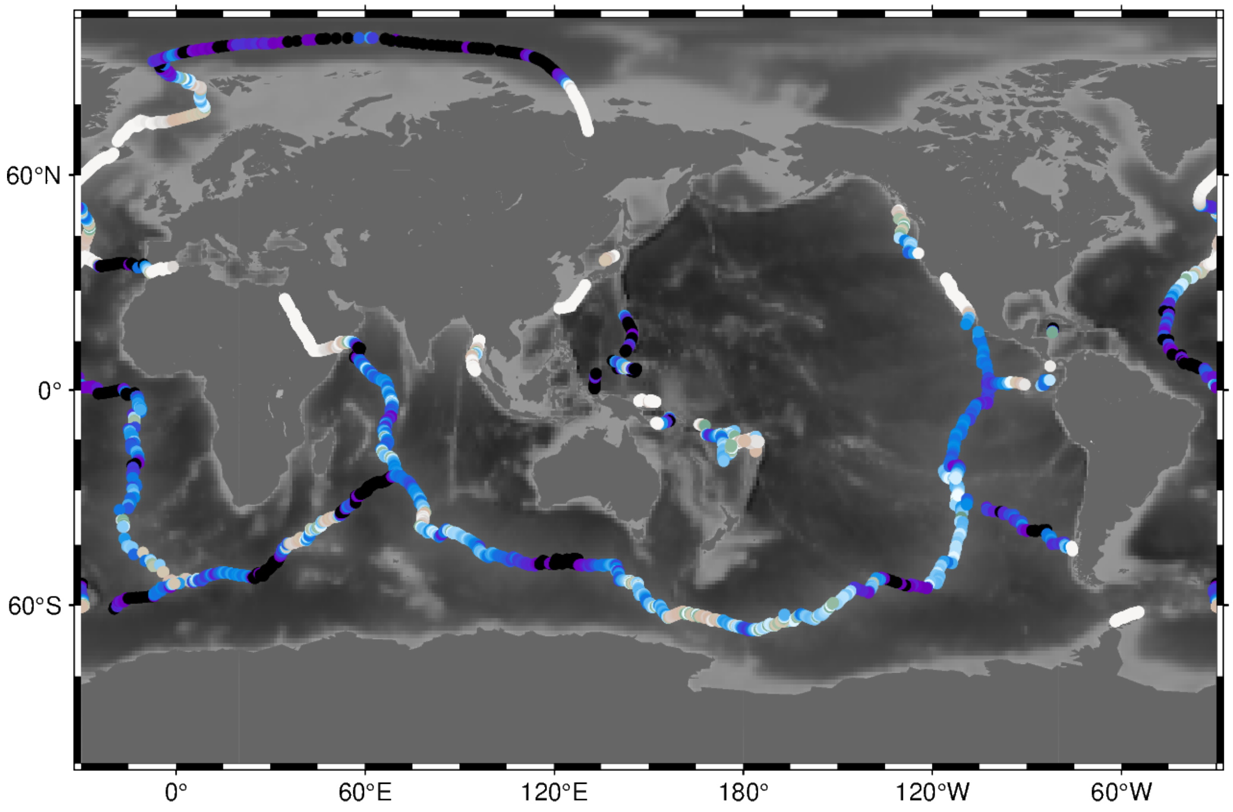
```
fig.show()
```

Total running time of the script: (0 minutes 3.172 seconds)

Sampling along tracks

The *pygmt.grdtrack* function samples a raster grid's value along specified points. We will need to input a 2D raster to `grid` which can be an `xarray.DataArray`. The argument passed to the `points` parameter can be a `pandas.DataFrame` table where the first two columns are x and y (or longitude and latitude). Note also that there is a `newcolname` parameter that will be used to name the new column of values sampled from the grid.

Alternatively, a NetCDF file path can be passed to `grid`. An ASCII file path can also be accepted for `points`. To save an output ASCII file, a file name argument needs to be passed to the `outfile` parameter.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

# Load sample grid and point datasets
grid = pygmt.datasets.load_earth_relief()
```

(continues on next page)

(continued from previous page)

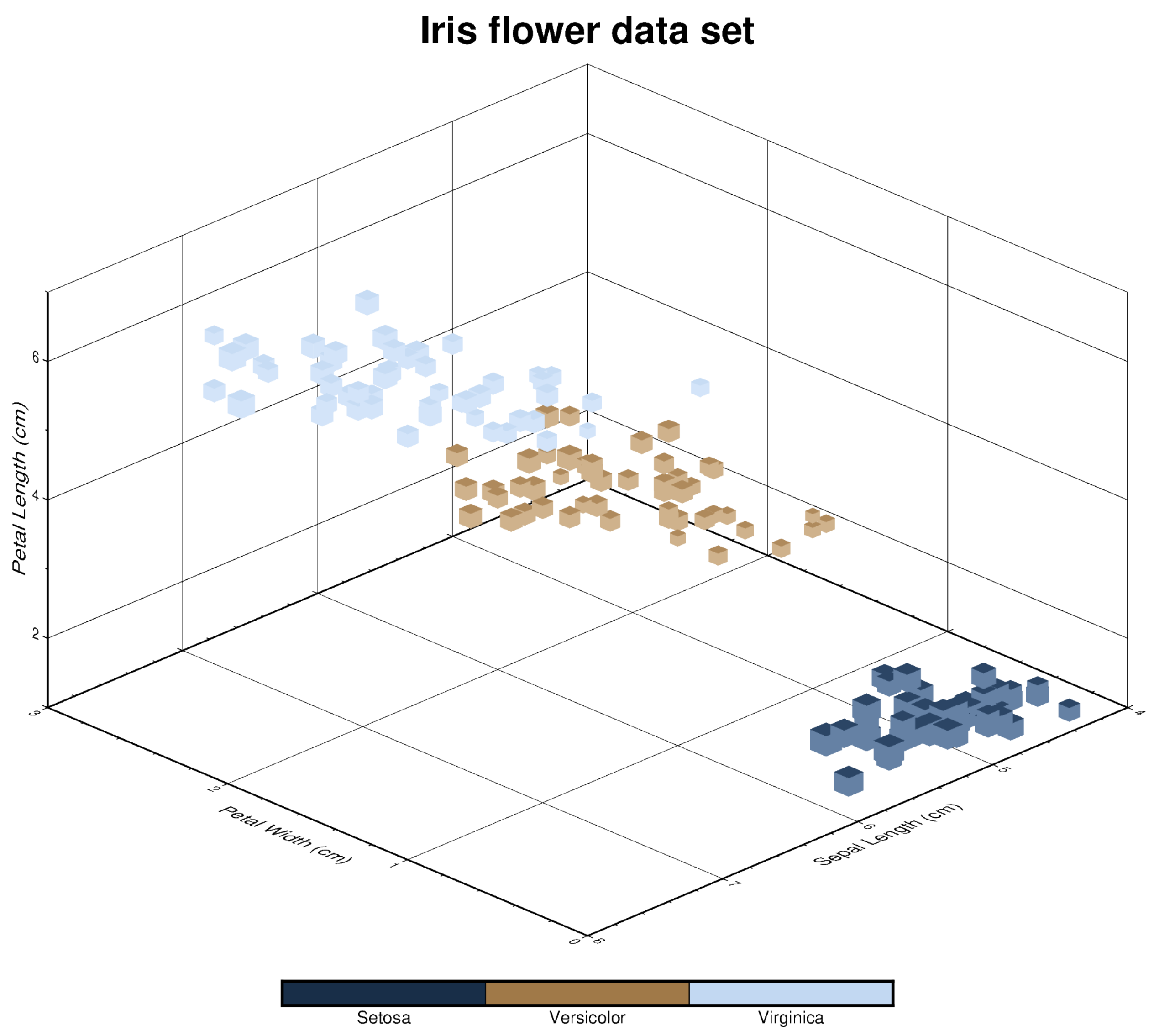
```
points = pygmt.datasets.load_ocean_ridge_points()
# Sample the bathymetry along the world's ocean ridges at specified track
# points
track = pygmt.grdtrack(points=points, grid=grid, newcolname="bathymetry")

fig = pygmt.Figure()
# Plot the earth relief grid on Cylindrical Stereographic projection, masking
# land areas
fig.basemap(region="g", projection="Cyl_stere/150/-20/15c", frame=True)
fig.grdimage(grid=grid, cmap="gray")
fig.coast(land="#666666")
# Plot the sampled bathymetry points using circles (c) of 0.15 cm size
# Points are colored using elevation values (normalized for visual purposes)
fig.plot(
    x=track.longitude,
    y=track.latitude,
    style="c0.15c",
    cmap="terra",
    color=(track.bathymetry - track.bathymetry.mean()) / track.bathymetry.std(),
)
fig.show()
```

Total running time of the script: (0 minutes 4.415 seconds)

9.4.5 3D Plots

3D Scatter plots

The *pygmt.Figure.plot3d* method can be used to plot symbols in 3D. In the example below, we show how the Iris flower dataset can be visualized using a perspective 3D plot. The `region` parameter has to include the x, y, z axis limits in the form of (xmin, xmax, ymin, ymax, zmin, zmax), which can be done automatically using *pygmt.info*. To plot the z-axis frame, set `frame` as a minimum to something like `frame=["WsNeZ", "zaf"]`. Use `perspective` to control the azimuth and elevation angle of the view, and `zscale` to adjust the vertical exaggeration factor.



Out:

```
<IPython.core.display.Image object>
```

```
import pandas as pd
import pygmt

# Load sample iris data and convert 'species' column to categorical dtype
df = pd.read_csv("https://github.com/mwaskom/seaborn-data/raw/master/iris.csv")
df.species = df.species.astype(dtype="category")

# Use pygmt.info to get region bounds (xmin, xmax, ymin, ymax, zmin, zmax)
# The below example will return a numpy array [0.0, 3.0, 4.0, 8.0, 1.0, 7.0]
```

(continues on next page)

(continued from previous page)

```
region = pygmt.info(
    data=df[["petal_width", "sepal_length", "petal_length"]],  # x, y, z columns
    per_column=True,  # report the min/max values per column as a numpy array
    # round the min/max values of the first three columns to the nearest
    # multiple of 1, 2 and 0.5, respectively
    spacing=(1, 2, 0.5),
)

# Make a 3D scatter plot, coloring each of the 3 species differently
fig = pygmt.Figure()

# Define a colormap to be used for three categories, define the range of the
# new discrete CPT using series=(lowest_value, highest_value, interval), use
# color_model="+cSetosa,Versicolor,Virginica" to write the discrete color
# palette "cubhelix" in categorical format and add the species names as
# annotations for the colorbar
pygmt.makecpt(
    cmap="cubhelix", color_model="+cSetosa,Versicolor,Virginica", series=(0, 2, 1)
)

fig.plot3d(
    # Use petal width, sepal length and petal length as x, y and z data input,
    # respectively
    x=df.petal_width,
    y=df.sepal_length,
    z=df.petal_length,
    # Vary each symbol size according to another feature (sepal width, scaled
    # by 0.1)
    size=0.1 * df.sepal_width,
    # Use 3D cubes ("u") as symbols, with size in centimeter units ("c")
    style="uc",
    # Points colored by categorical number code
    color=df.species.cat.codes.astype(int),
    # Use colormap created by makecpt
    cmap=True,
    # Set map dimensions (xmin, xmax, ymin, ymax, zmin, zmax)
    region=region,
    # Set frame parameters
    frame=[
        'WsNeZ3+t"Iris flower data set"',  # z axis label positioned on 3rd corner, add␣
↪title
        'xafg+l"Petal Width (cm)"',
        'yafg+l"Sepal Length (cm)"',
        'zafg+l"Petal Length (cm)"',
    ],
    # Set perspective to azimuth NorthWest (315°), at elevation 25°
    perspective=[315, 25],
    # Vertical exaggeration factor
    zscale=1.5,
)

# Add colorbar legend
```

(continues on next page)

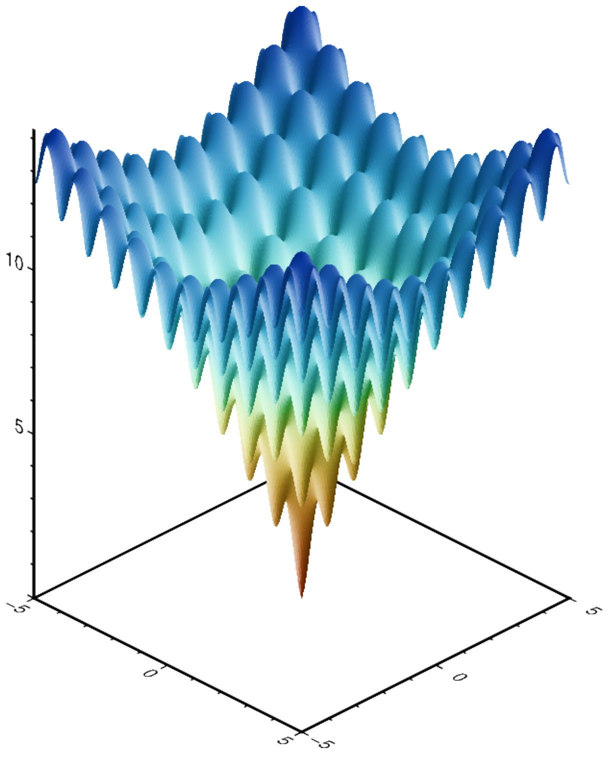
(continued from previous page)

```
fig.colorbar(xshift=3.1)

fig.show()
```

Total running time of the script: (0 minutes 4.263 seconds)

Plotting a surface

The *pygmt.Figure.grdview* method can plot 3-D surfaces with `surftype="s"`. Here, we supply the data as an `xarray.DataArray` with the coordinate vectors `x` and `y` defined. Note that the `perspective` parameter here controls the azimuth and elevation angle of the view. We provide a list of two arguments to `frame` - the first argument specifies the x- and y-axes frame attributes and the second argument, prepended with `"z"`, specifies the z-axis frame attributes. Specifying the same scale for the `projection` and `zscale` parameters ensures equal axis scaling. The `shading` parameter specifies illumination; here we choose an azimuth of 45° with `shading="+a45"`.



Out:

```
<IPython.core.display.Image object>
```

```
import numpy as np
import pygmt
import xarray as xr
```

(continues on next page)

(continued from previous page)

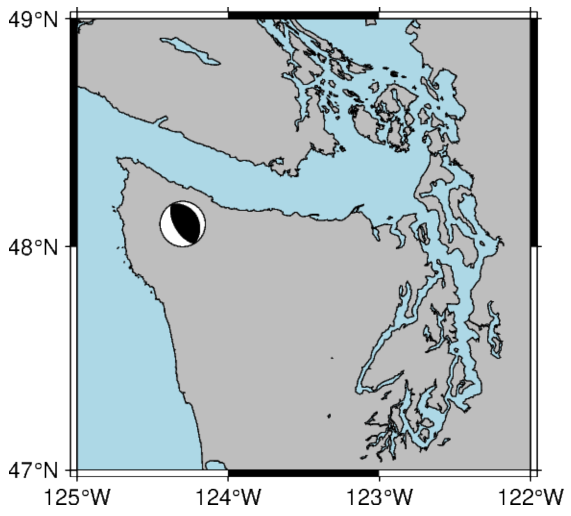
```python
# Define an interesting function of two variables, see:
# https://en.wikipedia.org/wiki/Ackley_function
def ackley(x, y):
    return (
        -20 * np.exp(-0.2 * np.sqrt(0.5 * (x ** 2 + y ** 2)))
        - np.exp(0.5 * (np.cos(2 * np.pi * x) + np.cos(2 * np.pi * y)))
        + np.exp(1)
        + 20
    )


# Create gridded data
INC = 0.05
x = np.arange(-5, 5 + INC, INC)
y = np.arange(-5, 5 + INC, INC)
data = xr.DataArray(ackley(*np.meshgrid(x, y)), coords=(x, y))

fig = pygmt.Figure()

# Plot grid as a 3-D surface
SCALE = 0.5  # in centimeter
fig.grdview(
    data,
    frame=["a5f1", "za5f1"],
    projection=f"x{SCALE}c",
    zscale=f"{SCALE}c",
    surftype="s",
    cmap="roma",
    perspective=[135, 30],  # Azimuth southeast (135°), at elevation 30°
    shading="+a45",
)

fig.show()
```

Total running time of the script: (0 minutes 16.606 seconds)

9.4.6 Seismology and geodesy

Focal mechanisms

The *pygmt.Figure.meca* method can plot focal mechanisms, or beachballs. We can specify the focal mechanism nodal planes or moment tensor components as a dict using the `spec` parameter (or they can be specified as a 1d or 2d array, or within a specified file). The size of plotted beachballs can be specified using the `scale` parameter.



Out:

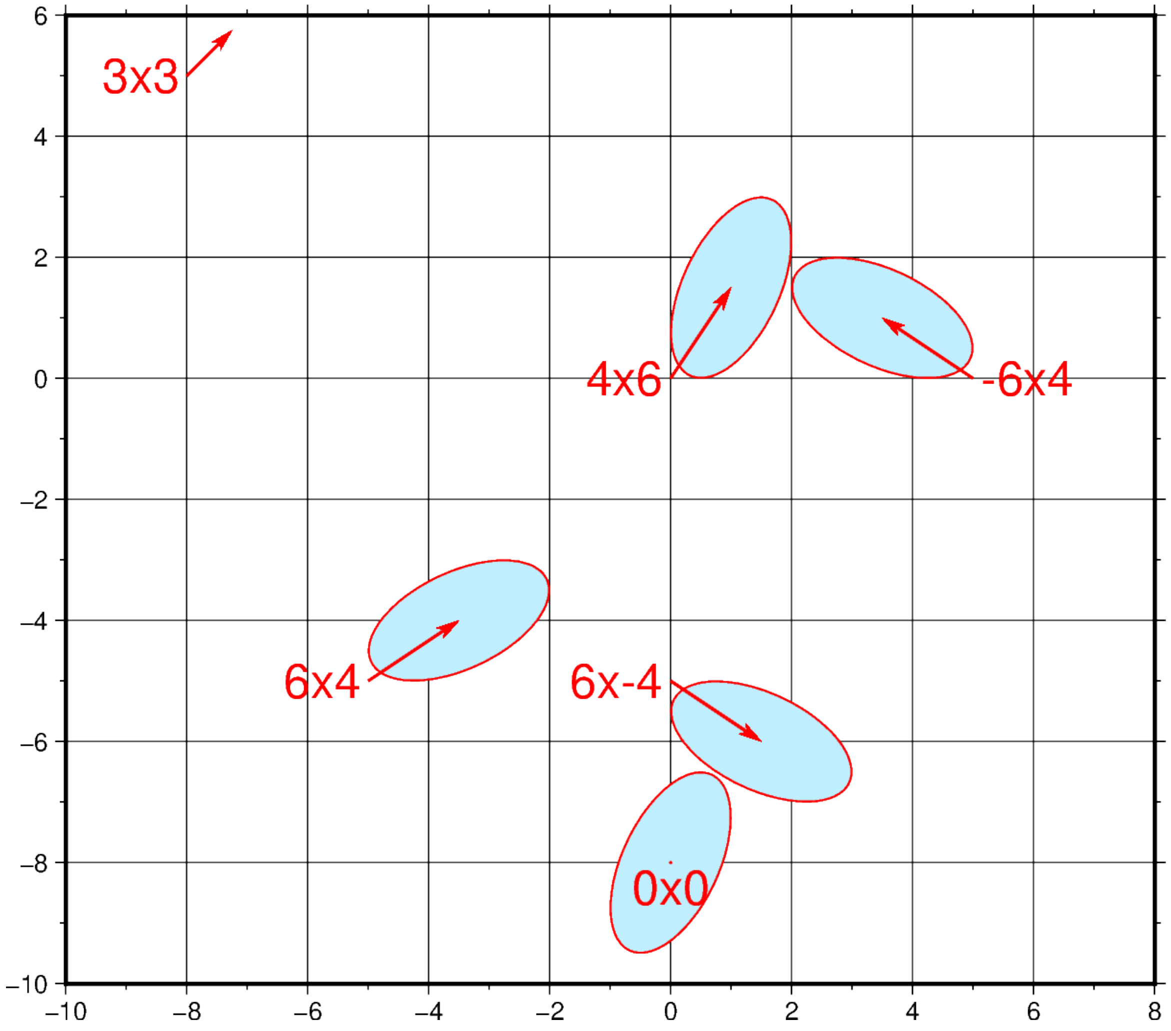
```
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()

# generate a basemap near Washington state showing coastlines, land, and water
fig.coast(
    region=[-125, -122, 47, 49],
    projection="M6c",
    land="grey",
    water="lightblue",
    shorelines=True,
    frame="a",
)

# store focal mechanisms parameters in a dict
focal_mechanism = dict(strike=330, dip=30, rake=90, magnitude=3)

# pass the focal mechanism data to meca in addition to the scale and event
# location
fig.meca(focal_mechanism, scale="1c", longitude=-124.3, latitude=48.1, depth=12.0)

fig.show()
```

Total running time of the script: (0 minutes 2.179 seconds)

Velocity arrows and confidence ellipses

The `pygmt.Figure.velo` method can be used to plot mean velocity arrows and confidence ellipses. The example below plots red velocity arrows with light-blue confidence ellipses outlined in red with the east_velocity x north_velocity used for the station names. Note that the velocity arrows are scaled by 0.2 and the 39% confidence limit will give an ellipse which fits inside a rectangle of dimension east_sigma by north_sigma.



Out:

```
<IPython.core.display.Image object>
```

```
import pandas as pd
import pygmt

fig = pygmt.Figure()
df = pd.DataFrame(
```

(continues on next page)

(continued from previous page)

```
    data={
        "x": [0, -8, 0, -5, 5, 0],
        "y": [-8, 5, 0, -5, 0, -5],
        "east_velocity": [0, 3, 4, 6, -6, 6],
        "north_velocity": [0, 3, 6, 4, 4, -4],
        "east_sigma": [4, 0, 4, 6, 6, 6],
        "north_sigma": [6, 0, 6, 4, 4, 4],
        "correlation_EN": [0.5, 0.5, 0.5, 0.5, -0.5, -0.5],
        "SITE": ["0x0", "3x3", "4x6", "6x4", "-6x4", "6x-4"],
    }
)
fig.velo(
    data=df,
    region=[-10, 8, -10, 6],
    pen="0.6p,red",
    uncertaintycolor="lightblue1",
    line=True,
    spec="e0.2/0.39/18",
    frame=["WSne", "2g2f"],
    projection="x0.8c",
    vector="0.3c+p1p+e+gred",
)

fig.show()
```

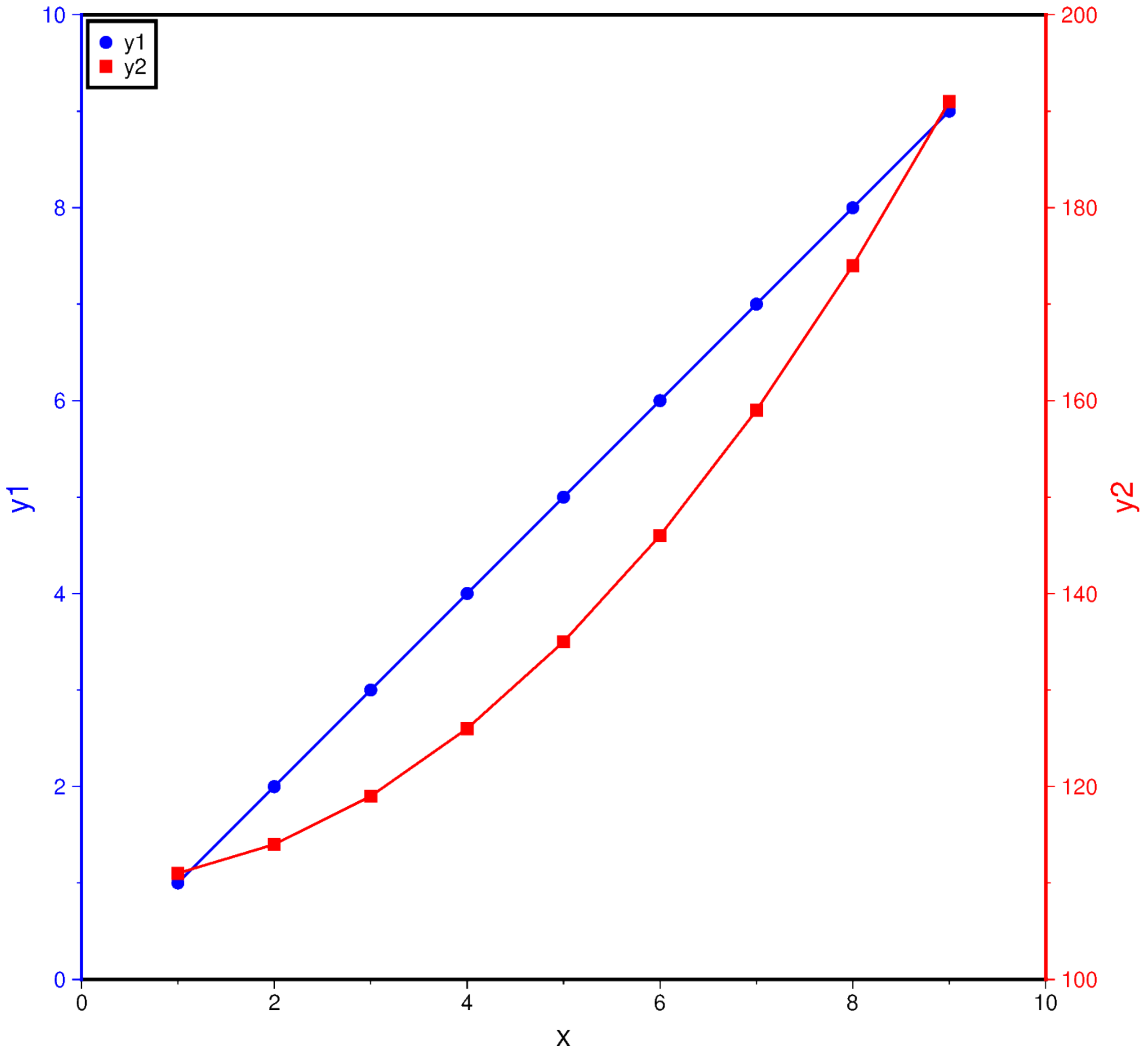
Total running time of the script: (0 minutes 2.024 seconds)

9.4.7 Base maps

Double Y axes graph

The `frame` parameter of the plotting methods of the *`pygmt.Figure`* class can control which axes should be plotted and optionally show annotations, tick marks, and gridlines. By default, all 4 axes are plotted, along with annotations and tick marks (denoted **W**, **S**, **E**, **N**). Lower case versions (**w**, **s**, **e**, **n**) can be used to denote to only plot the axes with tick marks. We can also only plot the axes without annotations and tick marks using **l** (left axis), **r** (right axis), **t** (top axis), **b** (bottom axis). When `frame` is used to change the frame settings, any axes that are not defined using one of these three options are not drawn.

To plot a double Y-axes graph using PyGMT, we need to plot at least two base maps separately. The base maps should share the same projection parameter and x-axis limits, but different y-axis limits.



Out:

```
<IPython.core.display.Image object>
```

```
import numpy as np
import pygmt

# Generate two sample Y data from one common X data
x = np.linspace(1.0, 9.0, num=9)
y1 = x
y2 = x ** 2 + 110
```

(continues on next page)

(continued from previous page)

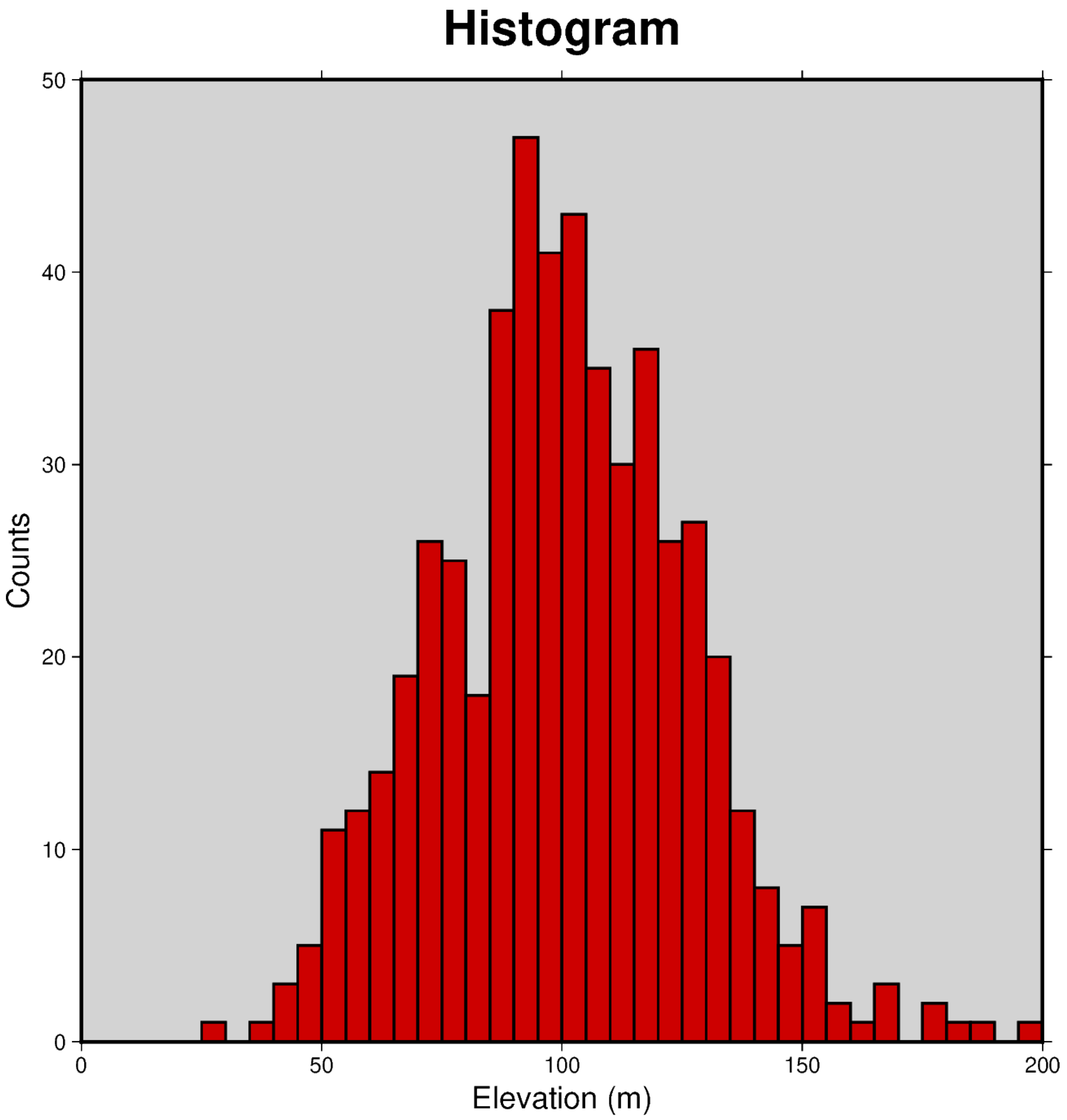
```
fig = pygmt.Figure()

# Plot the common X axes
# The bottom axis (S) is plotted with annotations and tick marks
# The top axis (t) is plotted without annotations and tick marks
# The left and right axes are not drawn
fig.basemap(region=[0, 10, 0, 10], projection="X15c/15c", frame=["St", "xaf+lx"])

# Plot the Y axis for y1 data
# The left axis (W) is plotted with blue annotations, ticks, and label
with pygmt.config(
    MAP_FRAME_PEN="blue",
    MAP_TICK_PEN="blue",
    FONT_ANNOT_PRIMARY="blue",
    FONT_LABEL="blue",
):
    fig.basemap(frame=["W", "yaf+ly1"])

# Plot the line for y1 data
fig.plot(x=x, y=y1, pen="1p,blue")
# Plot points for y1 data
fig.plot(x=x, y=y1, style="c0.2c", color="blue", label="y1")

# Plot the Y axis for y2 data
# The right axis (E) is plotted with red annotations, ticks, and label
with pygmt.config(
    MAP_FRAME_PEN="red",
    MAP_TICK_PEN="red",
    FONT_ANNOT_PRIMARY="red",
    FONT_LABEL="red",
):
    fig.basemap(region=[0, 10, 100, 200], frame=["E", "yaf+ly2"])
# Plot the line for y2 data
fig.plot(x=x, y=y2, pen="1p,red")
# Plot points for y2 data
fig.plot(x=x, y=y2, style="s0.28c", color="red", label="y2")

# Create a legend in the top-left corner of the plot
fig.legend(position="jTL+o0.1c", box=True)

fig.show()
```

Total running time of the script: (0 minutes 4.773 seconds)

9.4.8 Histograms

Histogram

The *pygmt.Figure.histogram* method can plot regular histograms. Using the `series` parameter allows to set the interval for the width of each bar. The type of histogram (frequency count or percentage) can be selected via the `histtype` parameter.



Out:

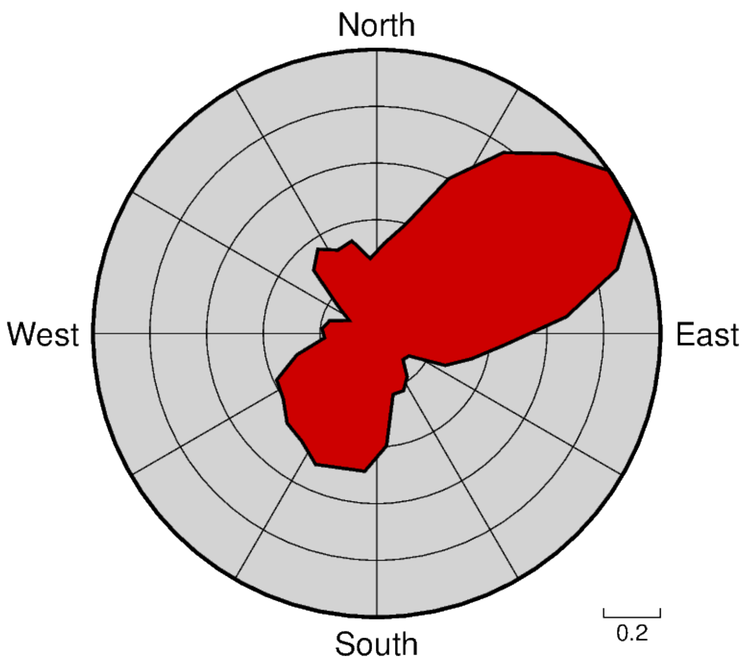
```
<IPython.core.display.Image object>
```

```
import numpy as np
import pygmt

np.random.seed(100)

# generate random elevation data from a normal distribution
mean = 100  # mean of distribution
stddev = 25  # standard deviation of distribution
data = mean + stddev * np.random.randn(521)

fig = pygmt.Figure()

fig.histogram(
    data=data,
    # define the frame, add title and set background color to
    # lightgray, add annotations for x and y axis
    frame=['WSne+t"Histogram"+glightgray', 'x+l"Elevation (m)"', 'y+l"Counts"'],
    # generate evenly spaced bins by increments of 5
    series=5,
    # use red3 as color fill for the bars
    fill="red3",
    # use a pen size of 1p to draw the outlines
    pen="1p",
    # choose histogram type 0 = counts [default]
    histtype=0,
)

fig.show()
```

Total running time of the script: (0 minutes 2.525 seconds)

Rose diagram

The *pygmt.Figure.rose* method can plot windrose diagrams or polar histograms.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

# Load sample compilation of fracture lengths and azimuth as
# hypothetically digitized from geological maps
data = pygmt.datasets.load_fractures_compilation()

fig = pygmt.Figure()

fig.rose(
    # use columns of the sample dataset as input for the length and azimuth
    # parameters
    length=data.length,
    azimuth=data.azimuth,
    # specify the "region" of interest in the (r,azimuth) space
    # [r0, r1, az0, az1], here, r0 is 0 and r1 is 1, for azimuth, az0 is 0 and
    # az1 is 360 which means we plot a full circle between 0 and 360 degrees
    region=[0, 1, 0, 360],
    # set the diameter of the rose diagram to 7.5 cm
    diameter="7.5c",
    # define the sector width in degrees, we append +r here to draw a rose
    # diagram instead of a sector diagram
    sector="10+r",
    # normalize bin counts by the largest value so all bin counts range from
    # 0 to 1
```

(continues on next page)

(continued from previous page)

```
    norm=True,
    # use red3 as color fill for the sectors
    color="red3",
    # define the frame with ticks and gridlines every 0.2
    # length unit in radial direction and every 30 degrees
    # in azimuthal direction, set background color to
    # lightgray
    frame=["x0.2g0.2", "y30g30", "+glightgray"],
    # use a pen size of 1p to draw the outlines
    pen="1p",
)

fig.show()
```

Total running time of the script: (0 minutes 1.912 seconds)

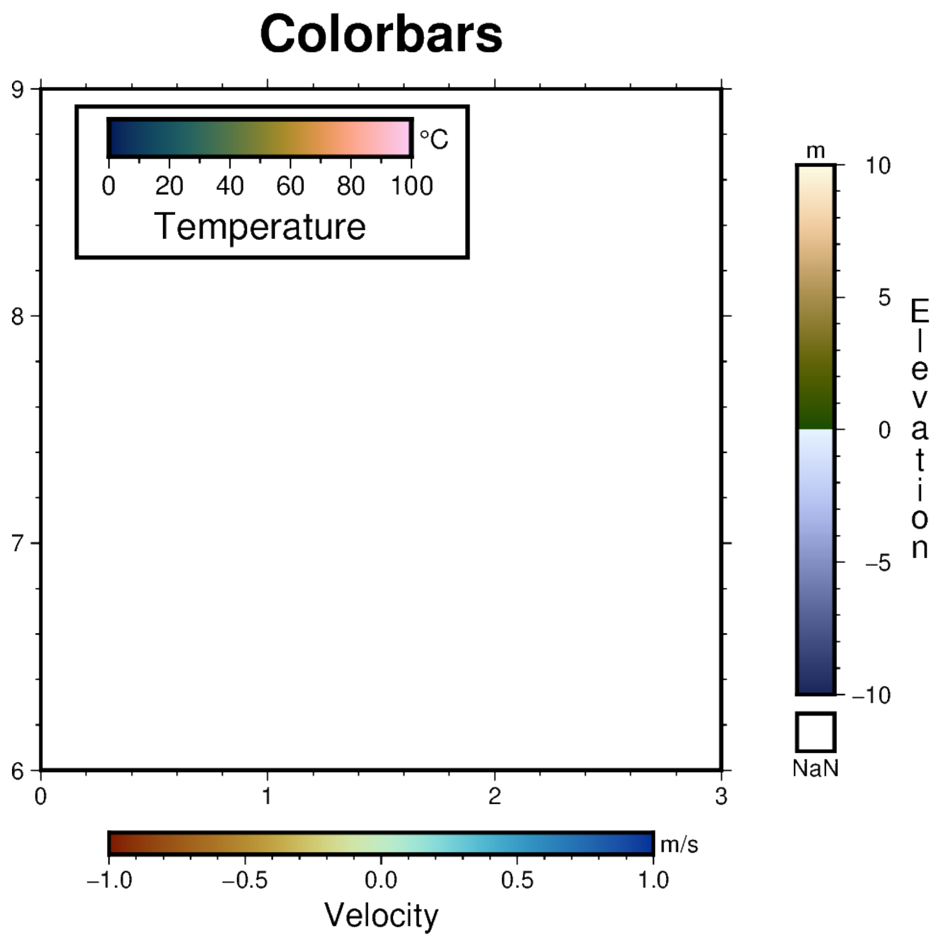
9.4.9 Plot embellishments

Colorbar

The *pygmt.Figure.colorbar* method creates a color scalebar. We must specify the colormap via the `cmap` parameter, and optionally set the placement via the `position` parameter. The full list of color palette tables can be found at https://docs.generic-mapping-tools.org/latest/cookbook/cpts.html. You can set the `position` of the colorbar using the following options:

- **j/J**: justified inside/outside the map frame using any 2 character combination of vertical (**T**op, **M**iddle, **B**ottom) and horizontal (**L**eft, **C**enter, **R**ight) alignment codes, e.g. `position="jTR"` for top right.
- **g**: using map coordinates, e.g. `position="g170/-45"` for longitude 170E, latitude 45S.
- **x**: using paper coordinates, e.g. `position="x5c/7c"` for 5 cm,7 cm from anchor point.
- **n**: using normalized (0-1) coordinates, e.g. `position="n0.4/0.8"`.

Note that the anchor point defaults to the bottom left (**BL**). Append `+h` to `position` to get a horizontal colorbar instead of a vertical one.



Out:

```
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()
fig.basemap(region=[0, 3, 6, 9], projection="x3c", frame=["af", 'WSne+t"Colorbars"'])

## Create a colorbar designed for seismic tomography - roma
# Colorbar is placed at bottom center (BC) by default if no position is given
fig.colorbar(cmap="roma", frame=["x+lVelocity", "y+lm/s"])

## Create a colorbar showing the scientific rainbow - batlow
fig.colorbar(
    cmap="batlow",
    # Colorbar positioned at map coordinates (g) longitude/latitude 0.3/8.7,
    # with a length/width (+w) of 4 cm by 0.5 cm, and plotted horizontally (+h)
    position="g0.3/8.7+w4c/0.5c+h",
```

(continues on next page)

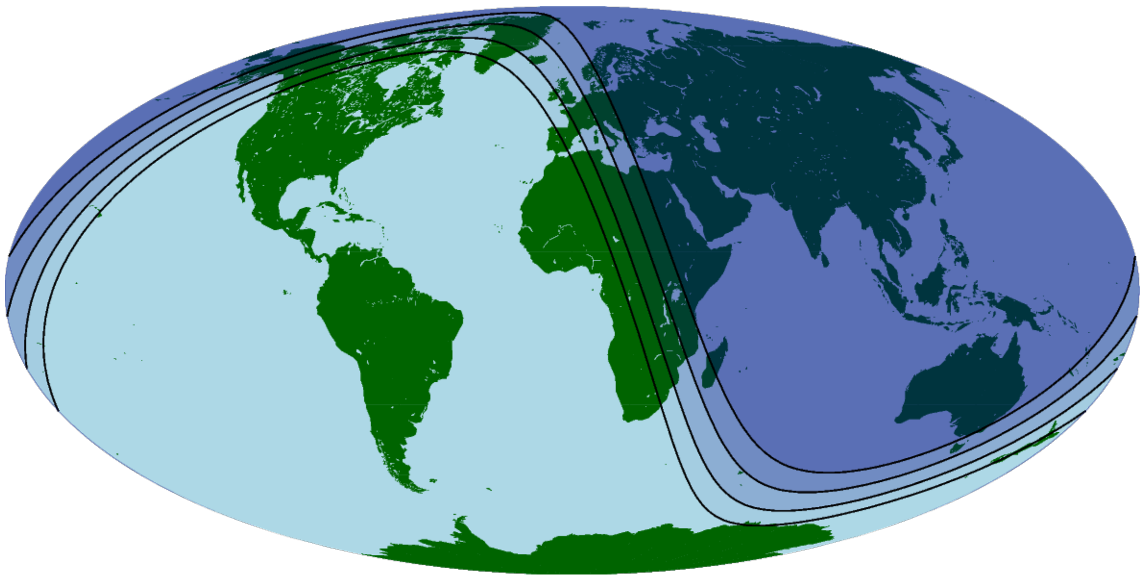
(continued from previous page)

```
    box=True,
    frame=["x+lTemperature", r"y+l\260C"],
    scale=100,
)

## Create a colorbar suitable for surface topography - oleron
fig.colorbar(
    cmap="oleron",
    # Colorbar position justified outside map frame (J) at Middle Right (MR),
    # offset (+o) by 1 cm horizontally and 0 cm vertically from anchor point,
    # with a length/width (+w) of 7 cm by 0.5 cm and a box for NaN values (+n)
    position="JMR+o1c/0c+w7c/0.5c+n+mc",
    # Note that the label 'Elevation' is moved to the opposite side and plotted
    # vertically as a column of text using '+mc' in the position parameter
    # above
    frame=["x+lElevation", "y+lm"],
    scale=10,
)

fig.show()
```

Total running time of the script: (0 minutes 2.711 seconds)

Day-night terminator line and twilights

Use *pygmt.Figure.solar* to show the different transition stages between daytime and nighttime. The parameter `terminator` is used to set the twilight stage, and can be either 'day-night' (brightest), 'civil', 'nautical', or 'astronomical' (darkest). Refer to https://en.wikipedia.org/wiki/Twilight for more information.



Out:

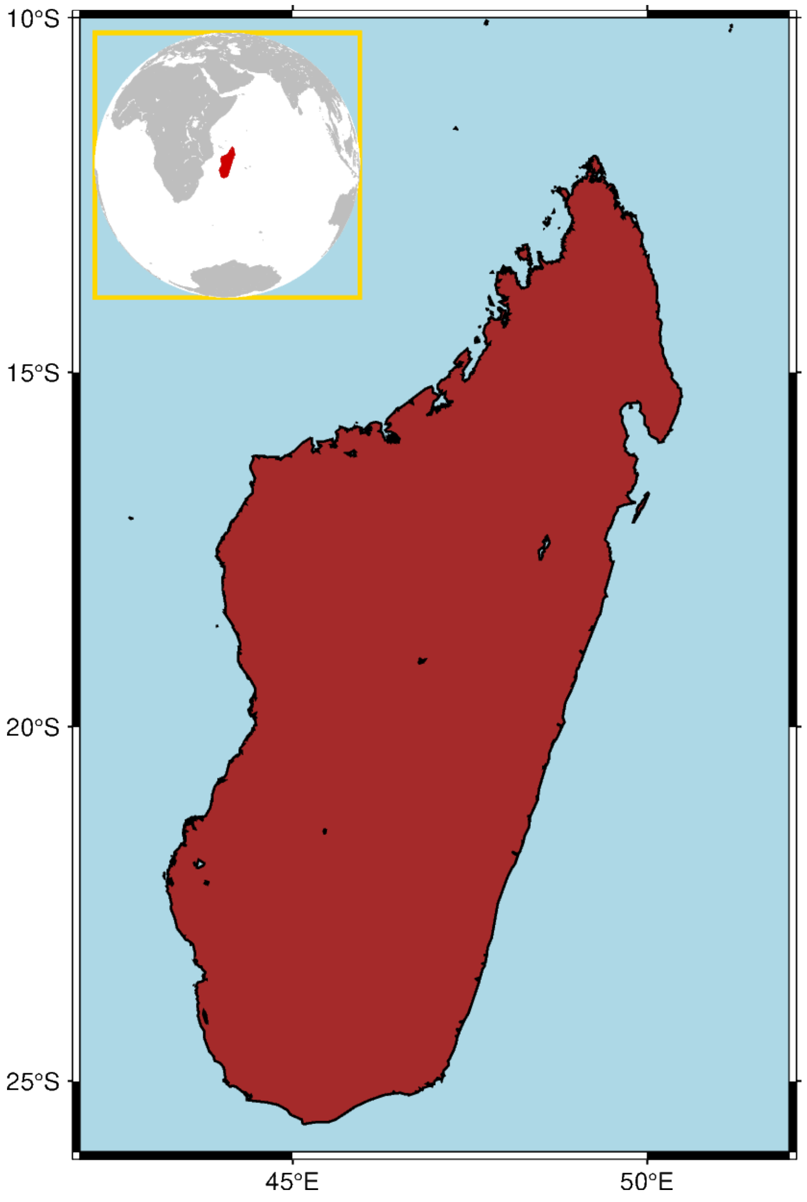
```
<IPython.core.display.Image object>
```

```python
import datetime

import pygmt

fig = pygmt.Figure()
# Create a figure showing the global region on a Mollweide projection
# Land color is set to dark green and water color is set to light blue
fig.coast(region="d", projection="W0/15c", land="darkgreen", water="lightblue")
# Set a time for the day-night terminator and twilights, 1700 UTC on
# January 1, 2000
terminator_datetime = datetime.datetime(
    year=2000, month=1, day=1, hour=17, minute=0, second=0
)
# Set the pen line to be 0.5p thick
# Set the fill for the night area to be navy blue at different transparency
# levels
fig.solar(
    terminator="day_night",
    terminator_datetime=terminator_datetime,
    fill="navyblue@95",
    pen="0.5p",
)
fig.solar(
    terminator="civil",
    terminator_datetime=terminator_datetime,
    fill="navyblue@85",
    pen="0.5p",
)
fig.solar(
    terminator="nautical",
    terminator_datetime=terminator_datetime,
    fill="navyblue@80",
    pen="0.5p",
)
fig.solar(
    terminator="astronomical",
    terminator_datetime=terminator_datetime,
    fill="navyblue@80",
    pen="0.5p",
)
fig.show()
```

Total running time of the script: (0 minutes 4.409 seconds)

Inset

The *pygmt.Figure.inset* method adds an inset figure inside a larger figure. The function is called using a `with` statement, and its `position`, `box`, `offset`, and `margin` parameters are set. Plotting methods called within the `with` statement are applied to the inset figure.



Out:

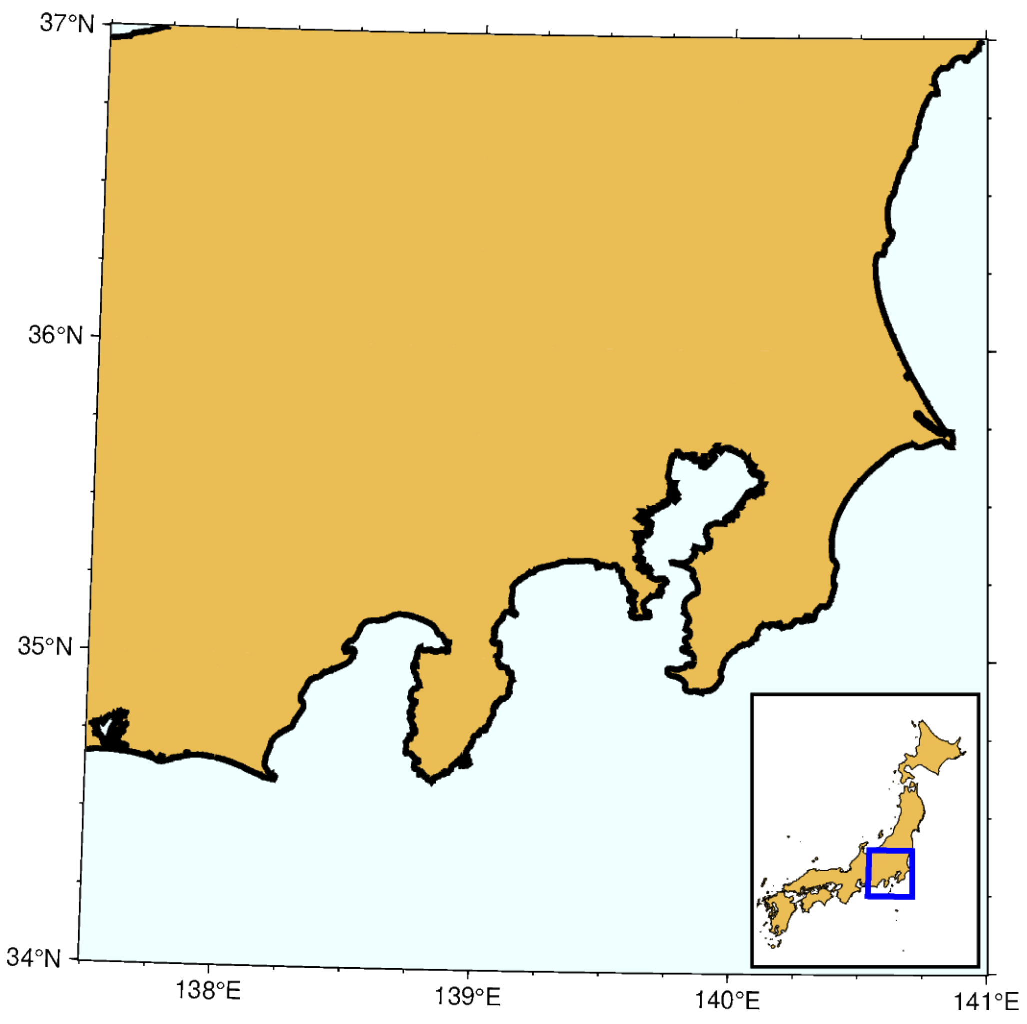
```
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()
# Create the primary figure, setting the region to Madagascar, the land color
# to "brown", the water to "lightblue", the shorelines width to "thin", and
# adding a frame
fig.coast(region="MG+r2", land="brown", water="lightblue", shorelines="thin", frame="a")
# Create an inset, setting the position to top left, the width to 3.5 cm, and
# the x- and y-offsets to 0.2 cm. The margin is set to 0, and the border is
# "gold" with a pen size of 1.5p.
with fig.inset(position="jTL+w3.5c+o0.2c", margin=0, box="+p1.5p,gold"):
    # Create a figure in the inset using coast. This example uses the azimuthal
    # orthogonal projection centered at 47E, 20S. The land color is set to
    # "gray" and Madagascar is highlighted in "red3".
    fig.coast(
        region="g",
        projection="G47/-20/?",
        land="gray",
        water="white",
        dcw="MG+gred3",
    )
fig.show()
```

Total running time of the script: (0 minutes 5.034 seconds)

Inset map showing a rectangular region

The *pygmt.Figure.inset* method adds an inset figure inside a larger figure. The function is called using a `with` statement, and its `position`, `box`, `offset`, and `margin` can be customized. Plotting methods called within the `with` statement plot into the inset figure.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

# Set the region of the main figure
region = [137.5, 141, 34, 37]

fig = pygmt.Figure()

# Plot the base map of the main figure. Universal Transverse Mercator (UTM)
# projection is used and the UTM zone is set to be "54S".
fig.basemap(region=region, projection="U54S/12c", frame=["WSne", "af"])

# Set the land color to "lightbrown", the water color to "azure1", the
# shoreline width to "2p", and the area threshold to 1000 km^2 for the main
```

(continues on next page)

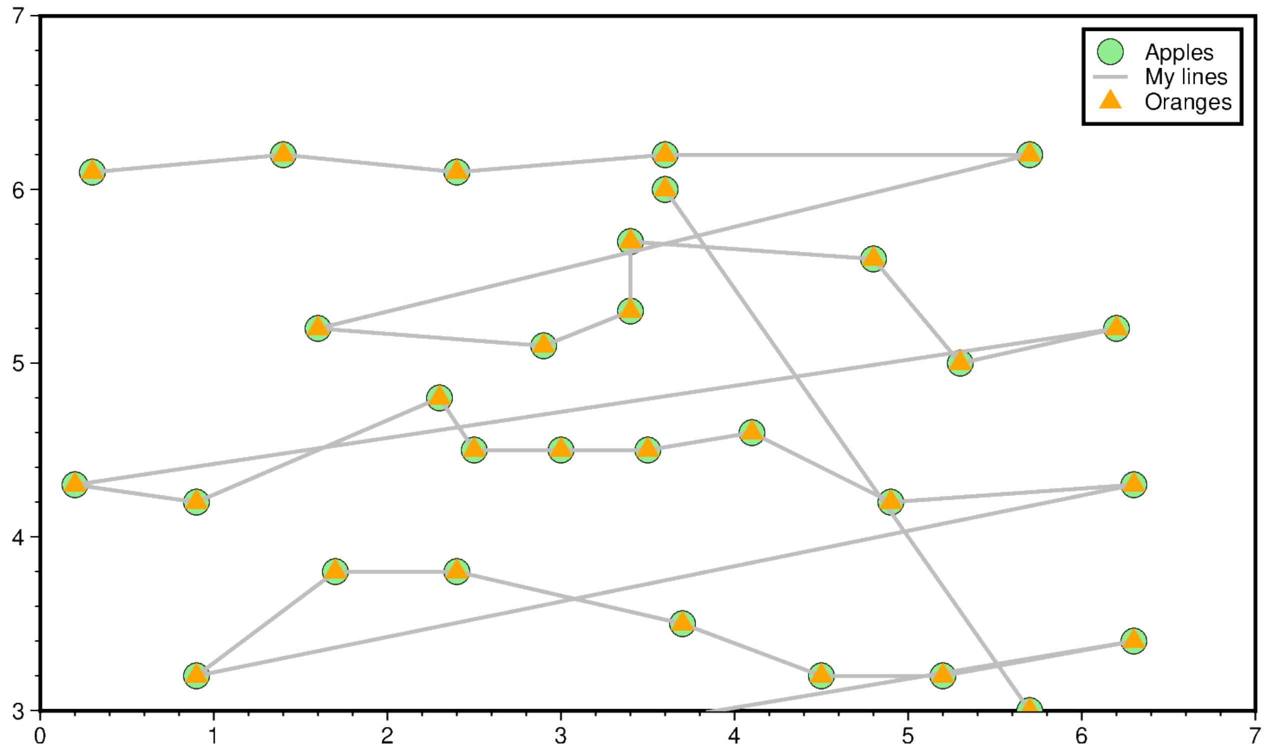
(continued from previous page)

```python
# figure
fig.coast(land="lightbrown", water="azure1", shorelines="2p", area_thresh=1000)

# Create an inset map, setting the position to bottom right, the width to
# 3 cm, the height to 3.6 cm, and the x- and y-offsets to
# 0.1 cm, respectively. Draws a rectangular box around the inset with a fill
# color of "white" and a pen of "1p".
with fig.inset(position="jBR+w3c/3.6c+o0.1c", box="+gwhite+p1p"):
    # Plot the Japan main land in the inset using coast. "U54S/?" means UTM
    # projection with map width automatically determined from the inset width.
    # Highlight the Japan area in "lightbrown"
    # and draw its outline with a pen of "0.2p".
    fig.coast(
        region=[129, 146, 30, 46],
        projection="U54S/?",
        dcw="JP+glightbrown+p0.2p",
        area_thresh=10000,
    )
    # Plot a rectangle ("r") in the inset map to show the area of the main
    # figure. "+s" means that the first two columns are the longitude and
    # latitude of the bottom left corner of the rectangle, and the last two
    # columns the longitude and latitude of the uppper right corner.
    rectangle = [[region[0], region[2], region[1], region[3]]]
    fig.plot(data=rectangle, style="r+s", pen="2p,blue")

fig.show()
```

Total running time of the script: (0 minutes 4.519 seconds)

Legend

The *pygmt.Figure.legend* method can automatically create a legend for symbols plotted using *pygmt.Figure.plot*. Legend entries are only created when the `label` parameter is used.



Out:

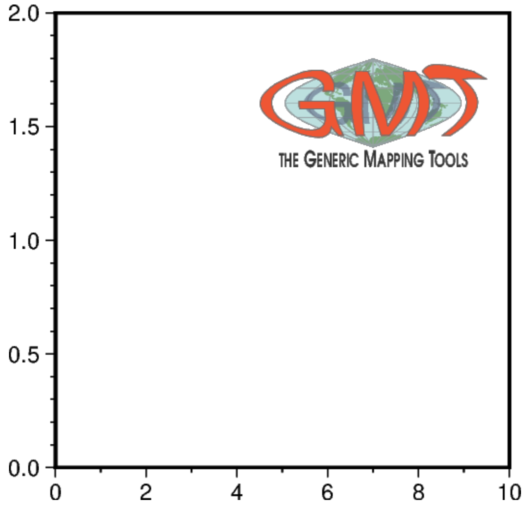
```
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()

fig.basemap(projection="x1i", region=[0, 7, 3, 7], frame=True)

fig.plot(
    data="@Table_5_11.txt",
    style="c0.15i",
    color="lightgreen",
    pen="faint",
    label="Apples",
)
fig.plot(data="@Table_5_11.txt", pen="1.5p,gray", label='"My lines"')
fig.plot(data="@Table_5_11.txt", style="t0.15i", color="orange", label="Oranges")

fig.legend(position="JTR+jTR+o0.2c", box=True)

fig.show()
```

Total running time of the script: (0 minutes 3.267 seconds)

Logo

The *pygmt.Figure.logo* method allows to place the GMT logo on a map.



Out:

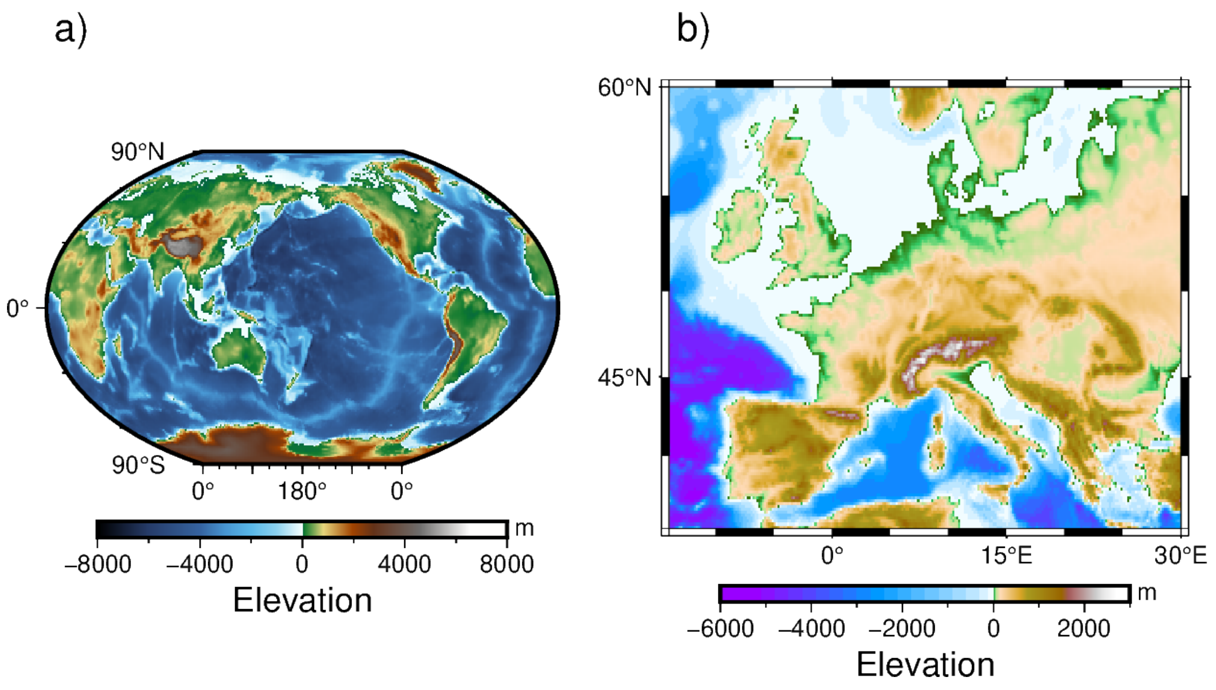
```
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()
fig.basemap(region=[0, 10, 0, 2], projection="X6c", frame=True)

# add the GMT logo in the Top Right corner of the current map,
# scaled up to be 3 cm wide and offset by 0.3 cm in X direction
# and 0.6 cm in Y direction.
fig.logo(position="jTR+o0.3c/0.6c+w3c")

fig.show()
```

Total running time of the script: (0 minutes 2.520 seconds)

Multiple colormaps

This gallery example shows how to create multiple colormaps for different subplots. To better understand how GMT modern mode maintains several levels of colormaps, please refer to https://docs.generic-mapping-tools.org/latest/cookbook/features.html#gmt-modern-mode-hierarchical-levels for details.



Out:

```
grdimage [WARNING]: 2 annotations along the bottom border were skipped due to crowding.
grdimage [WARNING]: Crowding decisions is controlled by MAP_ANNOT_MIN_SPACING, currently␣
↪set to 25.5563p.
grdimage [WARNING]: Decrease or increase MAP_ANNOT_MIN_SPACING to see more or fewer␣
↪annotations, with 0 showing all annotations.
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()

# Load Earth relief data for the entire globe and a subset region
grid_globe = pygmt.datasets.load_earth_relief(resolution="01d")
subset_region = [-14, 30, 35, 60]
grid_subset = pygmt.datasets.load_earth_relief(resolution="10m", region=subset_region)

# Define a 1-row, 2-column subplot layout. The overall figure dimensions is set
# to be 15 cm wide and 8 cm high. Each subplot is automatically labelled.
# The space between the subplots is set to be 0.5 cm.
with fig.subplot(
    nrows=1, ncols=2, figsize=("15c", "8c"), autolabel=True, margins="0.5c"
):
    # Activate the first panel so that the colormap created by the makecpt
    # method is a panel-level CPT
    with fig.set_panel(panel=0):
```

(continues on next page)

(continued from previous page)

```
        pygmt.makecpt(cmap="geo", series=[-8000, 8000])
        # "R?" means Winkel Tripel projection with map width automatically
        # determined from the subplot width.
        fig.grdimage(grid=grid_globe, projection="R?", region="g", frame=True)
        fig.colorbar(frame=["a4000f2000", "x+lElevation", "y+lm"])
    # Activate the second panel so that the colormap created by the makecpt
    # method is a panel-level CPT
    with fig.set_panel(panel=1):
        pygmt.makecpt(cmap="globe", series=[-6000, 3000])
        # "M?" means Mercator projection with map width also automatically
        # determined from the subplot width.
        fig.grdimage(
            grid=grid_subset, projection="M?", region=subset_region, frame=True
        )
        fig.colorbar(frame=["a2000f1000", "x+lElevation", "y+lm"])

fig.show()
```

Total running time of the script: (0 minutes 4.987 seconds)

9.5 External Resources

Below is a curated collection of external PyGMT resources.

To add your contribution to this collection, follow these instructions to submit a pull request with your recommended addition to the External Resources file.

9.5.1 Tutorials

- 2021 PyGMT course at the UAF Geophysical Institute by Liam Toney
- Remote Online Sessions for Emerging Seismologists (ROSES): Unit 8 - PyGMT by Liam Toney
- PyGMT Tutorial in 2021 by MIGG-NTU
- PyGMT Workshop at FOSS4G Oceania 2019 by Wei Ji Leong

9.5.2 Examples from Publications and Posters

- GMT and PyGMT plotting examples by Michael Grund
- NZ Antarctic Science Conference 2021 poster by Wei Ji Leong

9.6 Frames, ticks, titles, and labels

Setting the style of the map frames, ticks, etc, is handled by the `frame` parameter that all plotting methods of *`pygmt.Figure`*.

```
import pygmt
```

9.6.1 Plot frame

By default, PyGMT does not add a frame to your plot. For example, we can plot the coastlines of the world with a Mercator projection:

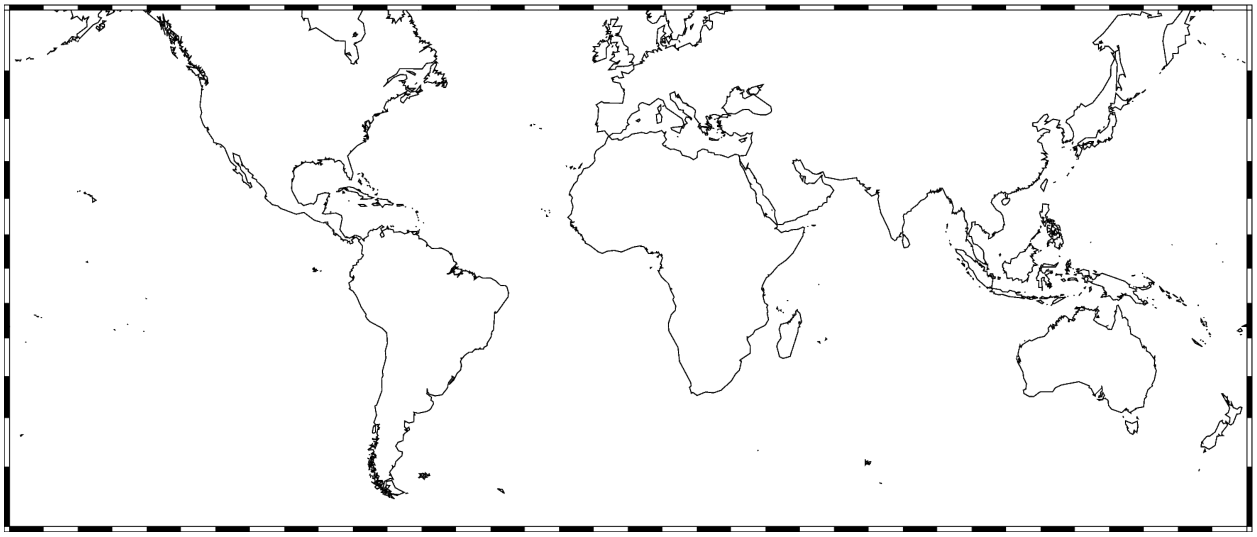
```
fig = pygmt.Figure()
fig.coast(shorelines="1/0.5p", region=[-180, 180, -60, 60], projection="M25c")
fig.show()
```

Out:

```
<IPython.core.display.Image object>
```

To add the default GMT frame to the plot, use `frame="f"` in *`pygmt.Figure.basemap`* or any other plotting module:

```
fig = pygmt.Figure()
fig.coast(shorelines="1/0.5p", region=[-180, 180, -60, 60], projection="M25c")
fig.basemap(frame="f")
fig.show()
```



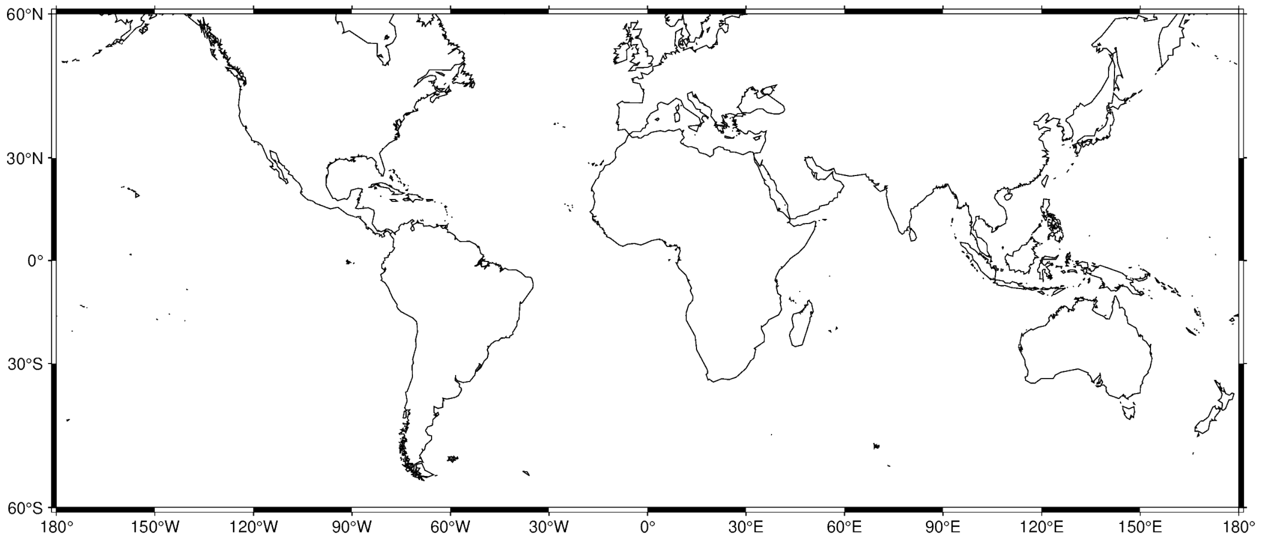
Out:

```
<IPython.core.display.Image object>
```

9.6.2 Ticks and grid lines

The automatic frame (`frame=True` or `frame="a"`) sets the default GMT style frame and automatically determines tick labels from the plot region.

```
fig = pygmt.Figure()
fig.coast(shorelines="1/0.5p", region=[-180, 180, -60, 60], projection="M25c")
fig.basemap(frame="a")
fig.show()
```

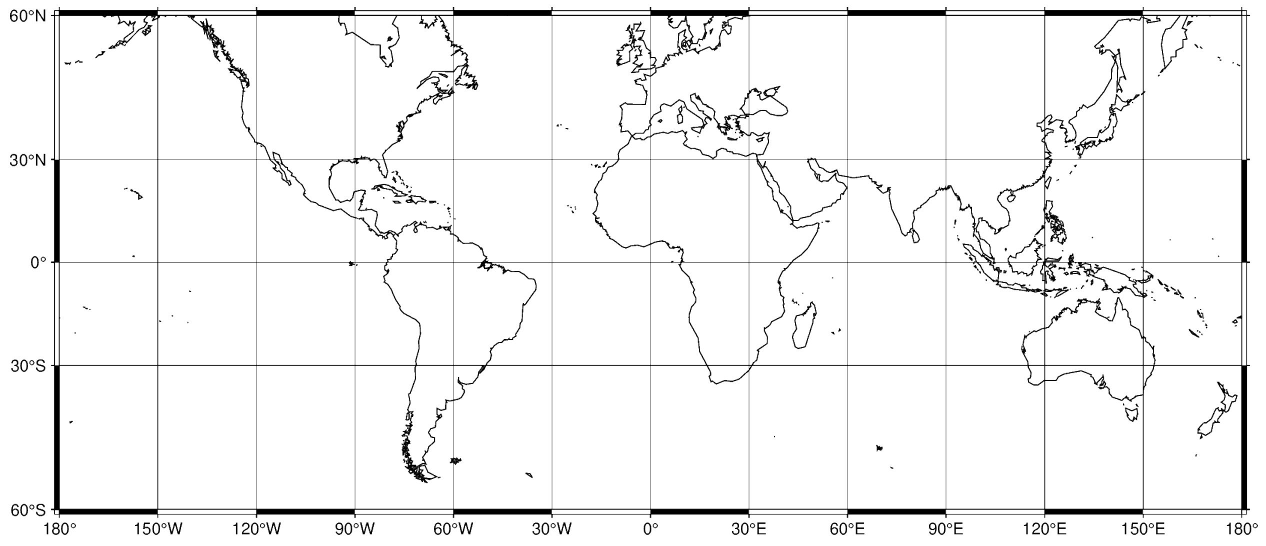


Out:

```
<IPython.core.display.Image object>
```

Add automatic grid lines to the plot by adding a `g` to `frame`:

```
fig = pygmt.Figure()
fig.coast(shorelines="1/0.5p", region=[-180, 180, -60, 60], projection="M25c")
fig.basemap(frame="ag")
fig.show()
```



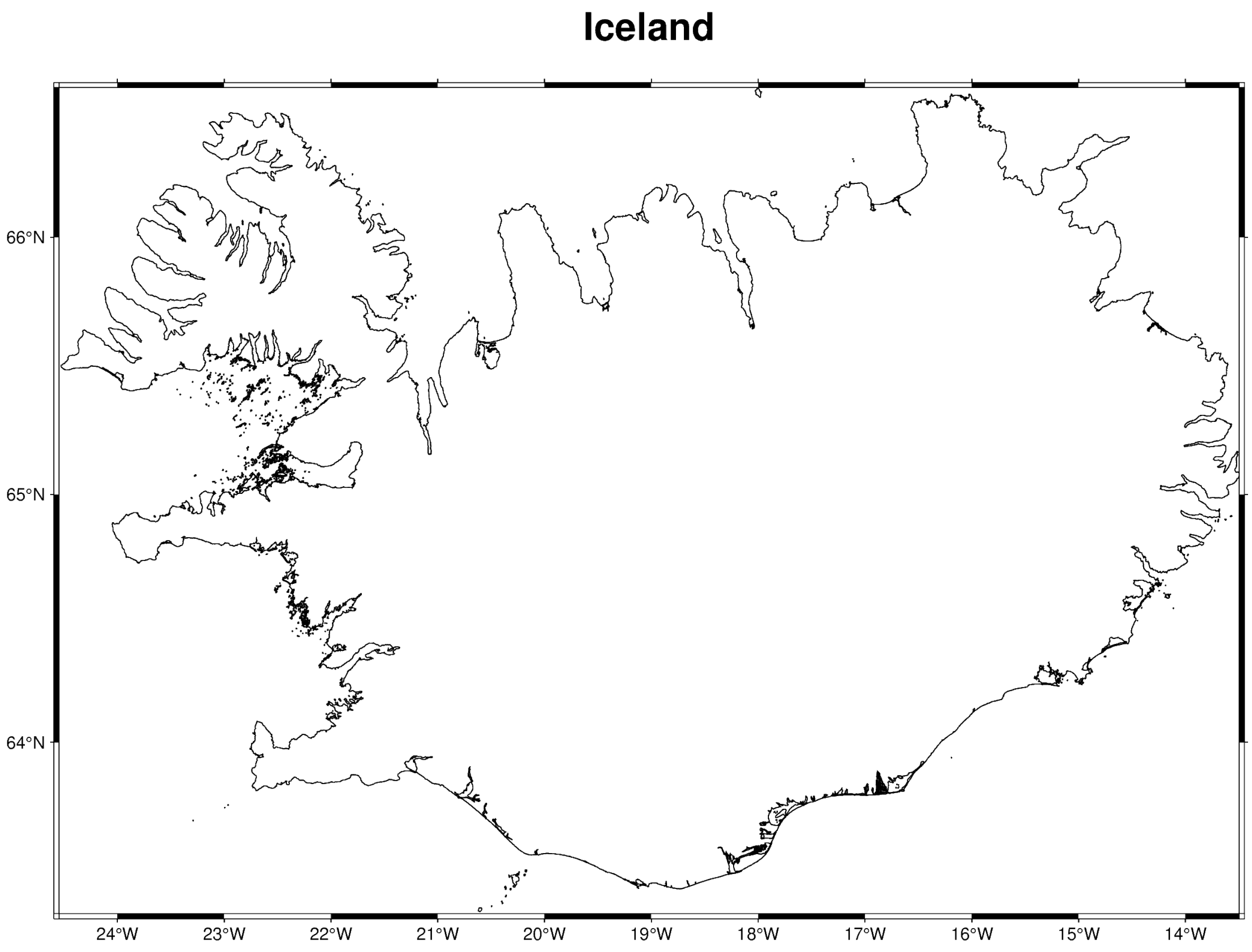
Out:

```
<IPython.core.display.Image object>
```

9.6.3 Title

The figure title can be set by passing **+t***title* to the `frame` parameter of *`pygmt.Figure.basemap`*. Passing multiple arguments to `frame` can be done by using a list, as show in the example below.

```
fig = pygmt.Figure()
# region="IS" specifies Iceland using the ISO country code
fig.coast(shorelines="1/0.5p", region="IS", projection="M25c")
fig.basemap(frame=["a", "+tIceland"])
fig.show()
```

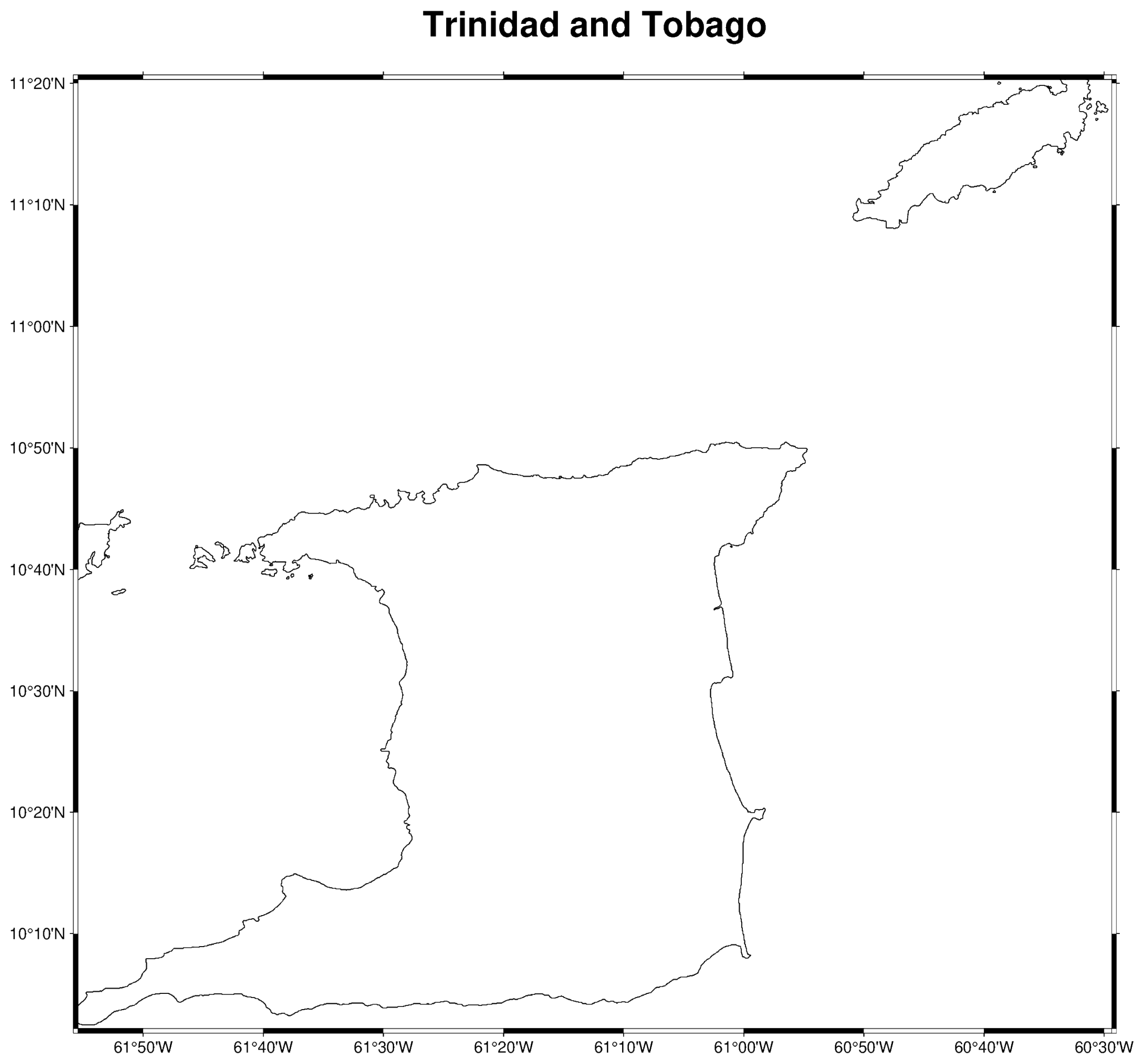


Out:

```
<IPython.core.display.Image object>
```

To use a title with multiple words, the title must be placed inside another set of quotation marks. To prevent the quotation marks from appearing in the figure title, the `frame` parameter can be passed in single quotation marks and the title can be passed in double quotation marks.

```
fig = pygmt.Figure()
# region="TT" specifies Trinidad and Tobago
fig.coast(shorelines="1/0.5p", region="TT", projection="M25c")
fig.basemap(frame=["a", '+t"Trinidad and Tobago"'])
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

9.6.4 Axis labels

Axis labels can be set by passing **x+l***label* (or starting with **y** if labeling the y-axis) to the `frame` parameter of *`pygmt.Figure.basemap`*. By default, all 4 map boundaries (or plot axes) are plotted with both tick marks and axis labels. The axes are named as **W** (west/left), **S** (south/bottom), **N** (north/top), and **E** (east/right) sides of a figure. If an upper-case axis name is passed, the axis is plotted with tick marks and axis labels. A lower case axis name plots only the axis and tick marks.

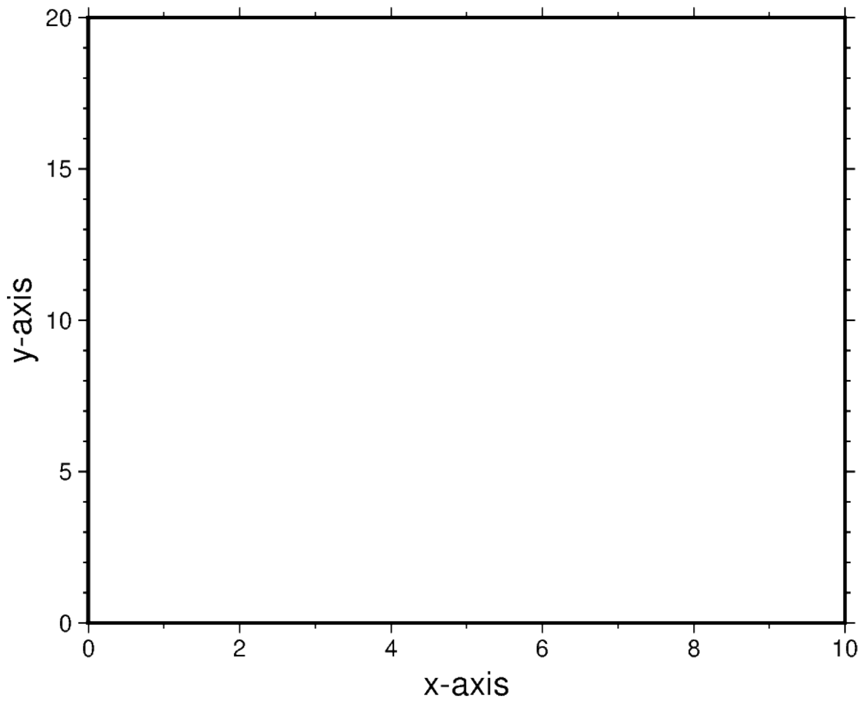
The example below uses a Cartesian projection, as GMT does not allow axis labels to be set for geographic maps.

```
fig = pygmt.Figure()
fig.basemap(
```

(continues on next page)

(continued from previous page)

```
    region=[0, 10, 0, 20],
    projection="X10c/8c",
    # Plot axis, tick marks, and axis labels on the west/left and south/bottom
    # axes
    # Plot axis and tick marks on the north/top and east/right axes
    frame=["WSne", "xaf+lx-axis", "yaf+ly-axis"],
)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

Total running time of the script: (0 minutes 21.894 seconds)

9.7 Projections

PyGMT support many map projections. Use the `projection` parameter to specify which one you want to use in all plotting modules. The projection is specified by a one letter code along with (sometimes optional) reference longitude and latitude and the width of the map (for example, **A***lon0/lat0*[*/horizon*]*/width*). The map height is determined based on the region and projection.

These are all the available projections:

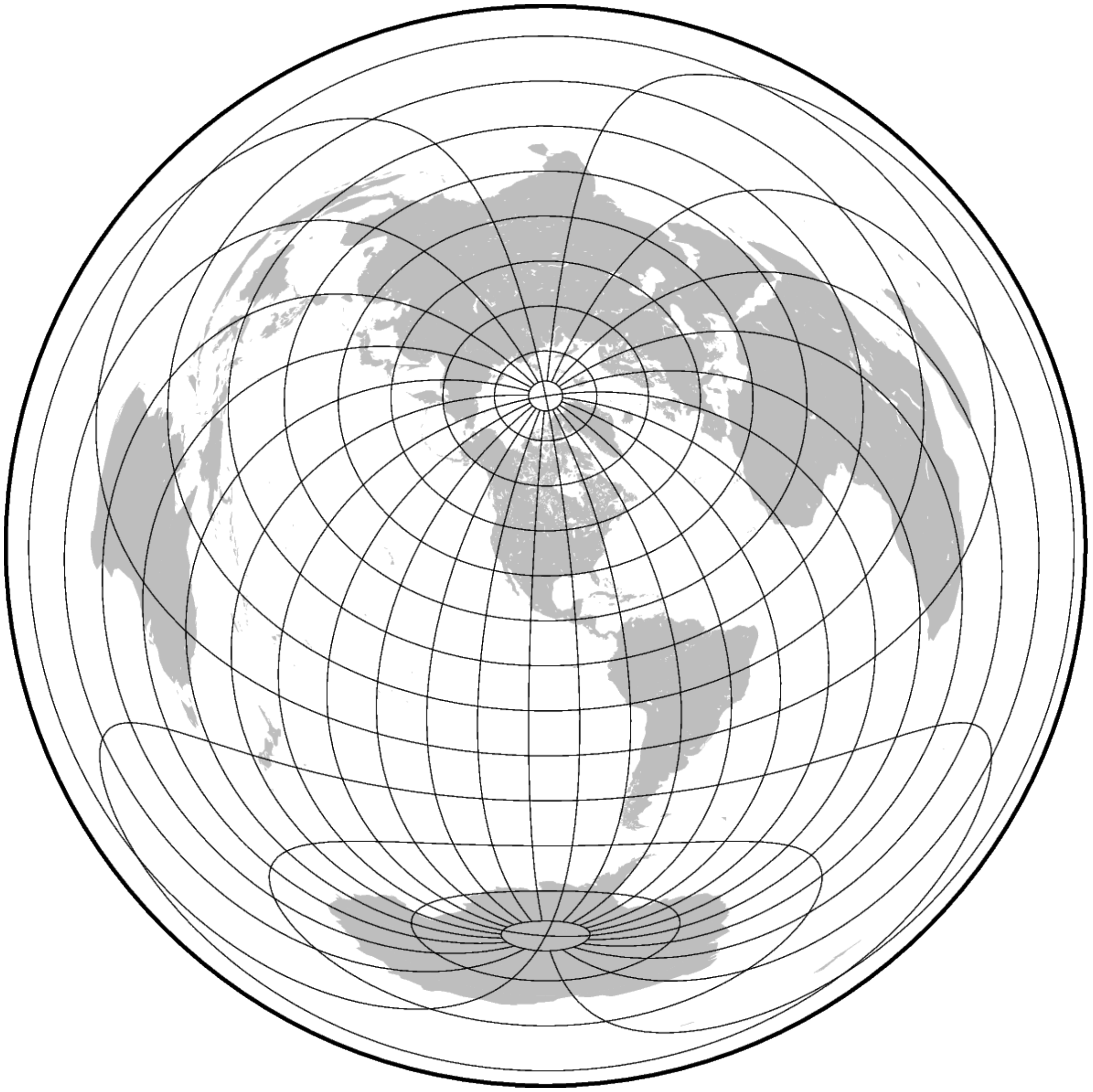
9.7.1 Azimuthal Projections

Azimuthal Equidistant

The main advantage of this projection is that distances from the projection center are displayed in correct proportions. Also directions measured from the projection center are correct. It is very useful for a global view on locations that lie within a certain distance or for comparing distances of different locations relative to the projection center.

e*lon0/lat0*[*/horizon*]*/scale* or **E***lon0/lat0*[*/horizon*]*/width*

The projection type is set with **e** or **E**. *lon0/lat0* specifies the projection center, and the optional parameter *horizon* specifies the maximum distance to the projection center (i.e. the visible portion of the rest of the world map) in degrees <= 180° (default 180°). The size of the figure is set by *scale* or *width*.



Out:

```
coast [WARNING]: Fill/clip continent option (-G) may not work for this projection.
coast [WARNING]: If the antipode (260/90) is in the ocean then chances are good it will␣
↪work.
```

(continues on next page)

(continued from previous page)

```
coast [WARNING]: Otherwise, avoid projection center coordinates that are exact multiples␣
↪of 80 degrees.
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()
fig.coast(projection="E-100/40/15c", region="g", frame="g", land="gray")
fig.show()
```

Total running time of the script: (0 minutes 3.017 seconds)

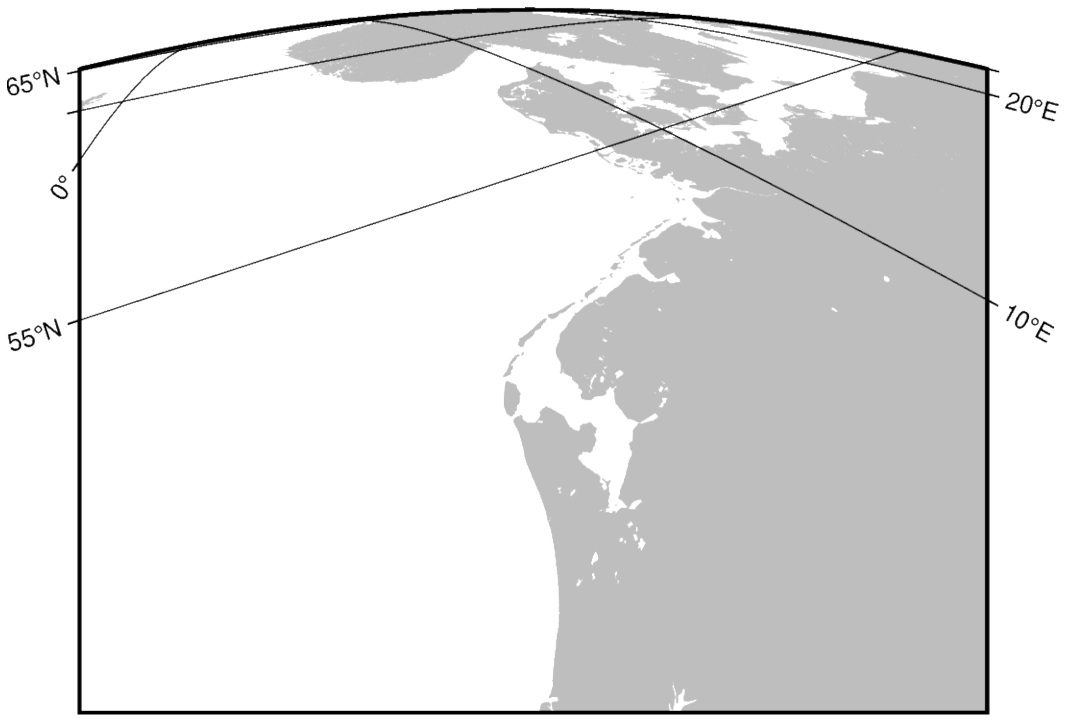
General Perspective

The general perspective projection imitates the view of the Earth from a finite point in space. In a full view of the earth one third of its surface area can be seen.

g*lon0/lat0/altitude/azimuth/tilt/twist/Width/Height/scale* or **G***lon0/lat0/altitude/azimuth/tilt/twist/Width/Height/width*

The projection type is set with **g** or **G**. *lon0/lat0* specifies the projection center and *altitude* sets the height in km of the viewpoint above local sea level (If altitude is less than 10, then it is the distance from the center of the earth to the viewpoint in earth radii). With *azimuth* the direction (in degrees) in which you are looking is specified, measured clockwise from north. *tilt* is given in degrees and is the viewing angle relative to zenith. A tilt of 0° is looking straight down, 60° is looking 30° above horizon. *twist* is the clockwise rotation of the image (in degrees). *Width* and *Height* describe the viewport angle in degrees, and *scale* or *width* determine the size of the figure.

The example shows the coast of Northern Europe viewed from 250 km above sea level looking 30° from north at a tilt of 45°. The height and width of the viewing angle is both 60°, which imitates viewing with naked eye.



Out:

```
coast [WARNING]: 1 annotations along the right border were skipped due to crowding.
coast [WARNING]: 1 annotations along the left border were skipped due to crowding.
coast [WARNING]: Crowding decisions is controlled by MAP_ANNOT_MIN_SPACING, currently␣
↪set to 25.9467p.
coast [WARNING]: Decrease or increase MAP_ANNOT_MIN_SPACING to see more or fewer␣
↪annotations, with 0 showing all annotations.
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()
fig.coast(
    projection="G4/52/250/30/45/0/60/60/12c",
    region="g",
    frame=["x10g10", "y5g5"],
    land="gray",
)
fig.show()
```

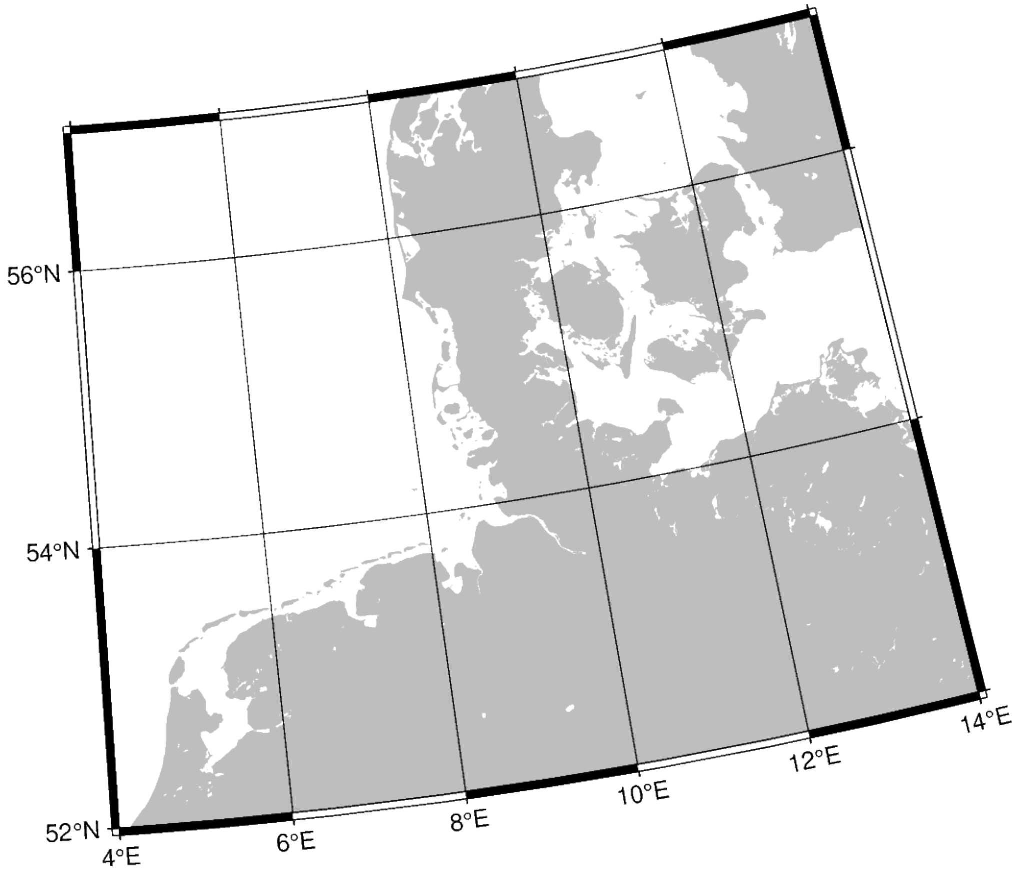
Total running time of the script: (0 minutes 8.281 seconds)

General Stereographic

This map projection is a conformal, azimuthal projection. It is mainly used with a projection center in one of the poles. Then meridians appear as straight lines and cross latitudes at a right angle. Unlike the azimuthal equidistant projection, the distances in this projection are not displayed in correct proportions. It is often used as a hemisphere map like the Lambert Azimuthal Equal Area projection.

s*lon0/lat0*[*/horizon*]*/scale* or **S***lon0/lat0*[*/horizon*]*/width*

The projection type is set with **s** or **S**. *lon0/lat0* specifies the projection center, the optional *horizon* parameter specifies the maximum distance from projection center (in degrees, < 180, default 90), and the *scale* or *width* sets the size of the figure.



Out:

```
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()
fig.coast(region=[4, 14, 52, 57], projection="S0/90/12c", frame="ag", land="gray")
fig.show()
```

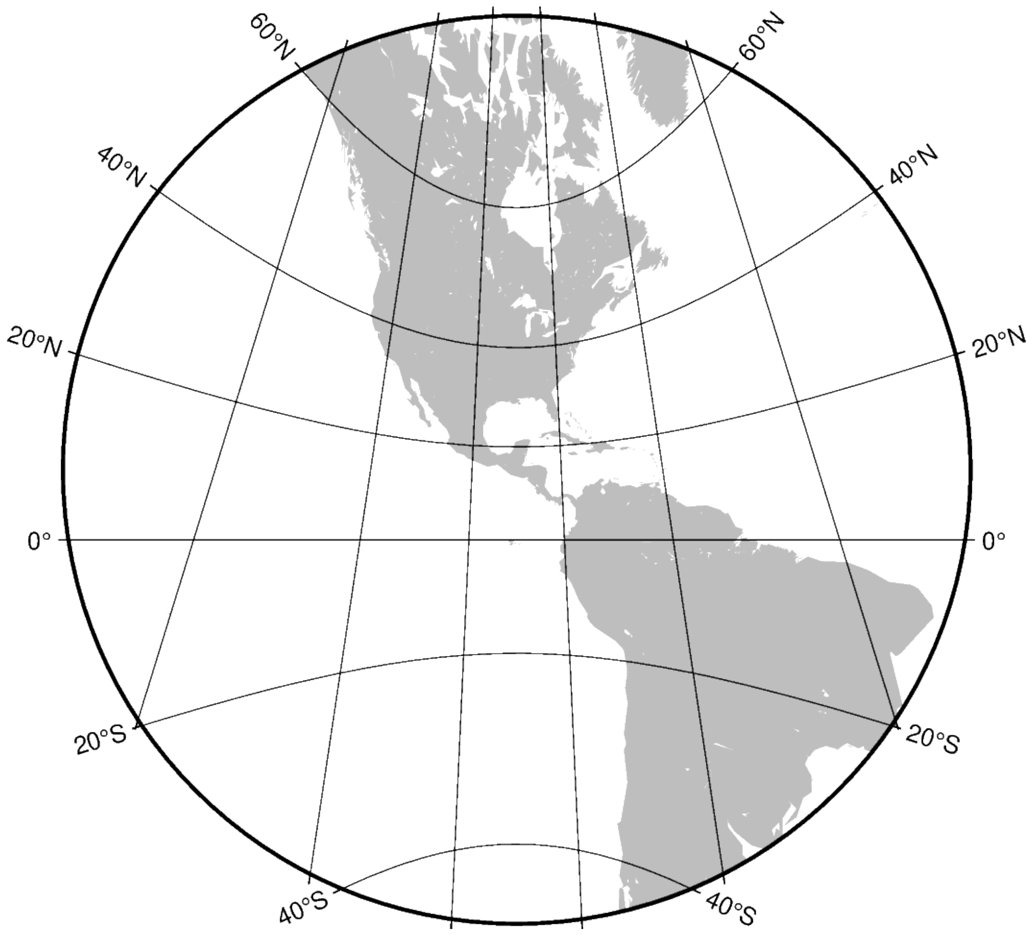
Total running time of the script: (0 minutes 2.159 seconds)

Gnomonic

The point of perspective of the gnomonic projection lies at the center of the earth. As a consequence great circles (orthodromes) on the surface of the Earth are displayed as straight lines, which makes it suitable for distance estimation for navigational purposes. It is neither conformal nor equal-area and the distortion increases greatly with distance to the projection center. It follows that the scope of application is restricted to a small area around the projection center (at a maximum of 60°).

f*lon0/lat0*[*/horizon*]*/scale* or **F***lon0/lat0*[*/horizon*]*/width*

f or **F** specifies the projection type, *lon0/lat0* specifies the projection center, the optional parameter *horizon* specifies the maximum distance from projection center (in degrees, < 90, default 60), and *scale* or *width* sets the size of the figure.



Out:

```
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()
fig.coast(projection="F-90/15/12c", region="g", frame="20g20", land="gray")
fig.show()
```

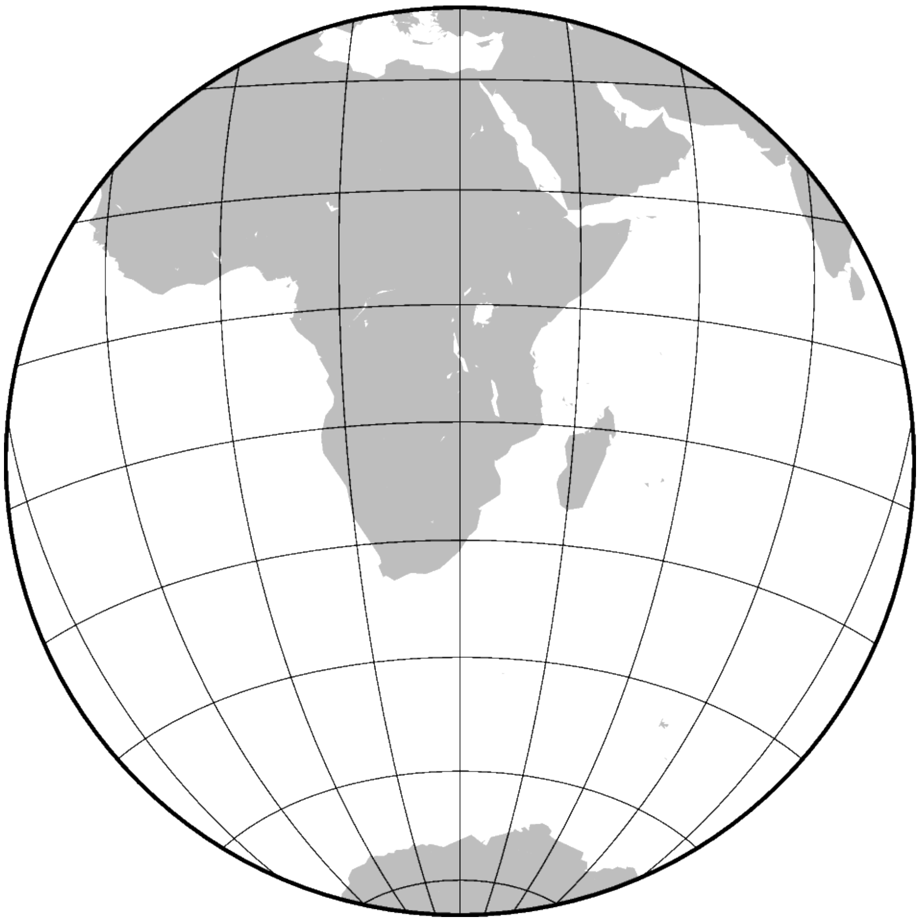
Total running time of the script: (0 minutes 2.352 seconds)

Lambert Azimuthal Equal Area

This projection was developed by Johann Heinrich Lambert in 1772 and is typically used for mapping large regions like continents and hemispheres. It is an azimuthal, equal-area projection, but is not perspective. Distortion is zero at the center of the projection, and increases radially away from this point.

a*lon0/lat0*[*/horizon*]*/scale* or **A***lon0/lat0*[*/horizon*]*/width*

a or **A** specifies the projection type, and *lon0/lat0* specifies the projection center, *horizon* specifies the maximum distance from projection center (in degrees, <= 180, default 90), and *scale* or *width* sets the size of the figure.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()
fig.coast(region="g", frame="afg", land="gray", projection="A30/-20/60/12c")
fig.show()
```

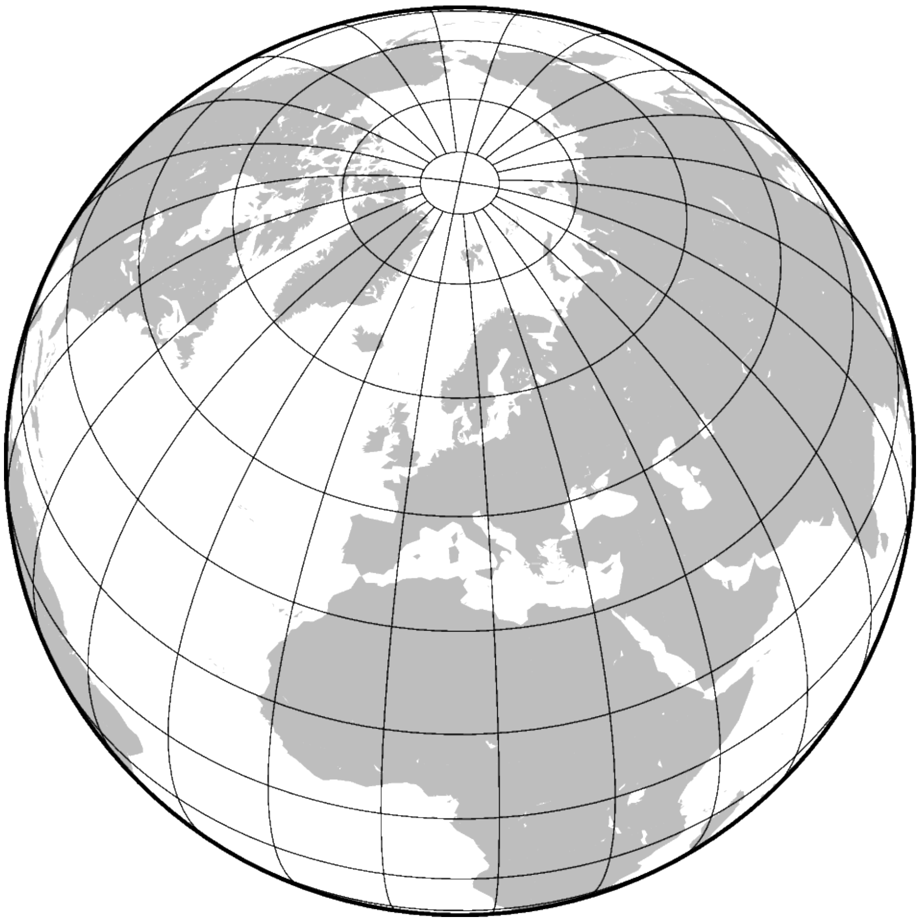
Total running time of the script: (0 minutes 2.224 seconds)

Orthographic

This is a perspective projection like the general perspective, but with the difference that the point of perspective lies in infinite distance. It is therefore often used to give the appearance of a globe viewed from outer space, were one hemisphere can be seen as a whole. It is neither conformal nor equal-area and the distortion increases near the edges.

g*lon0/lat0*[*/horizon*]*/scale* or **G***lon0/lat0*[*/horizon*]*/width*

g or **G** specifies the projection type, *lon0/lat0* specifies the projection center, the optional parameter *horizon* specifies the maximum distance from projection center (in degrees, <= 90, default 90), and *scale* and *width* set the figure size.



Out:

```
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()
fig.coast(projection="G10/52/12c", region="g", frame="g", land="gray")
fig.show()
```

Total running time of the script: (0 minutes 2.428 seconds)

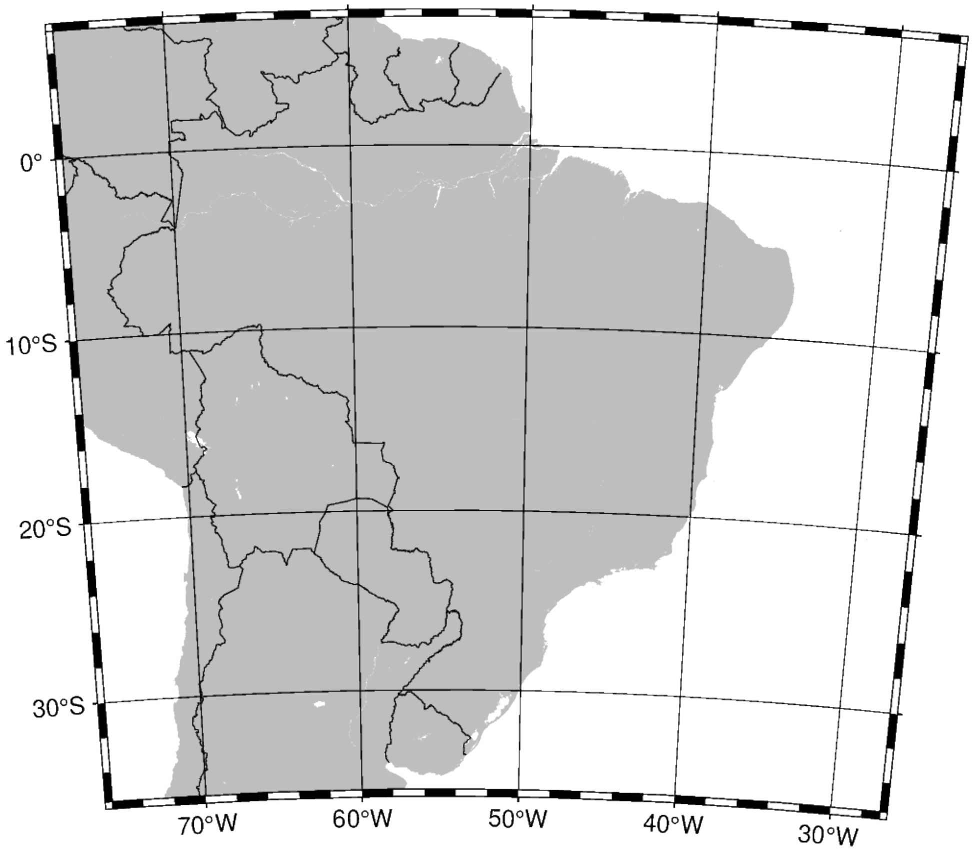
9.7.2 Conic Projections

Albers Conic Equal Area

This projection, developed by Heinrich C. Albers in 1805, is predominantly used to map regions of large east-west extent, in particular the United States. It is a conic, equal-area projection, in which parallels are unequally spaced arcs of concentric circles, more closely spaced at the north and south edges of the map. Meridians, on the other hand, are equally spaced radii about a common center, and cut the parallels at right angles. Distortion in scale and shape vanishes along the two standard parallels. Between them, the scale along parallels is too small; beyond them it is too large. The opposite is true for the scale along meridians.

b*lon0/lat0/lat1/lat2/scale* or **B***lon0/lat0/lat1/lat2/width*

The projection is set with **b** or **B**. The projection center is set by *lon0/lat0* and two standard parallels for the map are set with *lat1/lat2*. The figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()
# Use the ISO country code for Brazil and add a padding of 2 degrees (+R2)
fig.coast(
    projection="B-55/-15/-25/0/12c", region="BR+R2", frame="afg", land="gray", borders=1
)
fig.show()
```

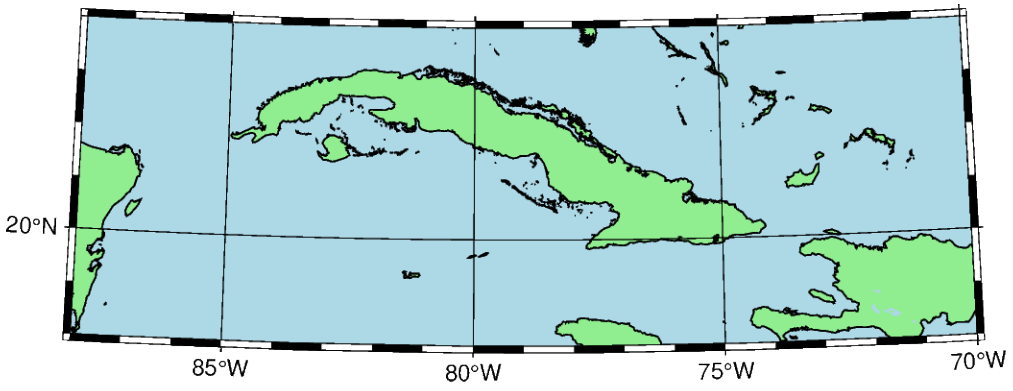
Total running time of the script: (0 minutes 3.008 seconds)

Equidistant conic

The equidistant conic projection was described by the Greek philosopher Claudius Ptolemy about A.D. 150. It is neither conformal or equal-area, but serves as a compromise between them. The scale is true along all meridians and the standard parallels.

d*lon0/lat0/lat1/lat2/scale* or **D***lon0/lat0/lat1/lat2/width*

The projection is set with **d** or **D**. The projection center is set by *lon0/lat0* and two standard parallels for the map are set with *lat1/lat2*. The figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()
```

(continues on next page)

(continued from previous page)

```
fig.coast(
    shorelines="1/0.5p",
    region=[-88, -70, 18, 24],
    projection="D-79/21/19/23/12c",
    land="lightgreen",
    water="lightblue",
    frame="afg",
)
fig.show()
```

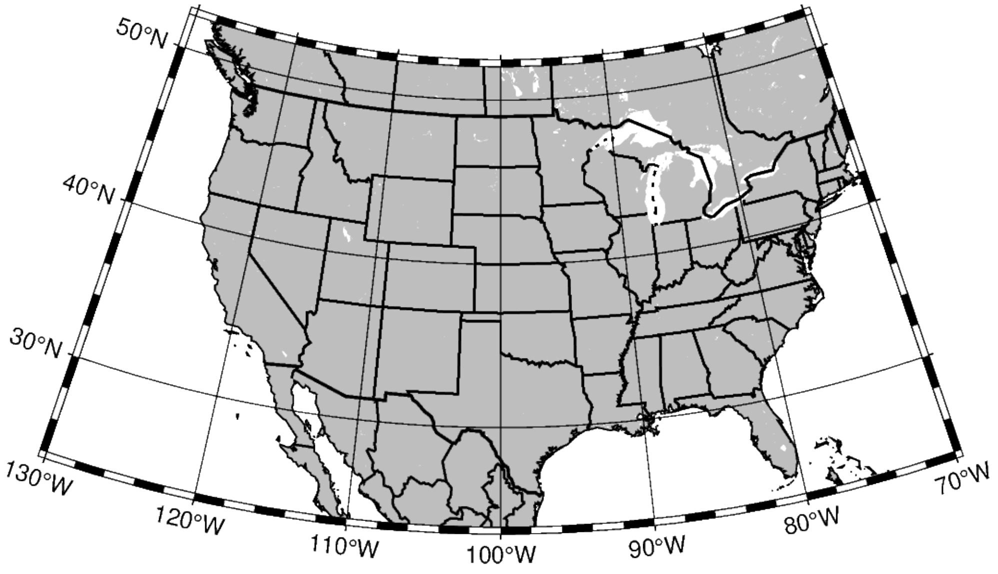
Total running time of the script: (0 minutes 1.967 seconds)

Lambert Conic Conformal Projection

This conic projection was designed by the Alsatian mathematician Johann Heinrich Lambert (1772) and has been used extensively for mapping of regions with predominantly east-west orientation, just like the Albers projection. Unlike the Albers projection, Lambert's conformal projection is not equal-area. The parallels are arcs of circles with a common origin, and meridians are the equally spaced radii of these circles. As with Albers projection, it is only the two standard parallels that are distortion-free.

l*lon0/lat0/lat1/lat2/scale* or **L***lon0/lat0/lat1/lat2/width*

The projection is set with **l** or **L**. The projection center is set by *lon0/lat0* and two standard parallels for the map are set with *lat1/lat2*. The figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()
fig.coast(
    shorelines="1/0.5p",
    region=[-130, -70, 24, 52],
    projection="L-100/35/33/45/12c",
    land="gray",
    borders=["1/thick,black", "2/thin,black"],
    frame="afg",
)

fig.show()
```

Total running time of the script: (0 minutes 2.345 seconds)

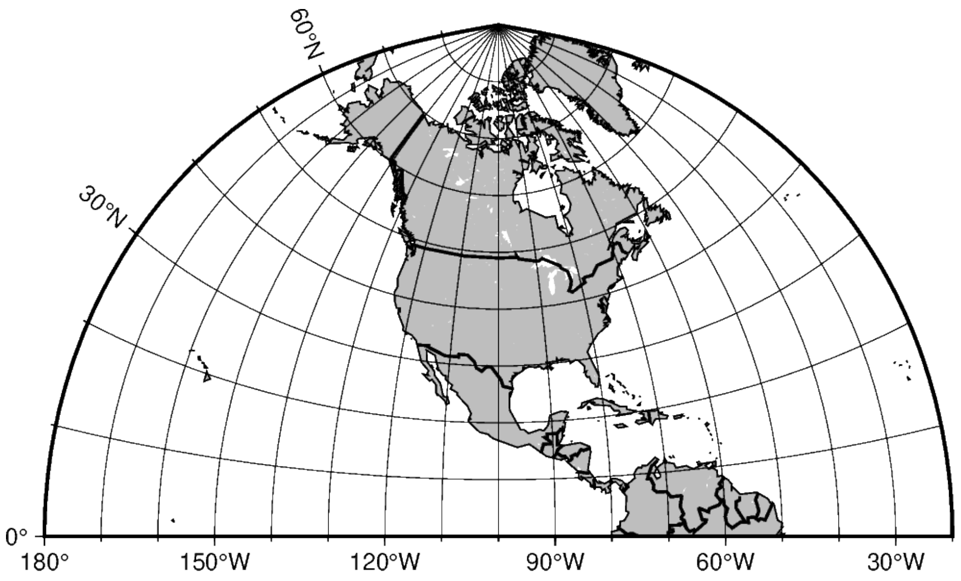
Polyconic Projection

The polyconic projection, in Europe usually referred to as the American polyconic projection, was introduced shortly before 1820 by the Swiss-American cartographer Ferdinand Rodulph Hassler (1770–1843). As head of the Survey of the Coast, he was looking for a projection that would give the least distortion for mapping the coast of the United States. The projection acquired its name from the construction of each parallel, which is achieved by projecting the parallel onto the cone while it is rolled around the globe, along the central meridian, tangent to that parallel. As a consequence, the projection involves many cones rather than a single one used in regular conic projections.

The polyconic projection is neither equal-area, nor conformal. It is true to scale without distortion along the central meridian. Each parallel is true to scale as well, but the meridians are not as they get further away from the central meridian. As a consequence, no parallel is standard because conformity is lost with the lengthening of the meridians.

poly/*scale* or **Poly**/*width*

The projection is set with **poly** or **Poly**. The figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()
fig.coast(
    shorelines="1/0.5p",
    region=[-180, -20, 0, 90],
    projection="Poly/12c",
    land="gray",
    borders="1/thick,black",
    frame="afg10",
)

fig.show()
```

Total running time of the script: (0 minutes 2.161 seconds)

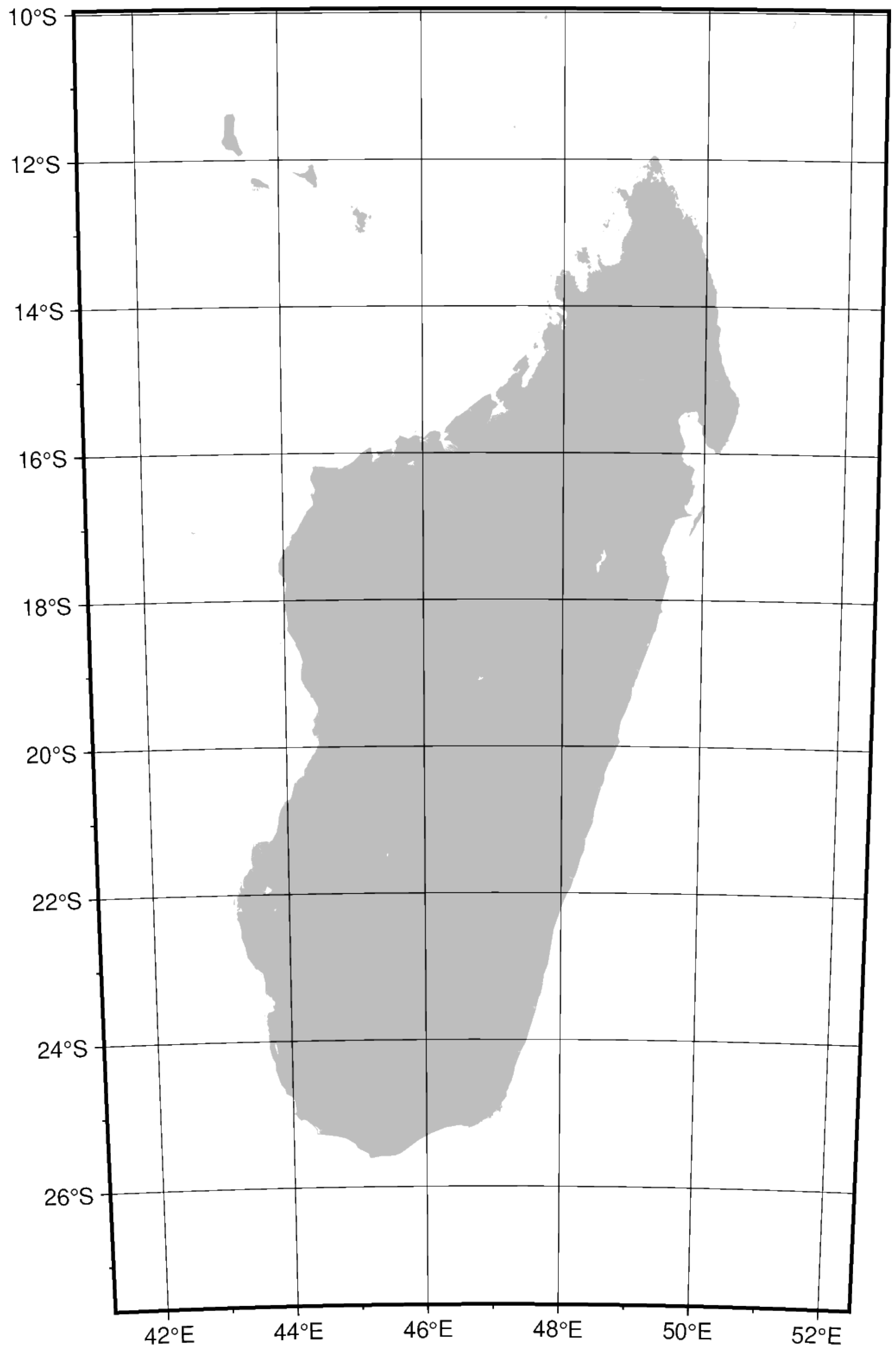
9.7.3 Cylindric Projections

Cassini Cylindrical

This cylindrical projection was developed in 1745 by César-François Cassini de Thury for the survey of France. It is occasionally called Cassini-Soldner since the latter provided the more accurate mathematical analysis that led to the development of the ellipsoidal formulae. The projection is neither conformal nor equal-area, and behaves as a compromise between the two end-members. The distortion is zero along the central meridian. It is best suited for mapping regions of north-south extent. The central meridian, each meridian 90° away, and equator are straight lines; all other meridians and parallels are complex curves.

c*lon0/lat0/scale* or **C***lon0/lat0/width*

The projection is set with **c** or **C**. The projection center is set by *lon0/lat0*, and the figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()
# Use the ISO code for Madagascar (MG) and pad it by 2 degrees (+R2)
fig.coast(projection="C47/-19/12c", region="MG+R2", frame="afg", land="gray", borders=1)
fig.show()
```

Total running time of the script: (0 minutes 3.050 seconds)

Cylindrical Stereographic

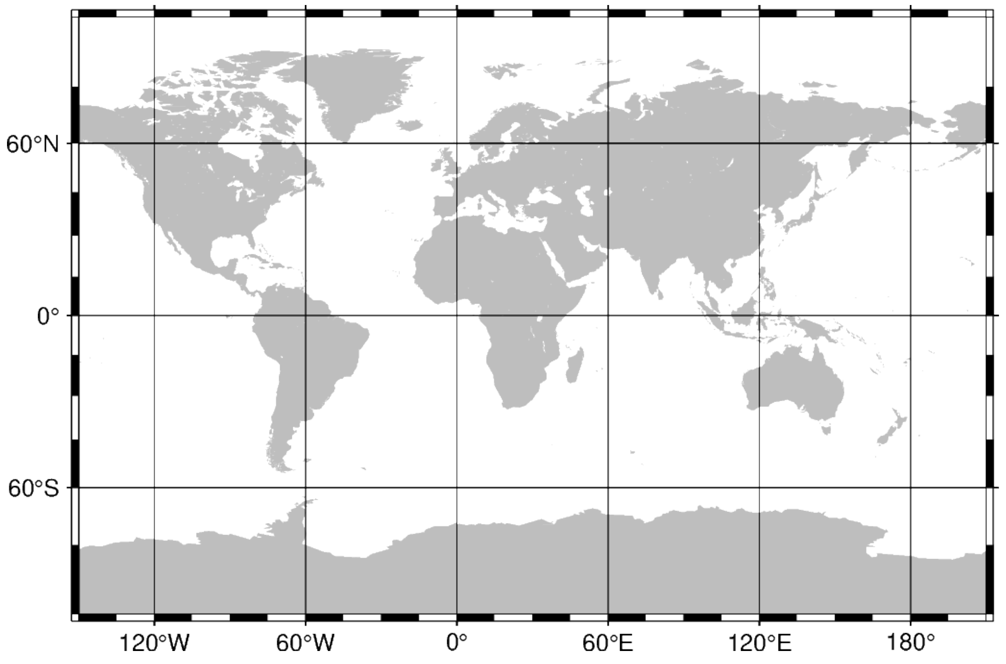
The cylindrical stereographic projections are certainly not as notable as other cylindrical projections, but are still used because of their relative simplicity and their ability to overcome some of the downsides of other cylindrical projections, like extreme distortions of the higher latitudes. The stereographic projections are perspective projections, projecting the sphere onto a cylinder in the direction of the antipodal point on the equator. The cylinder crosses the sphere at two standard parallels, equidistant from the equator.

cyl_stere/[*lon0/*][*lat0/*]*scale* or **Cyl_stere/**[*lon0/*][*lat0/*]*width*

The projection is set with **cyl_stere** or **Cyl_stere**. The central meridian is set by the optional *lon0*, and the figure size is set with *scale* or *width*.

The standard parallel is typically one of these (but can be any value):

- 66.159467 - Miller's modified Gall
- 55 - Kamenetskiy's First
- 45 - Gall's Stereographic
- 30 - Bolshoi Sovietskii Atlas Mira or Kamenetskiy's Second
- 0 - Braun's Cylindrical



Out:

```
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()
fig.coast(region="g", frame="afg", land="gray", projection="Cyl_stere/30/-20/12c")
fig.show()
```

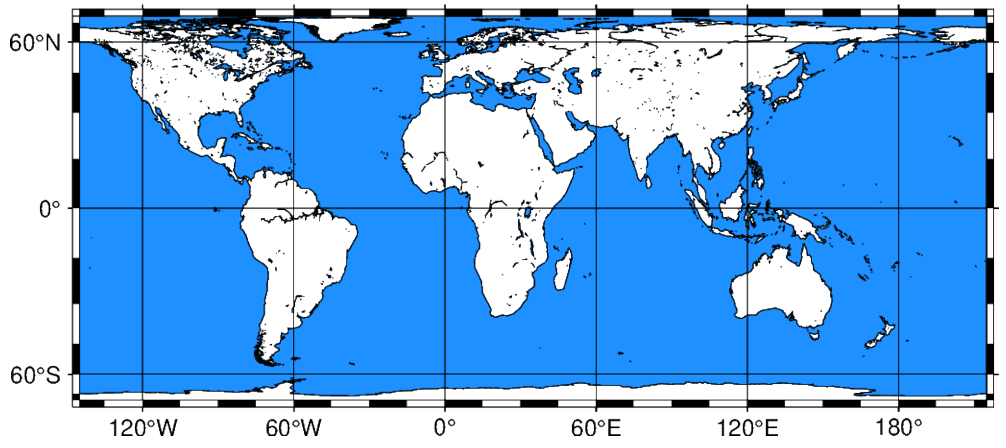
Total running time of the script: (0 minutes 2.178 seconds)

Cylindrical equal-area

This cylindrical projection is actually several projections, depending on what latitude is selected as the standard parallel. However, they are all equal area and hence non-conformal. All meridians and parallels are straight lines.

y*lon0/lat0/scale* or **Y***lon0/lat0/width*

The projection is set with **y** or **Y**. The projection center is set by *lon0/lat0*, and the figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()
# Use region "d" to specify global region (-180/180/-90/90)
fig.coast(
    region="d",
    projection="Y35/30/12c",
```

(continues on next page)

(continued from previous page)

```
    water="dodgerblue",
    shorelines="thinnest",
    frame="afg",
)
fig.show()
```

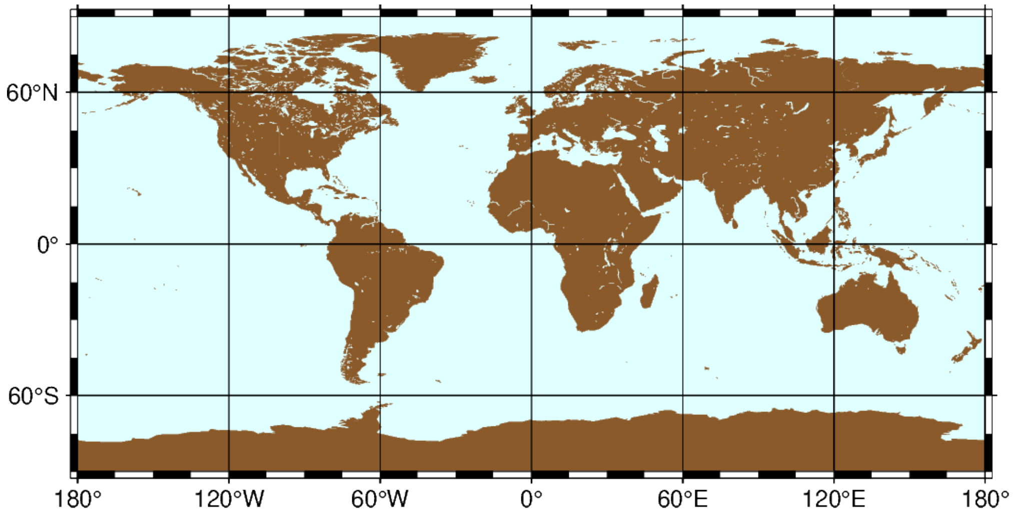
Total running time of the script: (0 minutes 2.206 seconds)

Cylindrical equidistant

This simple cylindrical projection is really a linear scaling of longitudes and latitudes. The most common form is the Plate Carrée projection, where the scaling of longitudes and latitudes is the same. All meridians and parallels are straight lines.

q*scale* or **Q***width*

The projection is set with **q** or **Q**, and the figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()
# Use region "d" to specify global region (-180/180/-90/90)
fig.coast(
    region="d",
    projection="Q12c",
    land="tan4",
    water="lightcyan",
    frame="afg",
```

(continues on next page)

(continued from previous page)

```
)
fig.show()
```

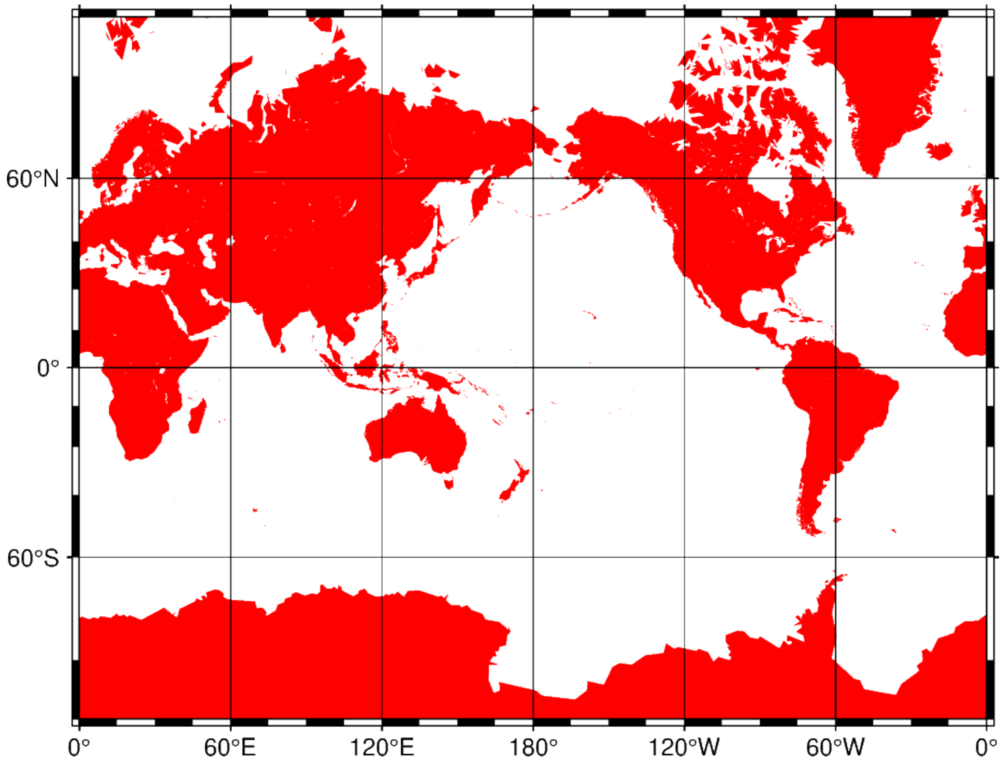
Total running time of the script: (0 minutes 2.158 seconds)

Mercator

The Mercator projection takes its name from the Flemish cartographer Gheert Cremer, better known as Gerardus Mercator, who presented it in 1569. The projection is a cylindrical and conformal, with no distortion along the equator. A major navigational feature of the projection is that a line of constant azimuth is straight. Such a line is called a rhumb line or loxodrome. Thus, to sail from one point to another one only had to connect the points with a straight line, determine the azimuth of the line, and keep this constant course for the entire voyage. The Mercator projection has been used extensively for world maps in which the distortion towards the polar regions grows rather large.

m[*lon0[/lat0]*]*/scale* or **M**[*lon0*][*/lat0*]*/width*

The projection is set with **m** or **M**. The central meridian is set with the optional *lon0* and the standard parallel is set with the optional *lat0*. The figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()
fig.coast(region=[0, 360, -80, 80], frame="afg", land="red", projection="M0/0/12c")
fig.show()
```

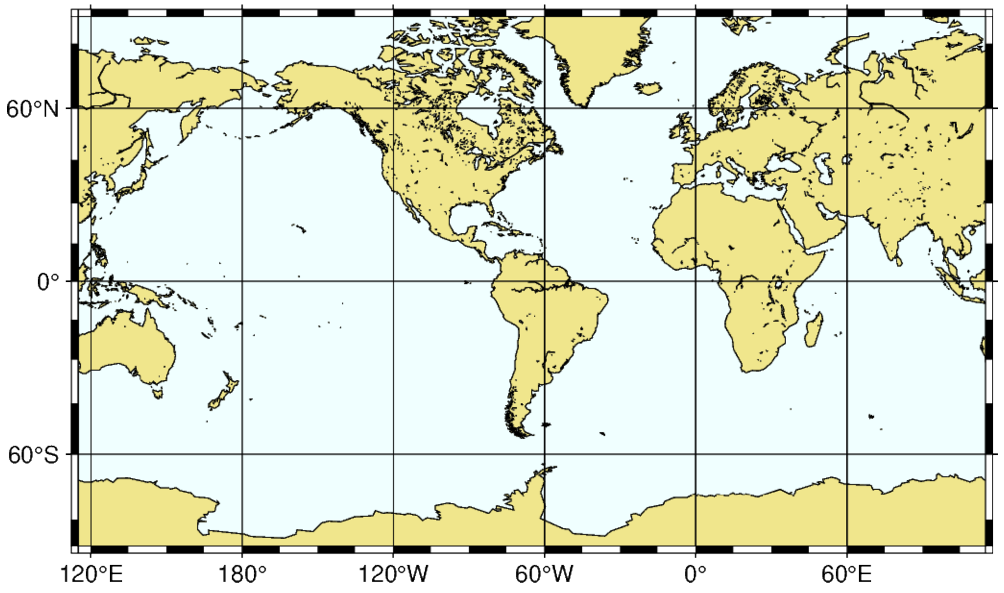
Total running time of the script: (0 minutes 2.222 seconds)

Miller cylindrical

This cylindrical projection, presented by Osborn Maitland Miller of the American Geographic Society in 1942, is neither equal nor conformal. All meridians and parallels are straight lines. The projection was designed to be a compromise between Mercator and other cylindrical projections. Specifically, Miller spaced the parallels by using Mercator's formula with 0.8 times the actual latitude, thus avoiding the singular poles; the result was then divided by 0.8.

j[*lon0/*]*/scale* or **J**[*lon0/*]*/width*

The projection is set with **j** or **J**. The central meridian is set by the optional *lon0*, and the figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()
fig.coast(
    region=[-180, 180, -80, 80],
    projection="J-65/12c",
```

(continues on next page)

(continued from previous page)

```
    land="khaki",
    water="azure",
    shorelines="thinnest",
    frame="afg",
)
fig.show()
```

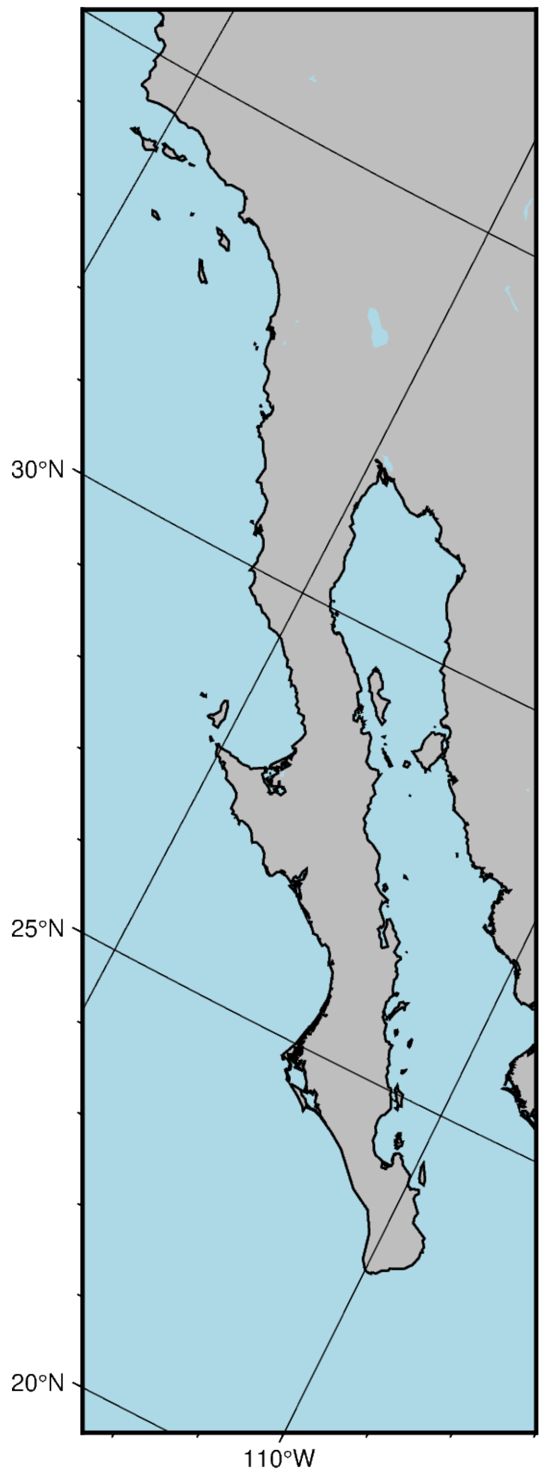
Total running time of the script: (0 minutes 2.512 seconds)

Oblique Mercator

Oblique configurations of the cylinder give rise to the oblique Mercator projection. It is particularly useful when mapping regions of large lateral extent in an oblique direction. Both parallels and meridians are complex curves. The projection was developed in the early 1900s by several workers.

oa|**oA***lon0/lat0/azimuth/scale*[**+v**] or **Oa**|**OA***lon0/lat0/azimuth/width*[**+v**]

The projection is set with **o** or **O**. The pole is set in the northern hemisphere with **a** or the southern hemisphere with **A**. The central meridian is set by *lon0/lat0*. The oblique equator is set by *azimuth*. Align the y-axis with the optional **+v**. The figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()
# Using the origin and azimuth
fig.coast(
    projection="Oa-120/25/-30/6c+v",
    # Set bottom left and top right coordinates of the figure with "+r"
    region="-122/35/-107/22+r",
    frame="afg",
    land="gray",
    shorelines="1/thin",
    water="lightblue",
)
fig.show()
```

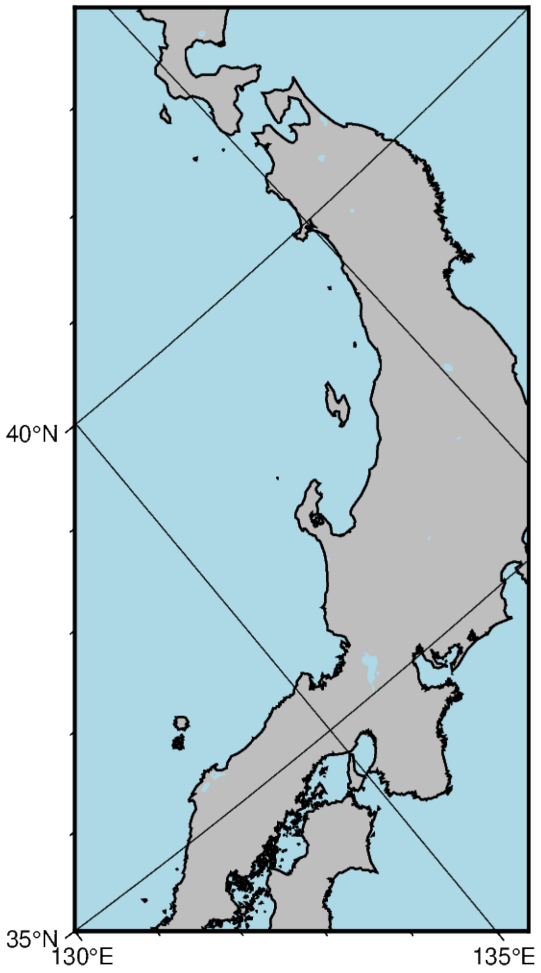
Total running time of the script: (0 minutes 2.127 seconds)

Oblique Mercator

Oblique configurations of the cylinder give rise to the oblique Mercator projection. It is particularly useful when mapping regions of large lateral extent in an oblique direction. Both parallels and meridians are complex curves. The projection was developed in the early 1900s by several workers.

ob|oB*lon0/lat0/lon1/lat1/scale*[**+v**] or **Ob|OB***lon0/lat0/lon1/lat1/width*[**+v**]

The projection is set with **o** or **O**. The pole is set in the northern hemisphere with **b** or the southern hemisphere with **B**. The central meridian is set by *lon0/lat0*. The oblique equator is set by *lon1/lat1*. Align the y-axis with the optional **+v**. The figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()
# Using the origin and two points
fig.coast(
    projection="Ob130/35/25/35/6c",
    # Set bottom left and top right coordinates of the figure with "+r"
    region="130/35/145/40+r",
    frame="afg",
    land="gray",
    shorelines="1/thin",
    water="lightblue",
)
fig.show()
```

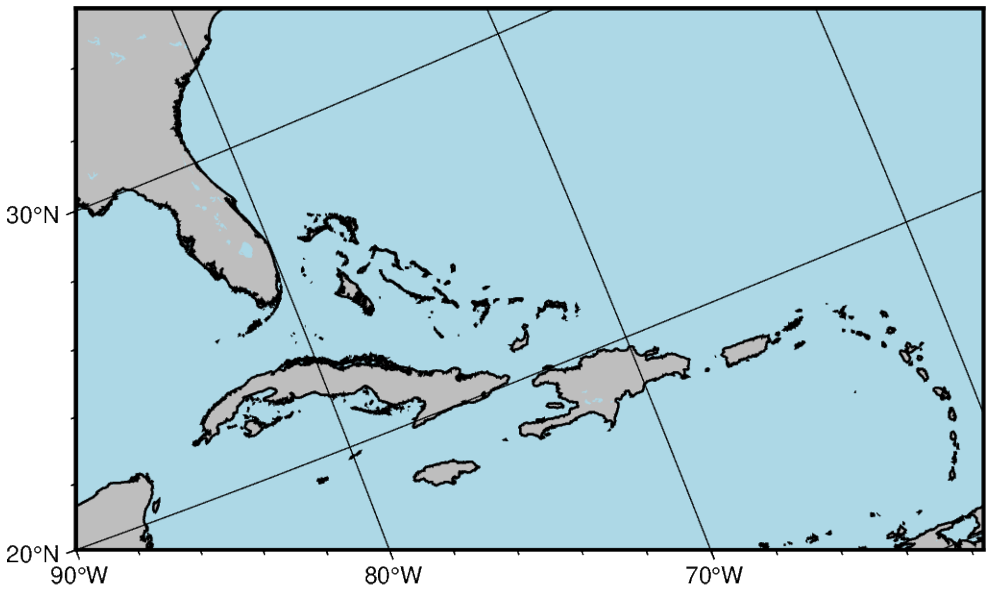
Total running time of the script: (0 minutes 2.029 seconds)

Oblique Mercator

Oblique configurations of the cylinder give rise to the oblique Mercator projection. It is particularly useful when mapping regions of large lateral extent in an oblique direction. Both parallels and meridians are complex curves. The projection was developed in the early 1900s by several workers.

oc|oC*lon0/lat0/lonp/latp/scale*[**+v**] or **Oc|OC***lon0/lat0/lonp/latp/width*[**+v**]

The projection is set with **o** or **O**. The central meridian is set by *lon0/lat0*. The projection pole is set by *lonp/latp* in option three. Align the y-axis with the optional **+v**. The figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()
# Using the origin projection pole
fig.coast(
    projection="Oc280/25.5/22/69/12c",
    # Set bottom left and top right coordinates of the figure with "+r"
    region="270/20/305/25+r",
    frame="afg",
    land="gray",
    shorelines="1/thin",
    water="lightblue",
)
fig.show()
```

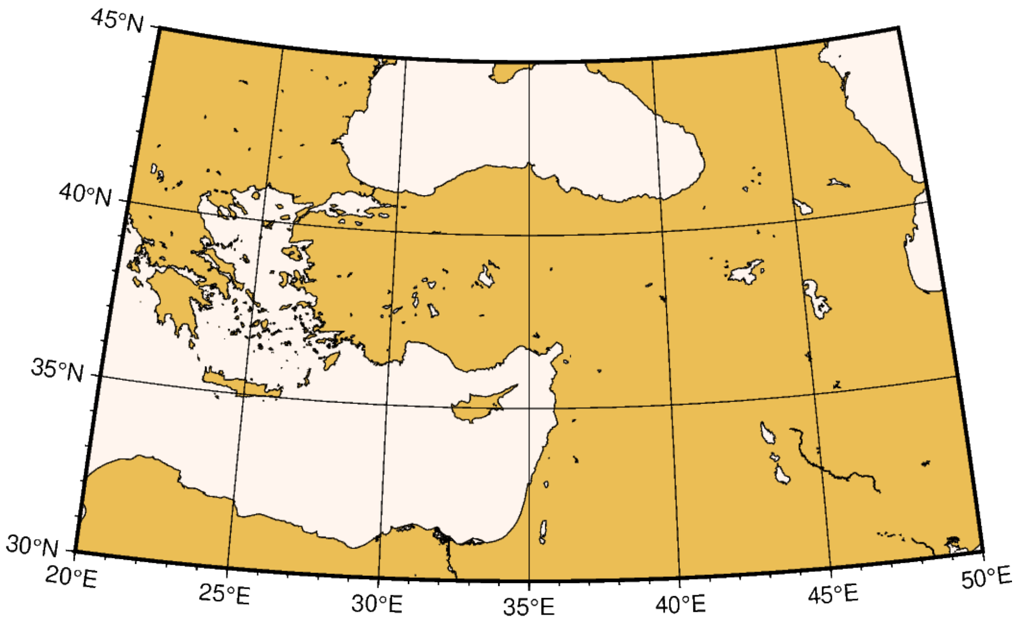
Total running time of the script: (0 minutes 2.361 seconds)

Transverse Mercator

The transverse Mercator was invented by Johann Heinrich Lambert in 1772. In this projection the cylinder touches a meridian along which there is no distortion. The distortion increases away from the central meridian and goes to infinity at 90° from center. The central meridian, each meridian 90° away from the center, and equator are straight lines; other parallels and meridians are complex curves.

t*lon0/*[*lat0/*]*scale* or **T***lon0/*[*lat0/*]*width*

The projection is set with **t** or **T**. The central meridian is set by *lon0*, the latitude of the origin is set by the optional *lat0*, and the figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()
fig.coast(
    region=[20, 50, 30, 45],
    projection="T35/12c",
    land="lightbrown",
    water="seashell",
    shorelines="thinnest",
    frame="afg",
)
fig.show()
```

Total running time of the script: (0 minutes 2.210 seconds)

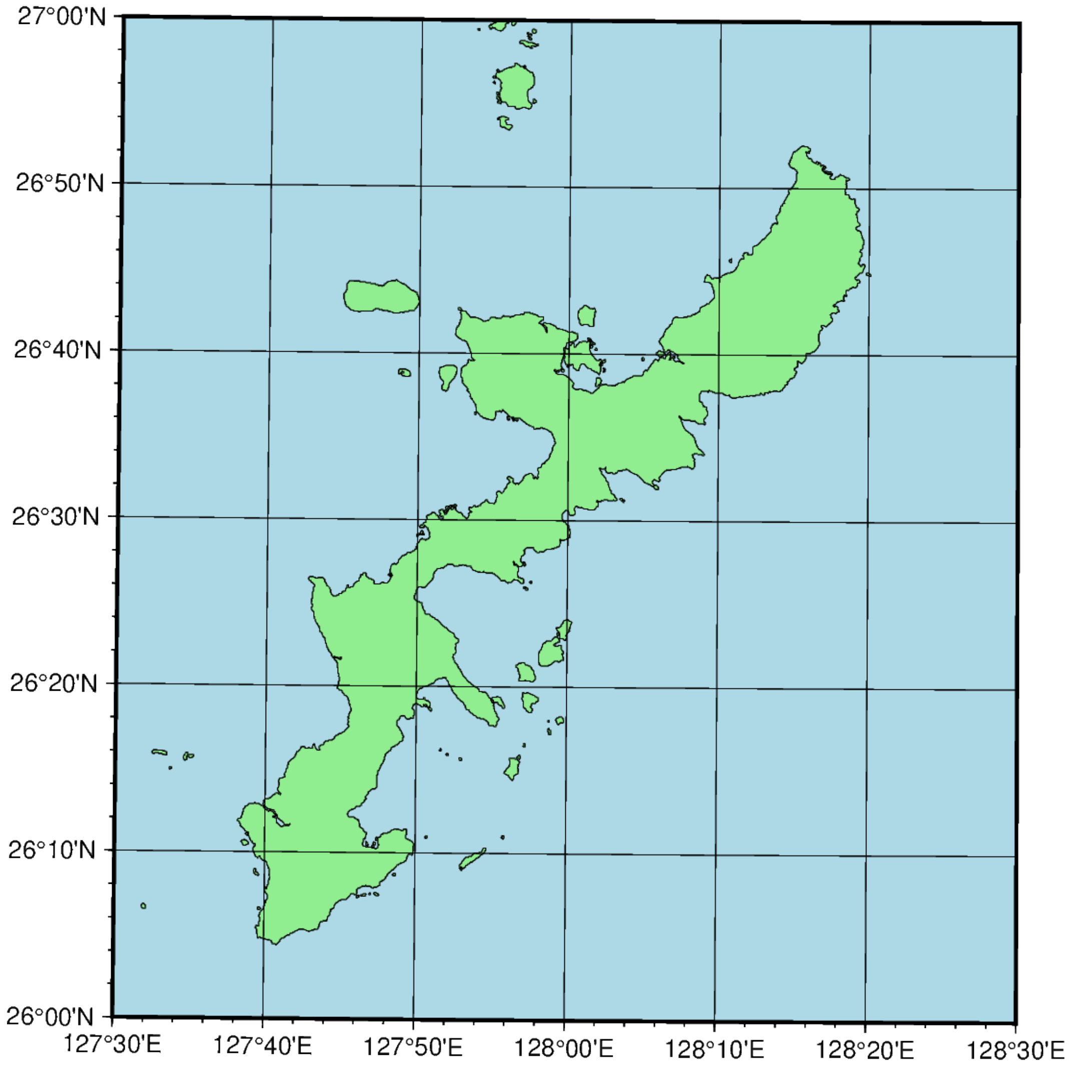
Universal Transverse Mercator

A particular subset of the transverse Mercator is the Universal Transverse Mercator (UTM) which was adopted by the US Army for large-scale military maps. Here, the globe is divided into 60 zones between 84°S and 84°N, most of which are 6 wide. Each of these UTM zones have their unique central meridian. Furthermore, each zone is divided into latitude bands but these are not needed to specify the projection for most cases.

In order to minimize the distortion in any given zone, a scale factor of 0.9996 has been factored into the formulae. This makes the UTM projection a secant projection and not a tangent projection like the transverse Mercator above. The scale only varies by 1 part in 1,000 from true scale at equator. The ellipsoidal projection expressions are accurate for map areas that extend less than 10 away from the central meridian.

u*zone/scale* or **U***zone/width*

the projection is set with **u** or **U**. *zone* sets the zone for the figure, and the figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()
# UTM Zone is set to 52R
fig.coast(
    region=[127.5, 128.5, 26, 27],
    projection="U52R/12c",
    land="lightgreen",
    water="lightblue",
    shorelines="thinnest",
    frame="afg",
)
fig.show()
```

Total running time of the script: (0 minutes 2.372 seconds)

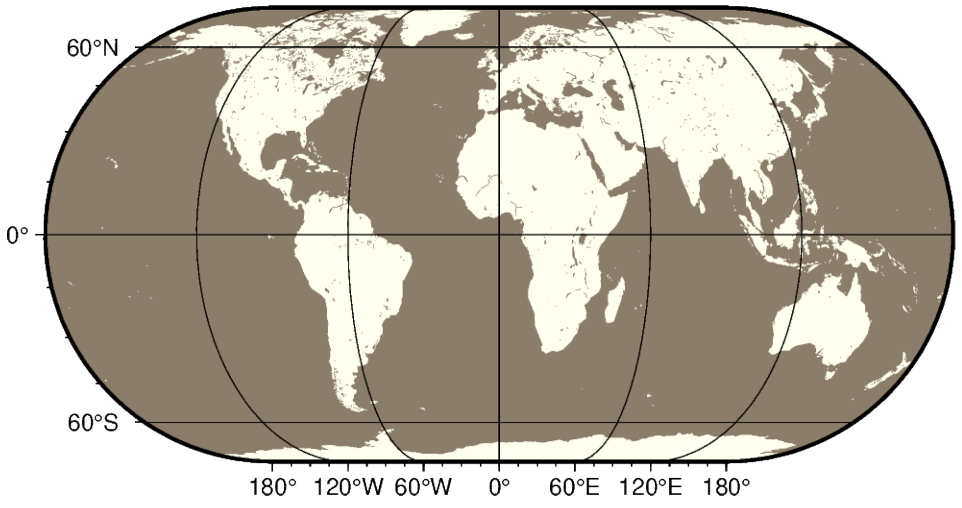
9.7.4 Miscellaneous Projections

Eckert IV

The Eckert IV projection, presented by the German cartographer Max Eckert-Greiffendorff in 1906, is a pseudo-cylindrical equal-area projection. Central meridian and all parallels are straight lines; other meridians are equally spaced elliptical arcs. The scale is true along latitude 40°30'.

kf[*lon0/*]*scale* or **Kf**[*lon0/*]*width*

The projection is set with **kf** or **Kf**. The central meridian is set with the optional *lon0*, and the figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()
# Use region "d" to specify global region (-180/180/-90/90)
fig.coast(region="d", projection="Kf12c", land="ivory", water="bisque4", frame="afg")
fig.show()
```

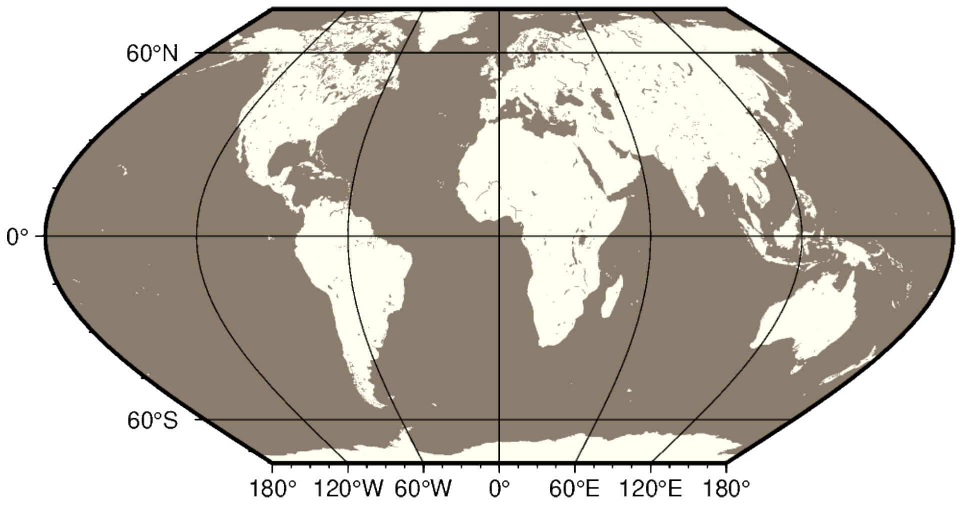
Total running time of the script: (0 minutes 2.195 seconds)

Eckert VI

The Eckert VI projections, presented by the German cartographer Max Eckert-Greiffendorff in 1906, is a pseudo-cylindrical equal-area projection. Central meridian and all parallels are straight lines; other meridians are equally spaced sinusoids. The scale is true along latitude 49°16'.

ks[*lon0/*]*scale* or **Ks**[*lon0/*]*width*

The projection is set with **ks** or **Ks**. The central meridian is set with the optional *lon0*, and the figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()
# Use region "d" to specify global region (-180/180/-90/90)
fig.coast(region="d", projection="Ks12c", land="ivory", water="bisque4", frame="afg")
fig.show()
```

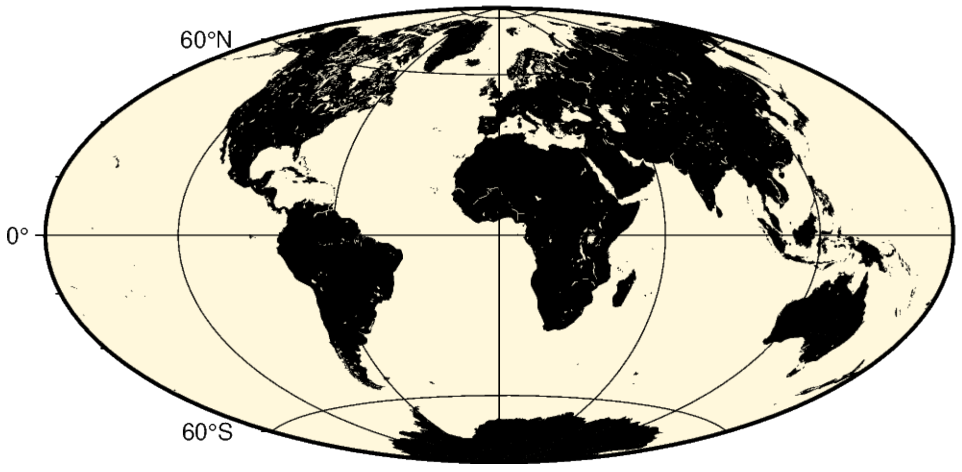
Total running time of the script: (0 minutes 2.163 seconds)

Hammer

The equal-area Hammer projection, first presented by the German mathematician Ernst von Hammer in 1892, is also known as Hammer-Aitoff (the Aitoff projection looks similar, but is not equal-area). The border is an ellipse, equator and central meridian are straight lines, while other parallels and meridians are complex curves.

h[*lon0/*]*scale* or **H**[*lon0/*]*width*

The projection is set with **h** or **H**. The central meridian is set with the optional *lon0*, and the figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()
# Use region "d" to specify global region (-180/180/-90/90)
fig.coast(region="d", projection="H12c", land="black", water="cornsilk", frame="afg")
fig.show()
```

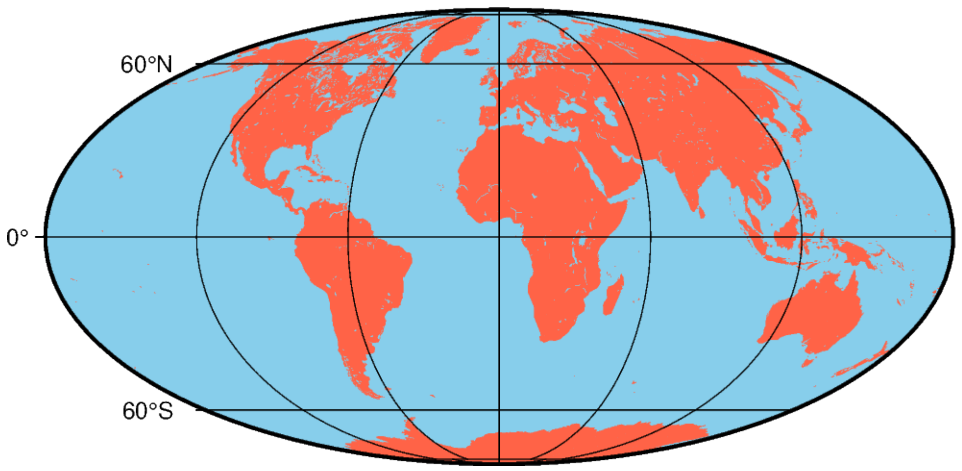
Total running time of the script: (0 minutes 2.150 seconds)

Mollweide

This pseudo-cylindrical, equal-area projection was developed by the German mathematician and astronomer Karl Brandan Mollweide in 1805. Parallels are unequally spaced straight lines with the meridians being equally spaced elliptical arcs. The scale is only true along latitudes 40°44' north and south. The projection is used mainly for global maps showing data distributions. It is occasionally referenced under the name homalographic projection.

w[*lon0/*]*scale* or **W**[*lon0/*]*width*

The projection is set with **w** or **W**. The central meridian is set with the optional *lon0*, and the figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()
# Use region "d" to specify global region (-180/180/-90/90)
fig.coast(region="d", projection="W12c", land="tomato1", water="skyblue", frame="ag")
fig.show()
```

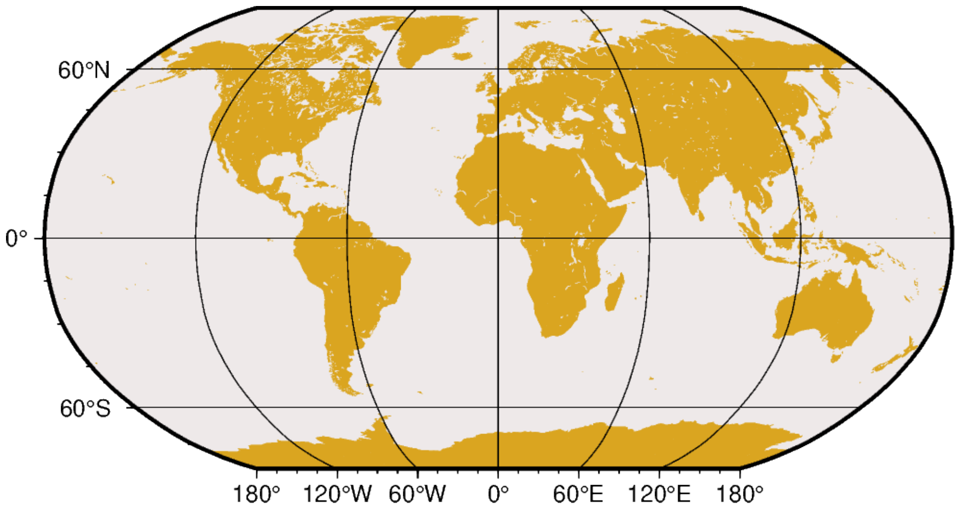
Total running time of the script: (0 minutes 2.185 seconds)

Robinson

The Robinson projection, presented by the American geographer and cartographer Arthur H. Robinson in 1963, is a modified cylindrical projection that is neither conformal nor equal-area. Central meridian and all parallels are straight lines; other meridians are curved. It uses lookup tables rather than analytic expressions to make the world map "look" right 22. The scale is true along latitudes 38. The projection was originally developed for use by Rand McNally and is currently used by the National Geographic Society.

n[*lon0/*]*scale* or **N**[*lon0/*]*width*

The projection is set with **n** or **N**. The central meridian is set with the optional *lon0*, and the figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()
# Use region "d" to specify global region (-180/180/-90/90)
fig.coast(region="d", projection="N12c", land="goldenrod", water="snow2", frame="afg")
fig.show()
```

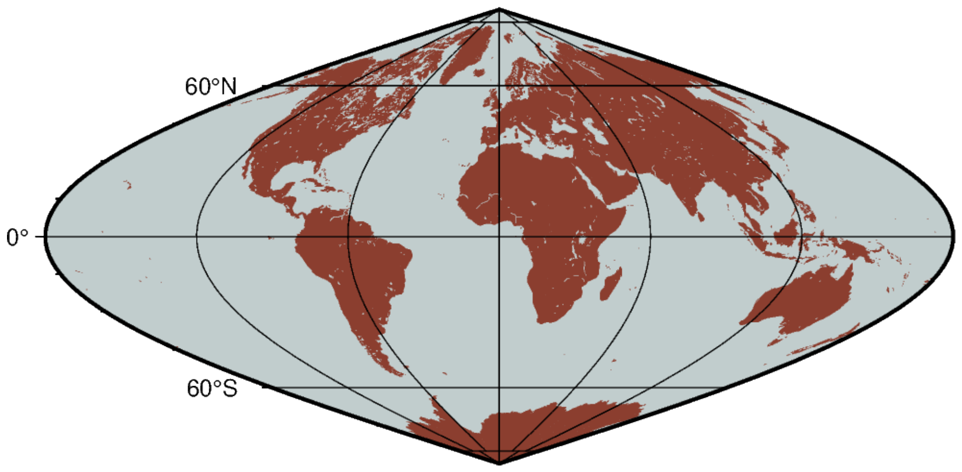
Total running time of the script: (0 minutes 2.150 seconds)

Sinusoidal

The sinusoidal projection is one of the oldest known projections, is equal-area, and has been used since the mid-16th century. It has also been called the "Equal-area Mercator" projection. The central meridian is a straight line; all other meridians are sinusoidal curves. Parallels are all equally spaced straight lines, with scale being true along all parallels (and central meridian).

i[*lon0/*]*scale* or **I**[*lon0/*]*width*

The projection is set with **i** or **I**. The central meridian is set with the optional *lon0*, and the figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()
# Use region "d" to specify global region (-180/180/-90/90)
fig.coast(region="d", projection="I12c", land="coral4", water="azure3", frame="afg")
fig.show()
```

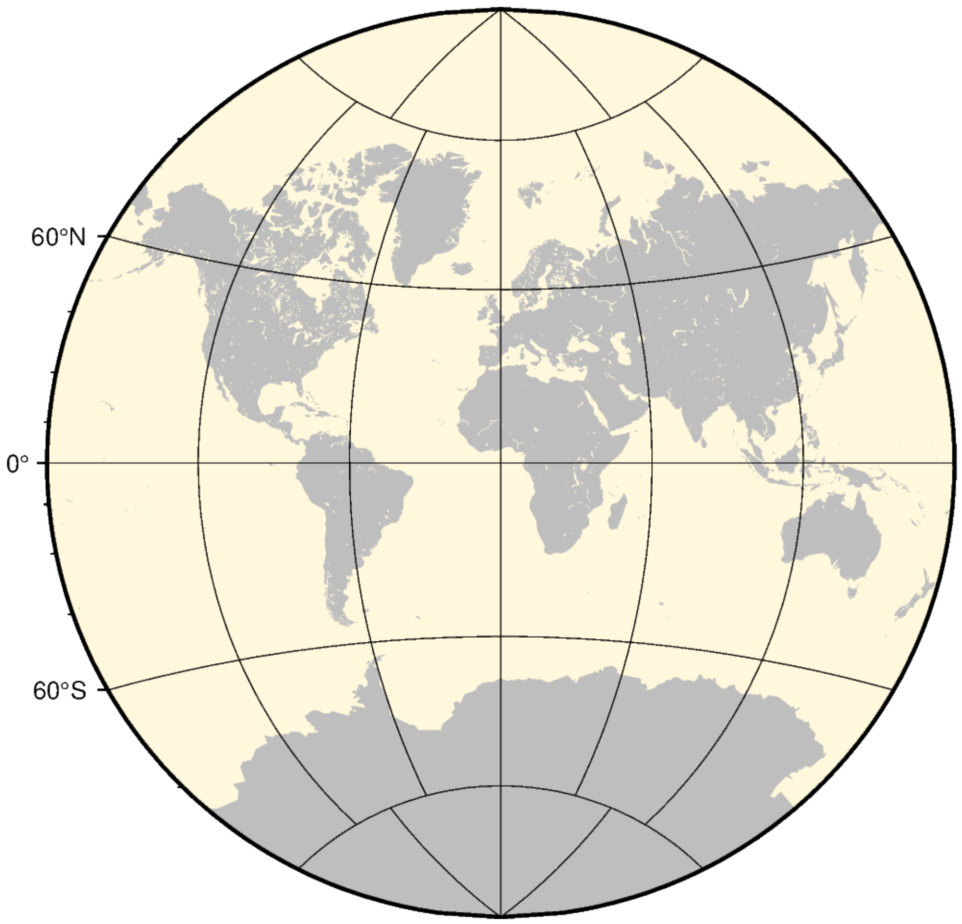
Total running time of the script: (0 minutes 2.193 seconds)

Van der Grinten

The Van der Grinten projection, presented by Alphons J. van der Grinten in 1904, is neither equal-area nor conformal. Central meridian and Equator are straight lines; other meridians are arcs of circles. The scale is true along the Equator only. Its main use is to show the entire world enclosed in a circle.

v[*lon0/*]*scale* or **V**[*lon0/*]*width*

The projection is set with **v** or **V**. The central meridian is set with the optional *lon0*, and the figure size is set with *scale* or *width*.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()
# Use region "d" to specify global region (-180/180/-90/90)
fig.coast(region="d", projection="V12c", land="gray", water="cornsilk", frame="afg")
fig.show()
```

Total running time of the script: (0 minutes 2.392 seconds)

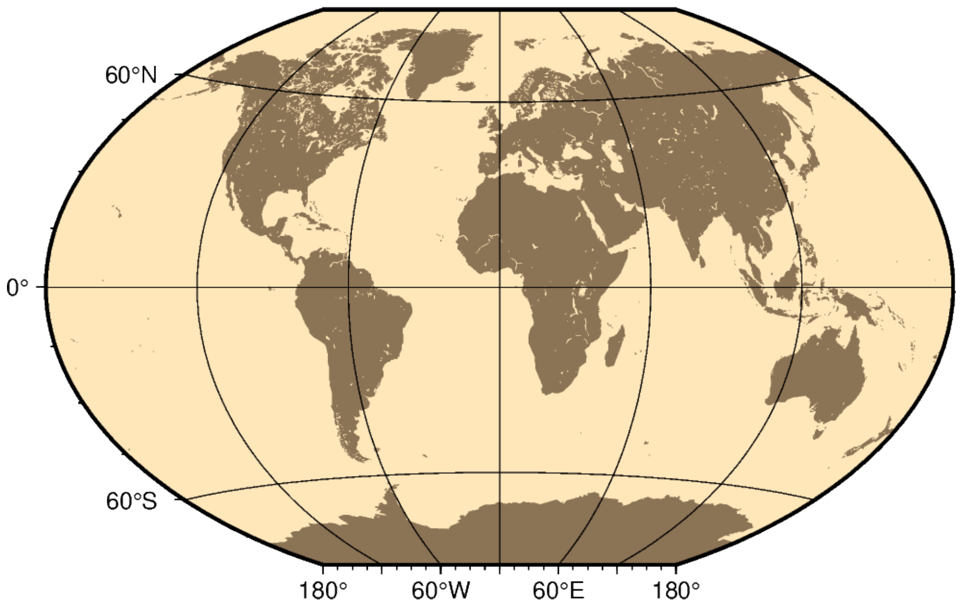
Winkel Tripel

In 1921, the German mathematician Oswald Winkel made a projection that was to strike a compromise between the properties of three elements (area, angle and distance). The German word “tripel” refers to this junction of where each of these elements are least distorted when plotting global maps. The projection was popularized when Bartholomew and Son started to use it in its world-renowned “The Times Atlas of the World” in the mid-20th century. In 1998, the National Geographic Society made the Winkel Tripel as its map projection of choice for global maps.

Naturally, this projection is neither conformal, nor equal-area. Central meridian and equator are straight lines; other parallels and meridians are curved. The projection is obtained by averaging the coordinates of the Equidistant Cylindrical and Aitoff (not Hammer-Aitoff) projections. The poles map into straight lines 0.4 times the length of equator.

r[*lon0/*]*scale* or **R**[*lon0/*]*width*

The projection is set with **r** or **R**. The central meridian is set with the optional *lon0*, and the figure size is set with *scale* or *width*.



Out:

```
coast [WARNING]: 3 annotations along the bottom border were skipped due to crowding.
coast [WARNING]: Crowding decisions is controlled by MAP_ANNOT_MIN_SPACING, currently␣
↪set to 24.9687p.
coast [WARNING]: Decrease or increase MAP_ANNOT_MIN_SPACING to see more or fewer␣
↪annotations, with 0 showing all annotations.
<IPython.core.display.Image object>
```

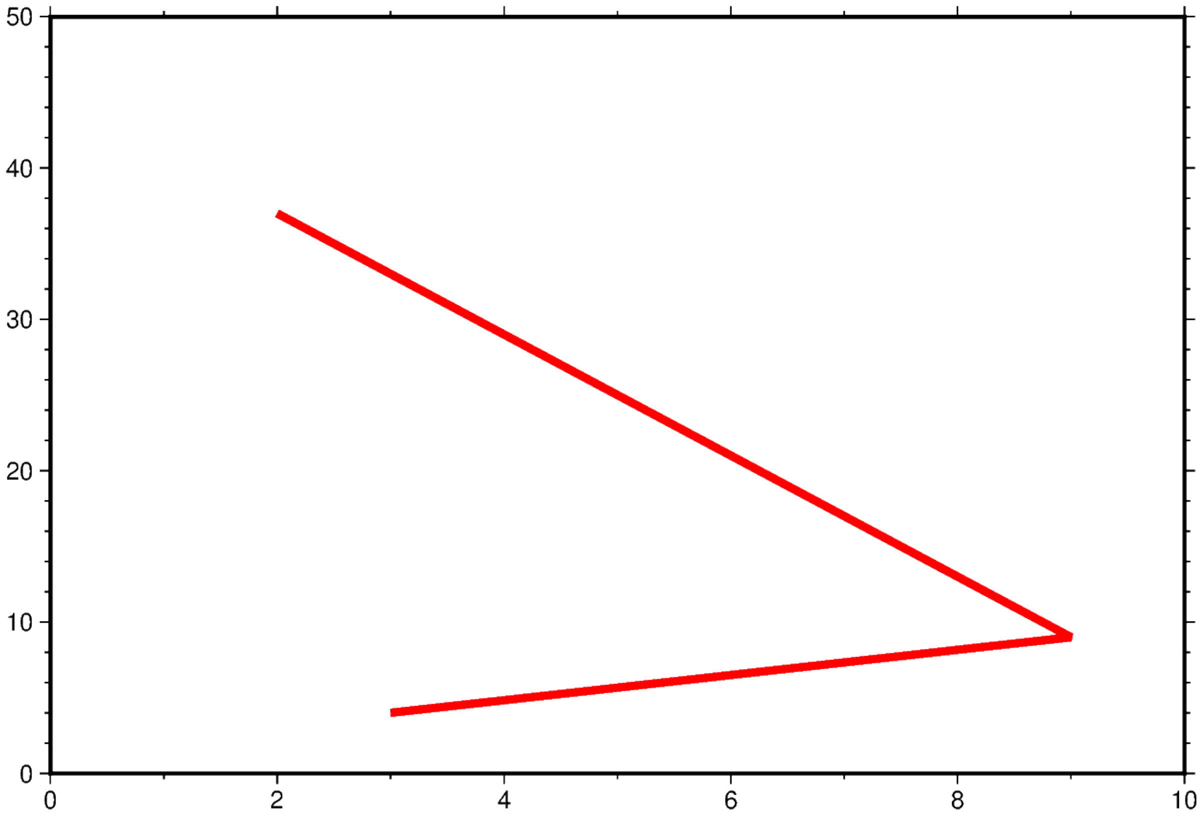
```python
import pygmt

fig = pygmt.Figure()
# Use region "d" to specify global region (-180/180/-90/90)
fig.coast(region="d", projection="R12c", land="burlywood4", water="wheat1", frame="afg")
fig.show()
```

Total running time of the script: (0 minutes 2.328 seconds)

9.7.5 Non-geographic Projections

Cartesian linear

X*width*/[*height*]: Give the *width* of the figure and the optional *height*.



Out:

```
<IPython.core.display.Image object>
```

```
import pygmt

fig = pygmt.Figure()
fig.plot(
    # The x and y parameters determine the coordinates of lines
    x=[3, 9, 2],
    y=[4, 9, 37],
    pen="3p,red",
    # ``region`` sets the x and y ranges or the Cartesian figure.
    region=[0, 10, 0, 50],
```

(continues on next page)

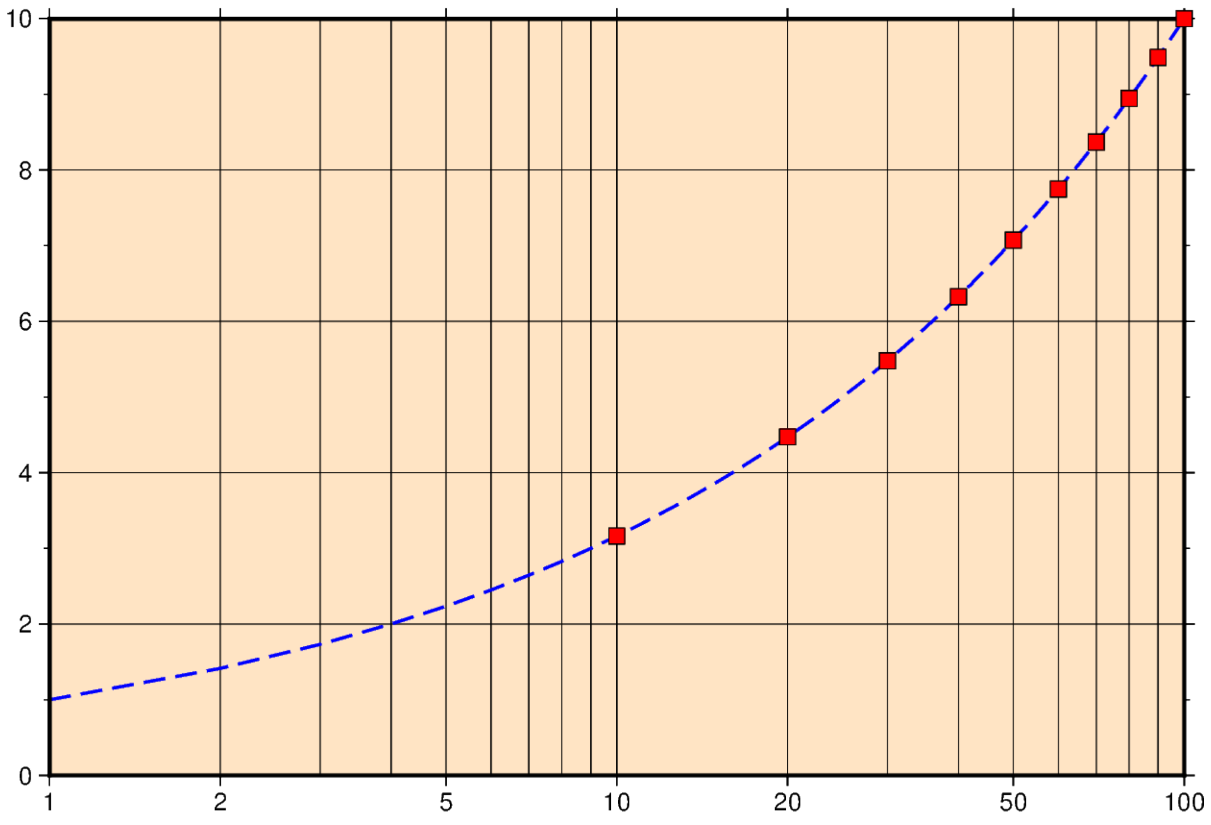
(continued from previous page)

```
    projection="X15c/10c",
    # ``WSne`` is passed to ``frame`` to put axis labels only on the left and
    # bottom axes.
    frame=["af", "WSne"],
)
fig.show()
```

Total running time of the script: (0 minutes 1.897 seconds)

Cartesian logarithmic

X*width*[**l**]/[*height*[**l**]]: Give the *width* of the figure and the optional *height*. The axis or axes with a logarithmic transformation requires **l** after its size argument.



Out:

```
<IPython.core.display.Image object>
```

```
import numpy as np
import pygmt
```

(continues on next page)

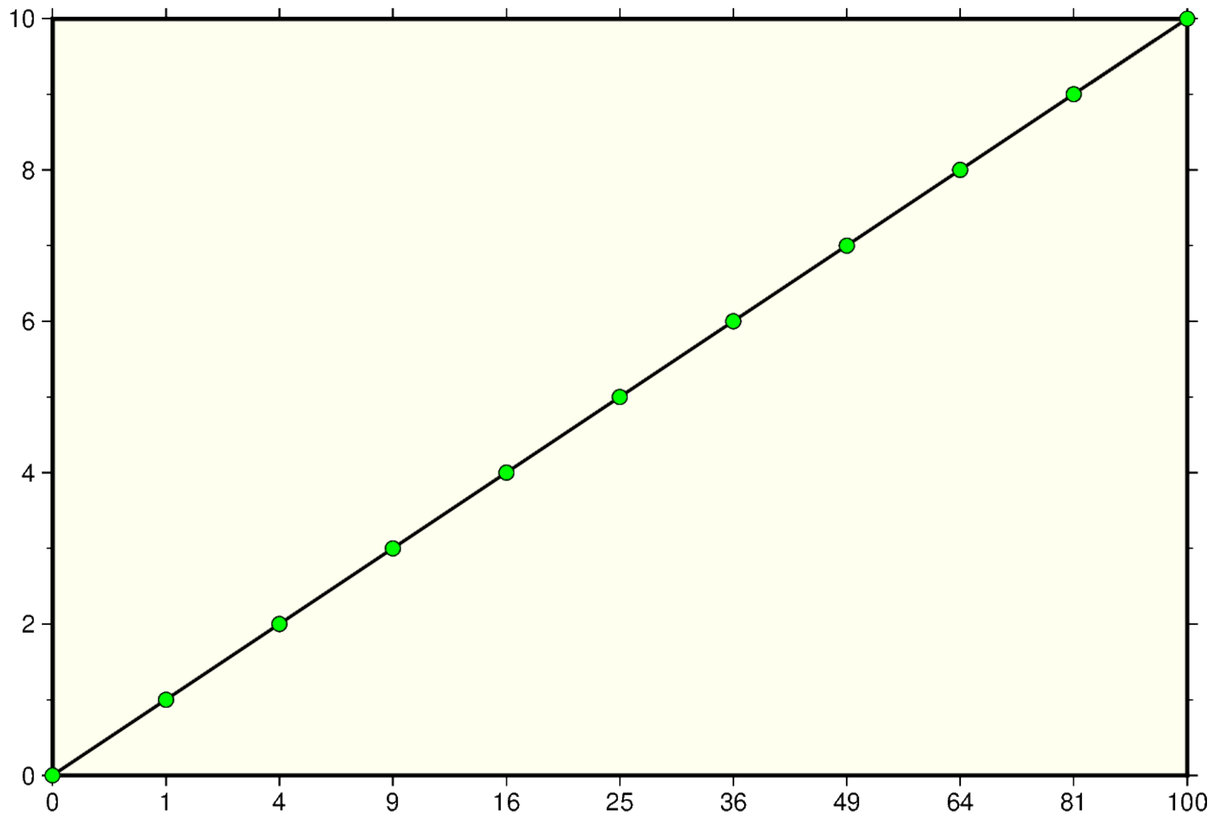
(continued from previous page)

```
# Create a list of x values 0-100
xline = np.arange(0, 101)
# Create a list of y-values that are the square root of the x-values
yline = xline ** 0.5
# Create a list of x values for every 10 in 0-100
xpoints = np.arange(0, 101, 10)
# Create a list of y-values that are the square root of the x-values
ypoints = xpoints ** 0.5

fig = pygmt.Figure()
fig.plot(
    region=[1, 100, 0, 10],
    # Set a logarithmic transformation on the x-axis
    projection="X15cl/10c",
    # Set the figures frame, color, and gridlines
    frame=["WSne+gbisque", "x2g3", "ya2f1g2"],
    x=xline,
    y=yline,
    # Set the line thickness to *1p*, the color to *blue*, and the style to
    # *dash*
    pen="1p,blue,-",
)
# Plot square root values as points on the line
# Style of points is 0.3 cm square, color is *red* with a *black* outline
# Points are not clipped if they go off the figure
fig.plot(x=xpoints, y=ypoints, style="s0.3c", color="red", no_clip=True, pen="black")
fig.show()
```

Total running time of the script: (0 minutes 2.345 seconds)

Cartesian power

X*width*[**p***pvalue*]/[*height*[**p***pvalue*]]: Give the *width* of the figure and the optional argument *height*. The axis or axes with a logarithmic transformation requires **p** and the power transformation for that axis.



Out:

```
<IPython.core.display.Image object>
```

```
import numpy as np
import pygmt

# Create a list of y values 0-10
yvalues = np.arange(0, 11)
# Create a list of x-values that are the square of the y-values
xvalues = yvalues ** 2

fig = pygmt.Figure()
fig.plot(
    region=[0, 100, 0, 10],
    # Set the power transformation of the x-axis, with a power of 0.5
    projection="X15cp0.5/10c",
    # Set the figures frame, color, and gridlines
    frame=["WSne+givory", "xa1p", "ya2f1"],
    # Set the line thickness to *thick*
    # Use the default color *black* and the default style *solid*
    pen="thick",
    x=xvalues,
```

(continues on next page)

(continued from previous page)

```
    y=yvalues,
)
# Plot x,y values as points on the line
# Style of points is 0.2 cm circles, color is *green* with a *black* outline
# Points are not clipped if they go off the figure
fig.plot(x=xvalues, y=yvalues, style="c0.2c", color="green", no_clip=True, pen="black")
fig.show()
```

Total running time of the script: (0 minutes 2.312 seconds)

Polar

Polar projections allow plotting polar coordinate data (e.g. angle θ and radius r).

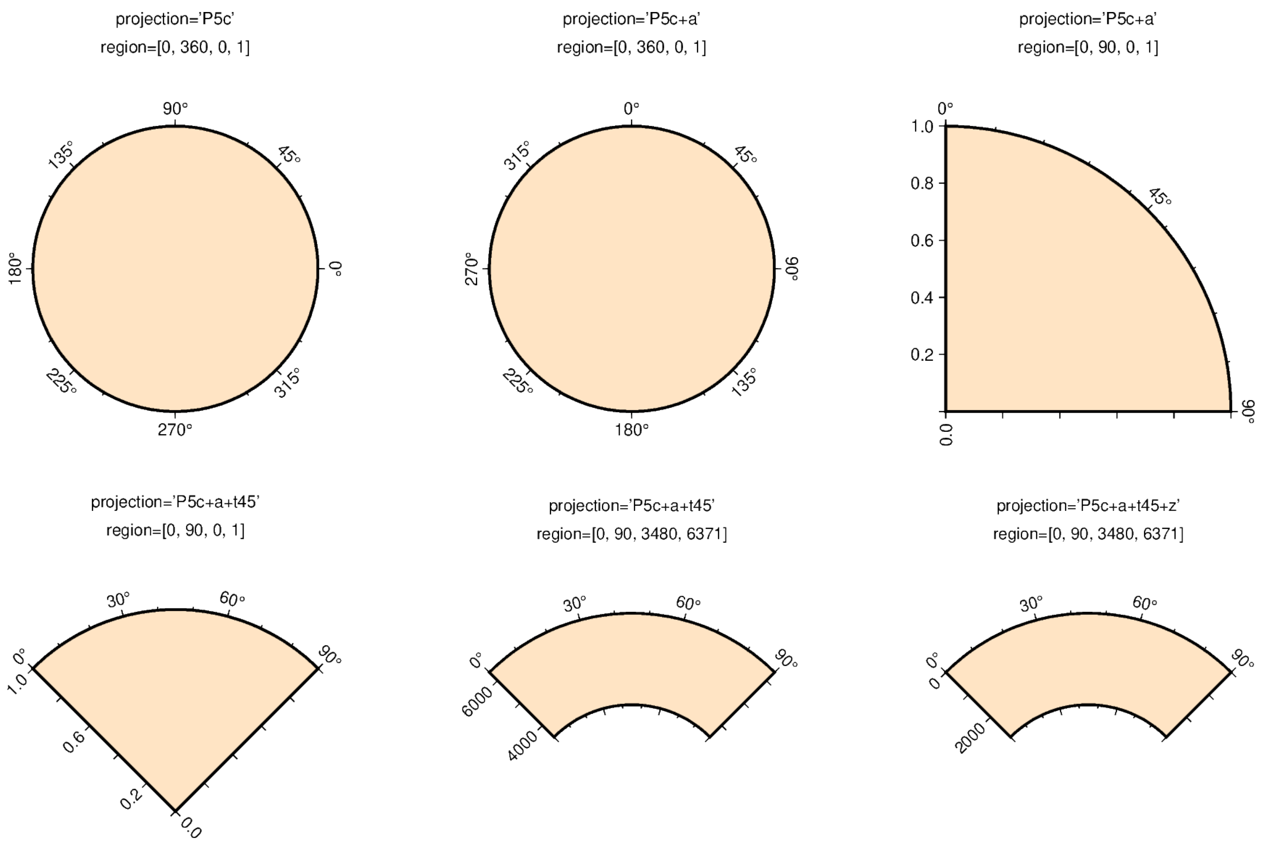
The full syntax for polar projections is:

P*width*[**+a**][**+f**[**e**|**p**|*radius*]][**+r***offset*][**+t***origin*][**+z**[**p**|*radius*]]

Limits are set via the `region` parameter ([*theta_min*, *theta_max*, *radius_min*, *radius_max*]). When using **P***width* you have to give the *width* of the figure. The lower-case version **p** is similar to **P** but expects a *scale* instead of a width (**p***scale*).

The following customizing modifiers are available:

- **+a**: by default, θ refers to the angle that is equivalent to a counterclockwise rotation with respect to the east direction (standard definition); **+a** indicates that the input data is rotated clockwise relative to the north direction (geographical azimuth angle).
- **+r***offset*: represents the offset of the r axis. This modifier allows you to offset the center of the circle from r=0.
- **+t***origin*: sets the angle corresponding to the east direction which is equivalent to rotating the entire coordinate axis clockwise; if the **+a** modifier is used, setting the angle corresponding to the north direction is equivalent to rotating the entire coordinate axis counterclockwise.
- **+f**: reverses the radial direction.
  - Append **e** to indicate that the r-axis is an elevation angle, and the range of the r-axis should be between 0 and 90.
  - Appending **p** sets the current earth radius (determined by PROJ_ELLIPSOID) to the maximum value of the r axis when the r axis is reversed.
  - Append *radius* to set the maximum value of the r axis.
- **+z**: indicates that the r axis is marked as depth instead of radius (e.g. $r = radius - z$).
  - Append **p** to set radius to the current earth radius.
  - Append *radius* to set the value of the radius.



Out:

```
basemap [WARNING]: 2 annotations along the right border were skipped due to crowding.
basemap [WARNING]: Crowding decisions is controlled by MAP_ANNOT_MIN_SPACING, currently␣
↪set to 23.0363p.
basemap [WARNING]: Decrease or increase MAP_ANNOT_MIN_SPACING to see more or fewer␣
↪annotations, with 0 showing all annotations.
basemap [WARNING]: 1 annotations along the right border were skipped due to crowding.
basemap [WARNING]: Crowding decisions is controlled by MAP_ANNOT_MIN_SPACING, currently␣
↪set to 22.6964p.
basemap [WARNING]: Decrease or increase MAP_ANNOT_MIN_SPACING to see more or fewer␣
↪annotations, with 0 showing all annotations.
basemap [WARNING]: 1 annotations along the right border were skipped due to crowding.
basemap [WARNING]: Crowding decisions is controlled by MAP_ANNOT_MIN_SPACING, currently␣
↪set to 22.6964p.
basemap [WARNING]: Decrease or increase MAP_ANNOT_MIN_SPACING to see more or fewer␣
↪annotations, with 0 showing all annotations.
<IPython.core.display.Image object>
```

```python
import pygmt

fig = pygmt.Figure()
```

(continues on next page)

(continued from previous page)

```
pygmt.config(FONT_TITLE="14p,Helvetica,black", FORMAT_GEO_MAP="+D")

# ============

fig.basemap(
    # set map limits to theta_min = 0, theta_max = 360, radius_min = 0,
    # radius_max = 1
    region=[0, 360, 0, 1],
    # set map width to 5 cm
    projection="P5c",
    # set the frame and color
    frame=["xa45f", "+gbisque"],
)

fig.text(position="TC", text="projection='P5c'", offset="0/2.0c", no_clip=True)
fig.text(position="TC", text="region=[0, 360, 0, 1]", offset="0/1.5c", no_clip=True)

fig.shift_origin(xshift="8c")

# ============
fig.basemap(
    # set map limits to theta_min = 0, theta_max = 360, radius_min = 0,
    # radius_max = 1
    region=[0, 360, 0, 1],
    # set map width to 5 cm and interpret input data as geographic azimuth
    # instead of standard angle
    projection="P5c+a",
    # set the frame and color
    frame=["xa45f", "+gbisque"],
)

fig.text(position="TC", text="projection='P5c+a'", offset="0/2.0c", no_clip=True)
fig.text(position="TC", text="region=[0, 360, 0, 1]", offset="0/1.5c", no_clip=True)

fig.shift_origin(xshift="8c")

# ============
fig.basemap(
    # set map limits to theta_min = 0, theta_max = 90, radius_min = 0,
    # radius_max = 1
    region=[0, 90, 0, 1],
    # set map width to 5 cm and interpret input data as geographic azimuth
    # instead of standard angle
    projection="P5c+a",
    # set the frame and color
    frame=["xa45f", "ya0.2", "WNe+gbisque"],
)

fig.text(position="TC", text="projection='P5c+a'", offset="0/2.0c", no_clip=True)
fig.text(position="TC", text="region=[0, 90, 0, 1]", offset="0/1.5c", no_clip=True)
```

(continues on next page)

(continued from previous page)

```python
fig.shift_origin(xshift="-16c", yshift="-7c")

# ============
fig.basemap(
    # set map limits to theta_min = 0, theta_max = 90, radius_min = 0,
    # radius_max = 1
    region=[0, 90, 0, 1],
    # set map width to 5 cm and interpret input data as geographic azimuth
    # instead of standard angle, rotate coordinate system counterclockwise by
    # 45 degrees
    projection="P5c+a+t45",
    # set the frame and color
    frame=["xa30f", "ya0.2", "WNe+gbisque"],
)

fig.text(position="TC", text=r"projection='P5c+a\+t45'", offset="0/2.0c", no_clip=True)
fig.text(position="TC", text="region=[0, 90, 0, 1]", offset="0/1.5c", no_clip=True)

fig.shift_origin(xshift="8c", yshift="1.3c")

# ============
fig.basemap(
    # set map limits to theta_min = 0, theta_max = 90, radius_min = 3480,
    # radius_max = 6371 (Earth's radius)
    region=[0, 90, 3480, 6371],
    # set map width to 5 cm and interpret input data as geographic azimuth
    # instead of standard angle, rotate coordinate system counterclockwise by
    # 45 degrees
    projection="P5c+a+t45",
    # set the frame and color
    frame=["xa30f", "ya", "WNse+gbisque"],
)

fig.text(position="TC", text=r"projection='P5c+a\+t45'", offset="0/2.0c", no_clip=True)
fig.text(
    position="TC", text="region=[0, 90, 3480, 6371]", offset="0/1.5c", no_clip=True
)

fig.shift_origin(xshift="8c")

# ============
fig.basemap(
    # set map limits to theta_min = 0, theta_max = 90, radius_min = 3480,
    # radius_max = 6371 (Earth's radius)
    region=[0, 90, 3480, 6371],
    # set map width to 5 cm and interpret input data as geographic azimuth
    # instead of standard angle, rotate coordinate system counterclockwise by
    # 45 degrees, r axis is marked as depth
    projection="P5c+a+t45+z",
    # set the frame and color
    frame=["xa30f", "ya", "WNse+gbisque"],
)
```

(continues on next page)

(continued from previous page)

```
fig.text(
    position="TC", text=r"projection='P5c+a\+t45+z'", offset="0/2.0c", no_clip=True
)
fig.text(
    position="TC", text="region=[0, 90, 3480, 6371]", offset="0/1.5c", no_clip=True
)

fig.show()
```

Total running time of the script: (0 minutes 8.796 seconds)

9.7.6 Projection Table

The below table shows the projection codes for the 31 GMT projections.

PyGMT Projection Argument	Projection Name
Alon$_0$/lat$_0$[/*horizon*]/*width*	*Lambert azimuthal equal area*
Blon$_0$/lat$_0$/lat$_1$/lat$_2$/*width*	*Albers conic equal area*
Clon$_0$/lat$_0$/*width*	*Cassini cylindrical*
Cyl_stere/[lon$_0$[/lat$_0$/]]*width*	*Cylindrical stereographic*
Dlon$_0$/lat$_0$/lat$_1$/lat$_2$/*width*	*Equidistant conic*
Elon$_0$/lat$_0$[/*horizon*]/*width*	*Azimuthal equidistant*
Flon$_0$/lat$_0$[/*horizon*]/*width*	*Azimuthal gnomonic*
Glon$_0$/lat$_0$[/*horizon*]/*width*	*Azimuthal orthographic*
Glon$_0$/lat$_0$/*alt*/*azim*/*tilt*/*twist*/*W*/*H*/*width*	*General perspective*
H[lon$_0$/]*width*	*Hammer equal area*
I[lon$_0$/]*width*	*Sinusoidal equal area*
J[lon$_0$/]*width*	*Miller cylindrical*
Kf[lon$_0$/]*width*	*Eckert IV equal area*
Ks[lon$_0$/]*width*	*Eckert VI equal area*
Llon$_0$/lat$_0$/lat$_1$/lat$_2$/*width*	*Lambert conic conformal*
M[lon$_0$[/lat$_0$]/]*width*	*Mercator cylindrical*
N[lon$_0$/]*width*	*Robinson*
Oalon$_0$/lat$_0$/*azim*/*width*[**+v**]	*Oblique Mercator, 1: origin and azim*
Oblon$_0$/lat$_0$/lon$_1$/lat$_1$/*width*[**+v**]	*Oblique Mercator, 2: two points*
Oclon$_0$/lat$_0$/lon$_p$/lat$_p$/*width*[**+v**]	*Oblique Mercator, 3: origin and pole*
P*width*[**+a**][**+f**[**e**|**p**|*radius*]][**+r***offset*][**+t***origin*][**+z**[**p**|*radius*]]	*Polar* [azimuthal] (θ, r) (or cylindrical)
Poly[lon$_0$[/lat$_0$]/]*width*	*Polyconic*
Q[lon$_0$[/lat$_0$/]]*width*	*Equidistant cylindrical*
R[lon$_0$/]*width*	*Winkel Tripel*
Slon$_0$/lat$_0$[/*horizon*]/*width*	*General stereographic*
T[lon$_0$[/lat$_0$]/]*width*	*Transverse Mercator*
U*zone*/*width*	*Universal Transverse Mercator (UTM)*
V[lon$_0$/]*width*	*Van der Grinten*
W[lon$_0$/]*width*	*Mollweide*
X*width*[**l**|**p***exp*|**T**|**t**][/*height*[**l**|**p***exp*|**T**|**t**]][**d**]	*Linear*, *logarithmic*, *power*, and time
Ylon$_0$/lat$_0$/*width*	*Cylindrical equal area*

9.8 Coastlines and borders

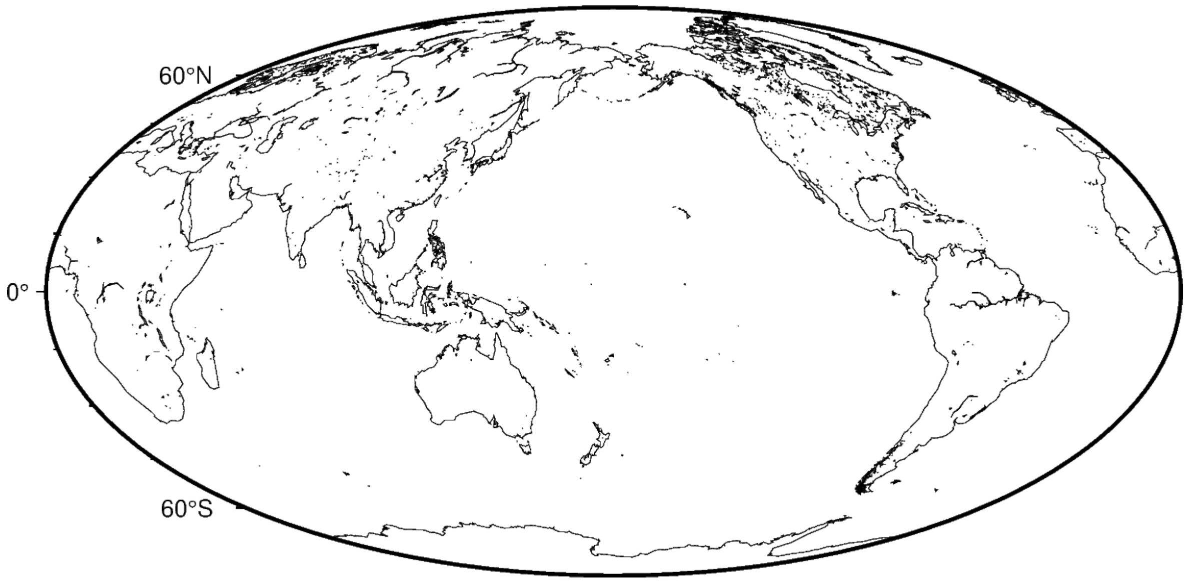
Plotting coastlines and borders is handled by *pygmt.Figure.coast*.

```
import pygmt
```

9.8.1 Shorelines

Use the `shorelines` parameter to plot only the shorelines:

```
fig = pygmt.Figure()
fig.basemap(region="g", projection="W15c", frame=True)
fig.coast(shorelines=True)
fig.show()
```



Out:

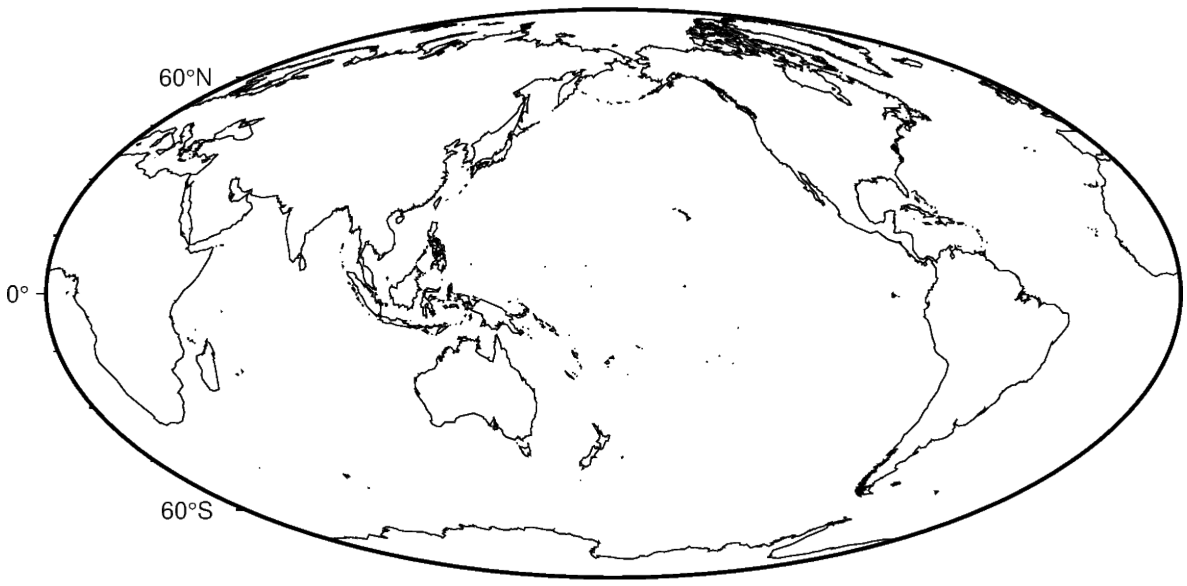
```
<IPython.core.display.Image object>
```

The shorelines are divided in 4 levels:

1. coastline
2. lakeshore
3. island-in-lake shore
4. lake-in-island-in-lake shore

You can specify which level you want to plot by passing the level number and a GMT pen configuration. For example, to plot just the coastlines with 0.5 thickness and black lines:

```
fig = pygmt.Figure()
fig.basemap(region="g", projection="W15c", frame=True)
fig.coast(shorelines="1/0.5p,black")
fig.show()
```

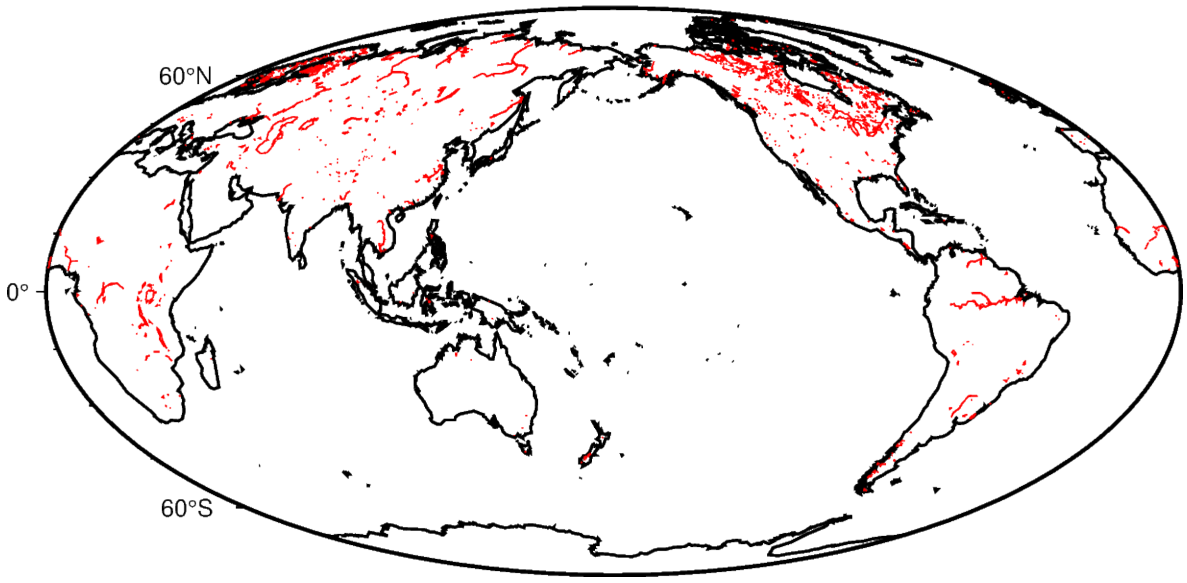


Out:

```
<IPython.core.display.Image object>
```

You can specify multiple levels (with their own pens) by passing a list to `shorelines`:

```
fig = pygmt.Figure()
fig.basemap(region="g", projection="W15c", frame=True)
fig.coast(shorelines=["1/1p,black", "2/0.5p,red"])
fig.show()
```



Out:

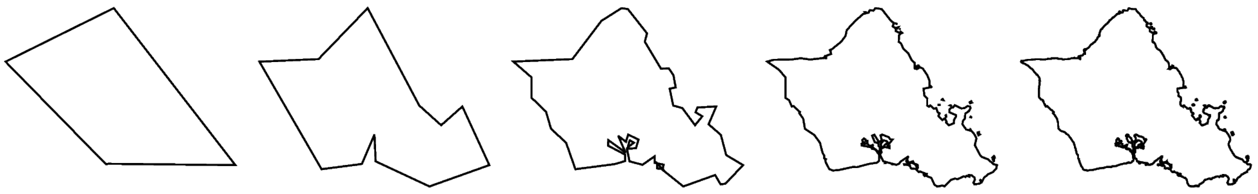
```
<IPython.core.display.Image object>
```

9.8.2 Resolutions

The coastline database comes with 5 resolutions. The resolution drops by 80% between levels:

1. `"c"`: crude
2. `"l"`: low (default)
3. `"i"`: intermediate
4. `"h"`: high
5. `"f"`: full

```
oahu = [-158.3, -157.6, 21.2, 21.8]
fig = pygmt.Figure()
for res in ["c", "l", "i", "h", "f"]:
    fig.coast(resolution=res, shorelines="1p", region=oahu, projection="M5c")
    fig.shift_origin(xshift="5c")
fig.show()
```



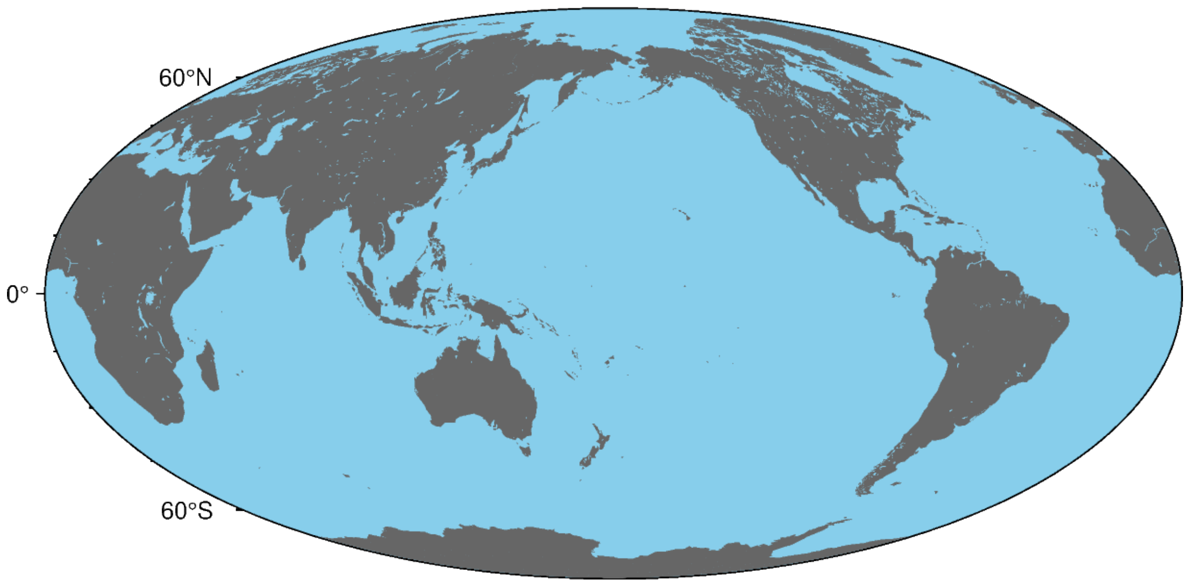
Out:

```
<IPython.core.display.Image object>
```

9.8.3 Land and water

Use the `land` and `water` parameters to specify a fill color for land and water bodies. The colors can be given by name or hex codes (like the ones used in HTML and CSS):

```
fig = pygmt.Figure()
fig.basemap(region="g", projection="W15c", frame=True)
fig.coast(land="#666666", water="skyblue")
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

Total running time of the script: (0 minutes 14.752 seconds)

9.9 Setting the region

Many of the plotting functions take the `region` parameter, which sets the area that will be shown in the figure. This tutorial covers the different types of inputs that it can accept.

```
import pygmt
```

9.9.1 Coordinates

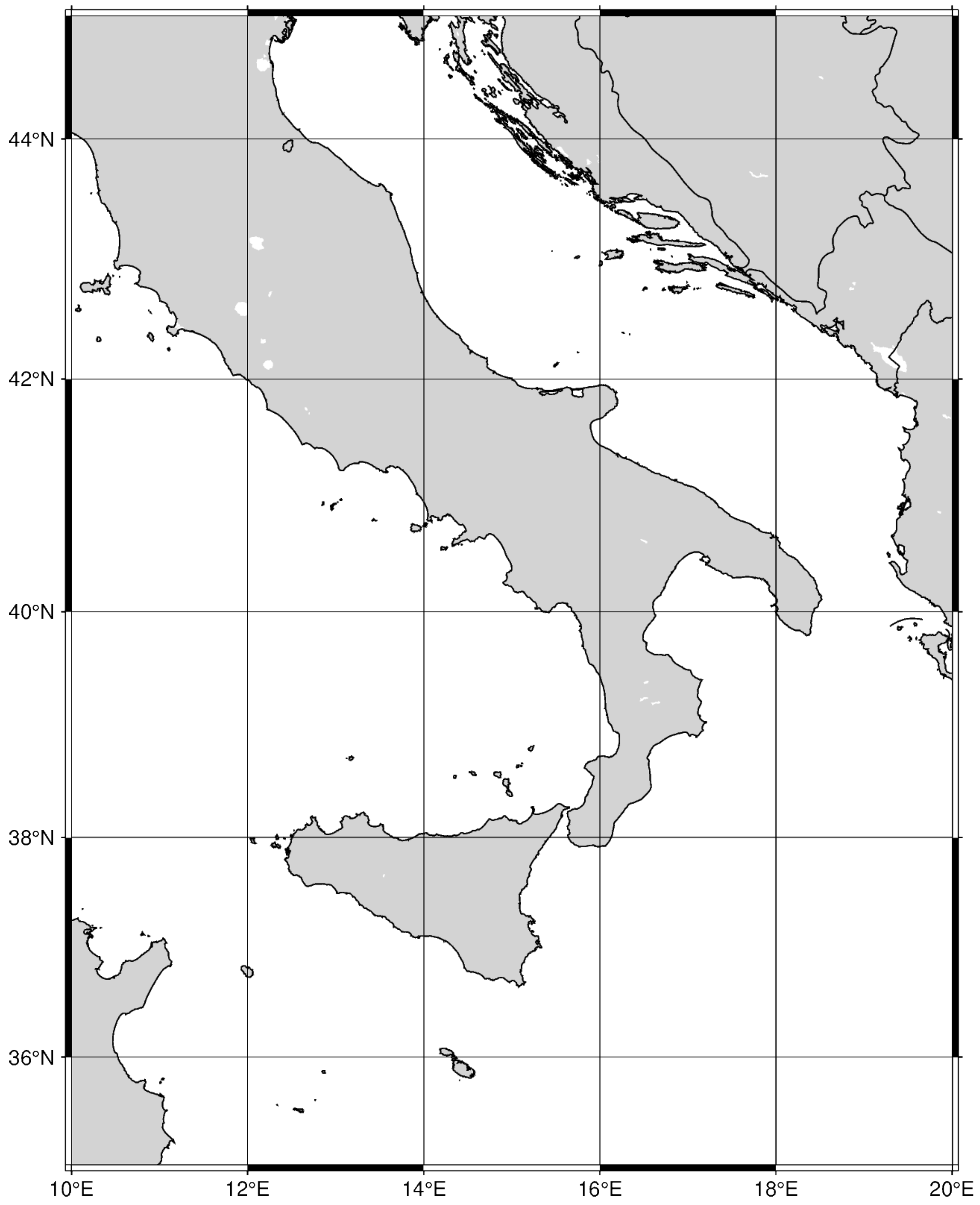
A string of coordinates can be passed to `region`, in the form of *xmin/xmax/ymin/ymax*.

```
fig = pygmt.Figure()
fig.coast(
    # Set the x-range from 10E to 20E and the y-range to 35N to 45N
    region="10/20/35/45",
    # Set projection to Mercator, and the figure size to 15 centimeters
    projection="M15c",
    # Set the color of the land to light gray
    land="lightgray",
    # Set the color of the water to white
    water="white",
    # Display the national borders and set the pen thickness to 0.5p
    borders="1/0.5p",
    # Display the shorelines and set the pen thickness to 0.5p
    shorelines="1/0.5p",
    # Set the frame to display annotations and gridlines
```

(continues on next page)

(continued from previous page)

```
    frame="ag",
)
fig.show()
```

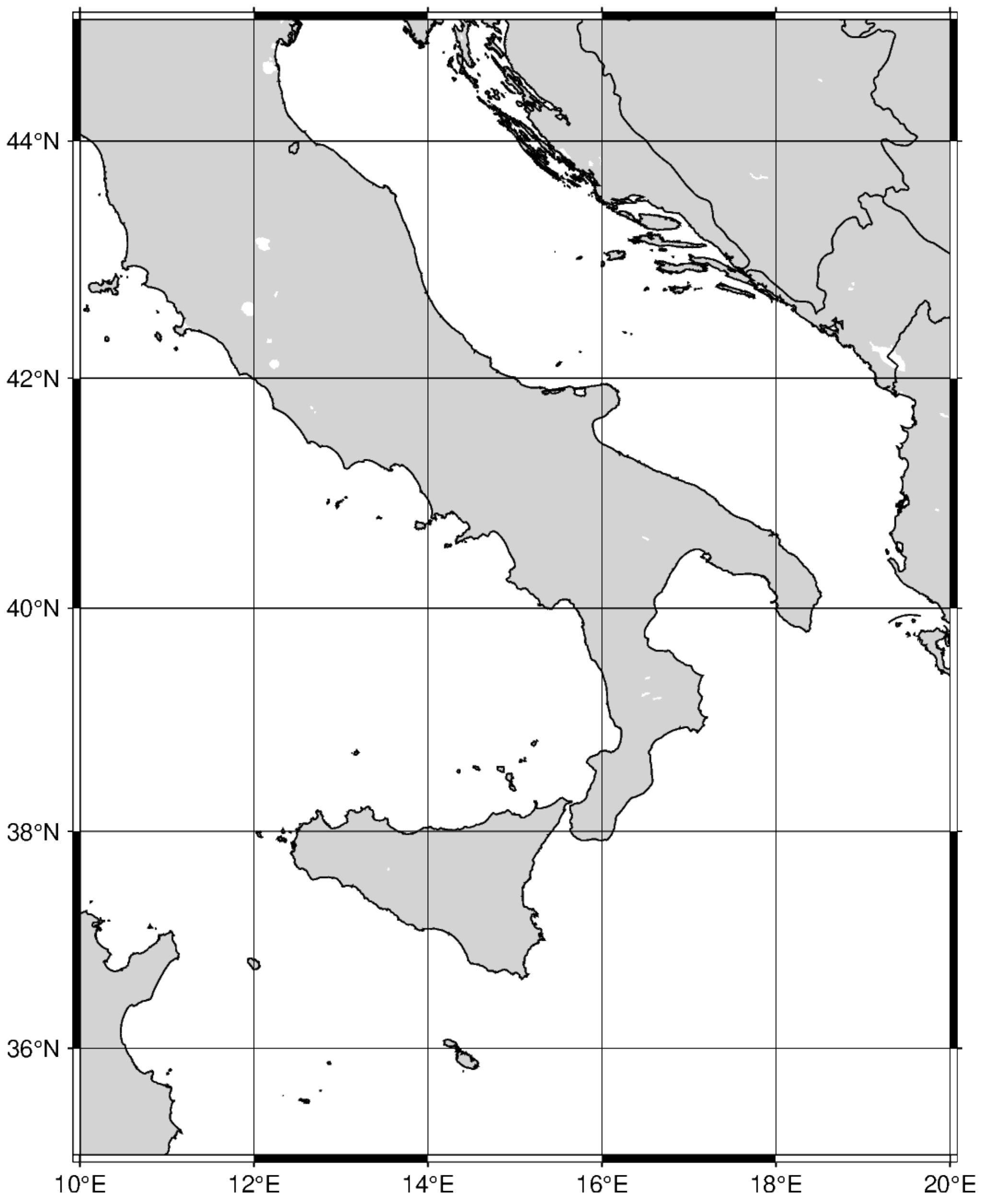


Out:

```
<IPython.core.display.Image object>
```

The coordinates can be passed to `region` as a list, in the form of [*xmin*,*xmax*,*ymin*,*ymax*].

```
fig = pygmt.Figure()
fig.coast(
    # Set the x-range from 10E to 20E and the y-range to 35N to 45N
    region=[10, 20, 35, 45],
    projection="M12c",
    land="lightgray",
    water="white",
    borders="1/0.5p",
    shorelines="1/0.5p",
    frame="ag",
)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

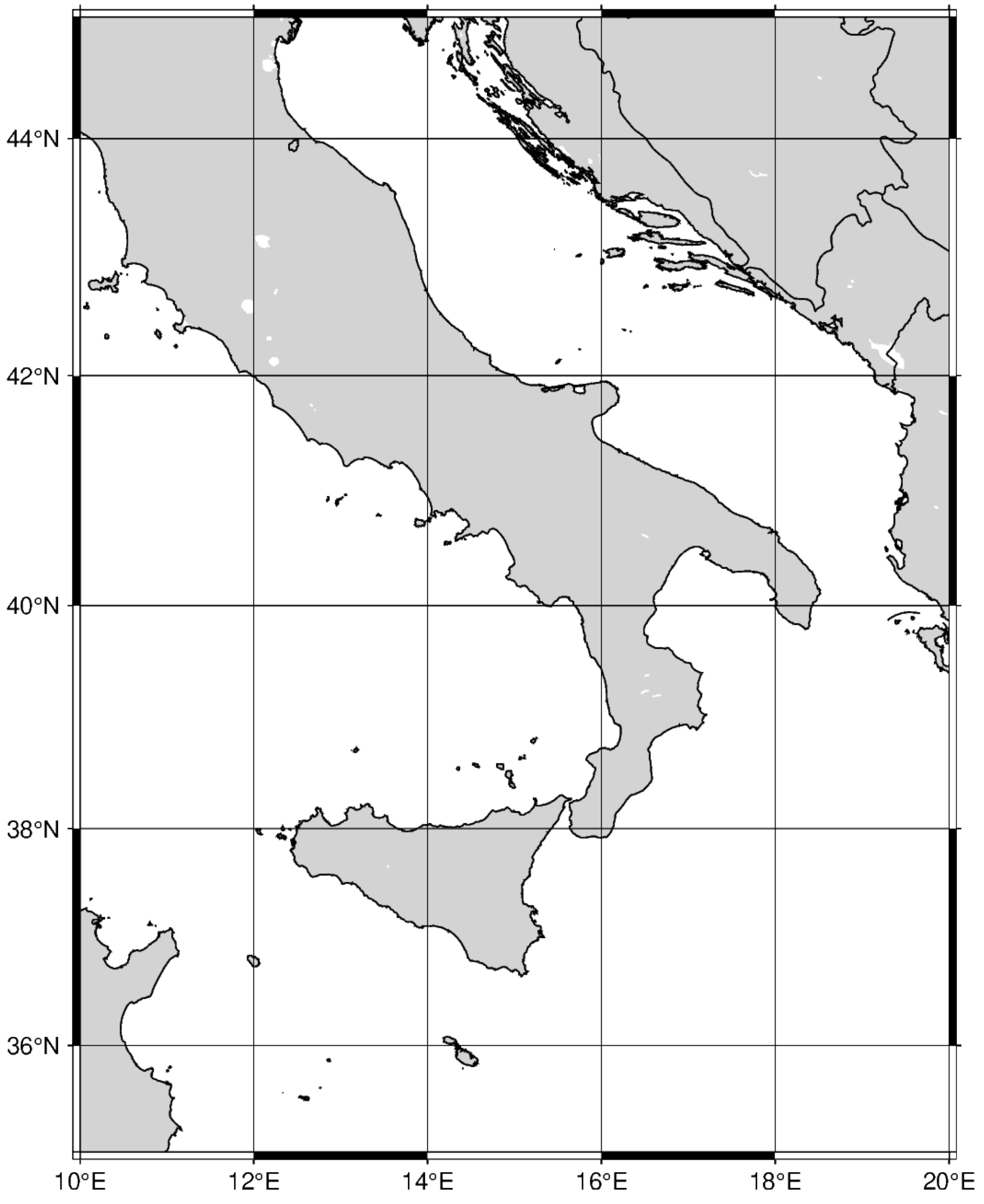
Instead of passing axes minima and maxima, the coordinates can be passed for the bottom-left and top-right corners. The string format takes the coordinates for the bottom-left and top-right coordinates. To specify corner coordinates, append **+r** at the end of the `region` string.

```
fig = pygmt.Figure()
fig.coast(
    # Set the bottom-left corner as 10E, 35N and the top-right corner as
    # 20E, 45N
    region="10/35/20/45+r",
    projection="M12c",
```

(continues on next page)

(continued from previous page)

```
    land="lightgray",
    water="white",
    borders="1/0.5p",
    shorelines="1/0.5p",
    frame="ag",
)
fig.show()
```



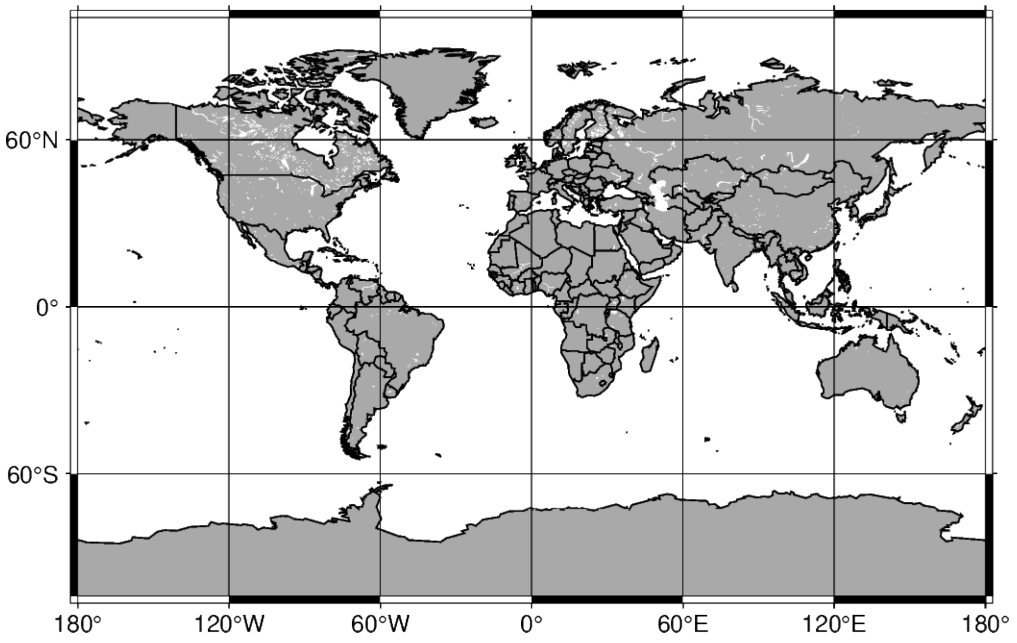
Out:

```
<IPython.core.display.Image object>
```

9.9.2 Global regions

In addition to passing coordinates, the argument **d** can be passed to set the region to the entire globe. The range is 180W to 180E (-180, 180) and 90S to 90N (-90 to 90). With no parameters set for the projection, the figure defaults to be centered at the mid-point of both x- and y-axes. Using **d**, the figure is centered at (0, 0), or the intersection of the equator and prime meridian.

```
fig = pygmt.Figure()
fig.coast(
    region="d",
    projection="Cyl_stere/12c",
    land="darkgray",
    water="white",
    borders="1/0.5p",
    shorelines="1/0.5p",
    frame="ag",
)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

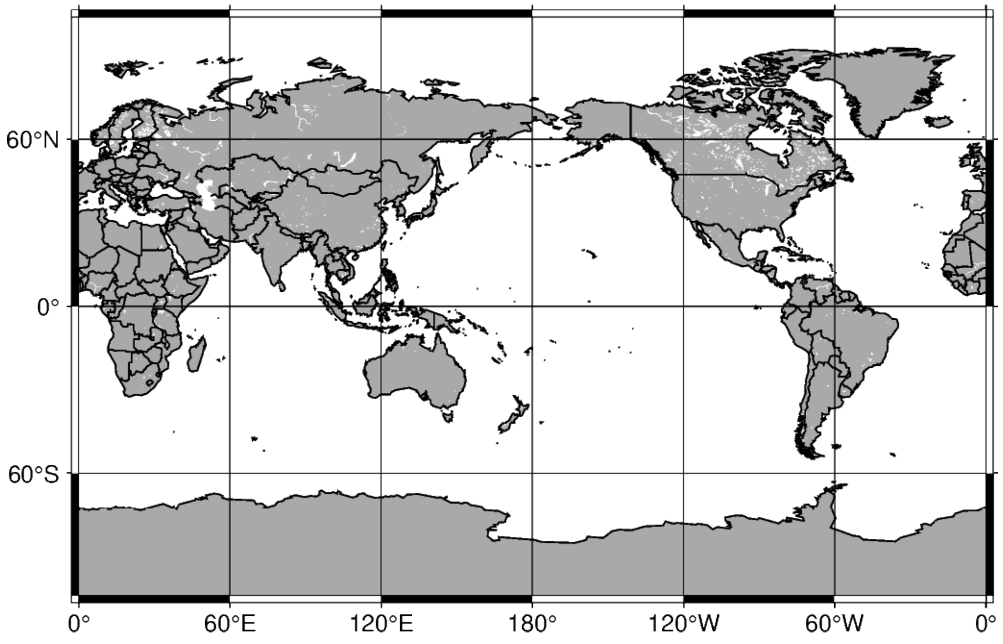
The argument **g** can be passed, which encompasses the entire globe. The range is 0E to 360E (0, 360) and 90S to 90N (-90 to 90). With no parameters set for the projection, the figure is centered at (180, 0), or the intersection of the equator and International Date Line.

```
fig = pygmt.Figure()
fig.coast(
    region="g",
    projection="Cyl_stere/12c",
    land="darkgray",
    water="white",
    borders="1/0.5p",
```

(continues on next page)

(continued from previous page)

```
    shorelines="1/0.5p",
    frame="ag",
)
fig.show()
```



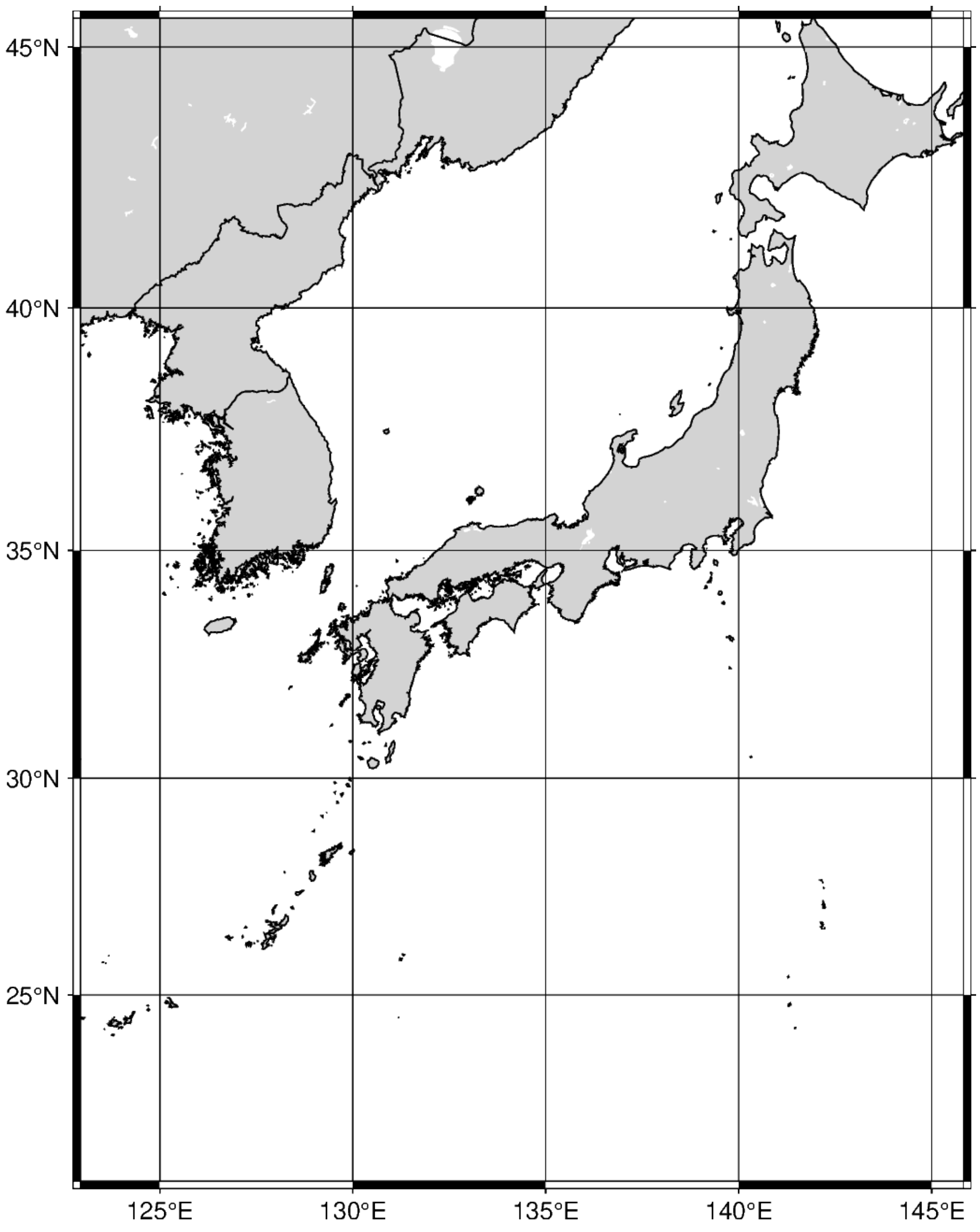
Out:

```
<IPython.core.display.Image object>
```

9.9.3 ISO code

The `region` can be set to include a specific area specified by the two-character ISO 3166-1 alpha-2 convention (for further information: https://en.wikipedia.org/wiki/ISO_3166-1_alpha-2).

```
fig = pygmt.Figure()
fig.coast(
    # Set the figure region to encompass Japan with the ISO code "JP"
    region="JP",
    projection="M12c",
    land="lightgray",
    water="white",
    borders="1/0.5p",
    shorelines="1/0.5p",
    frame="ag",
)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

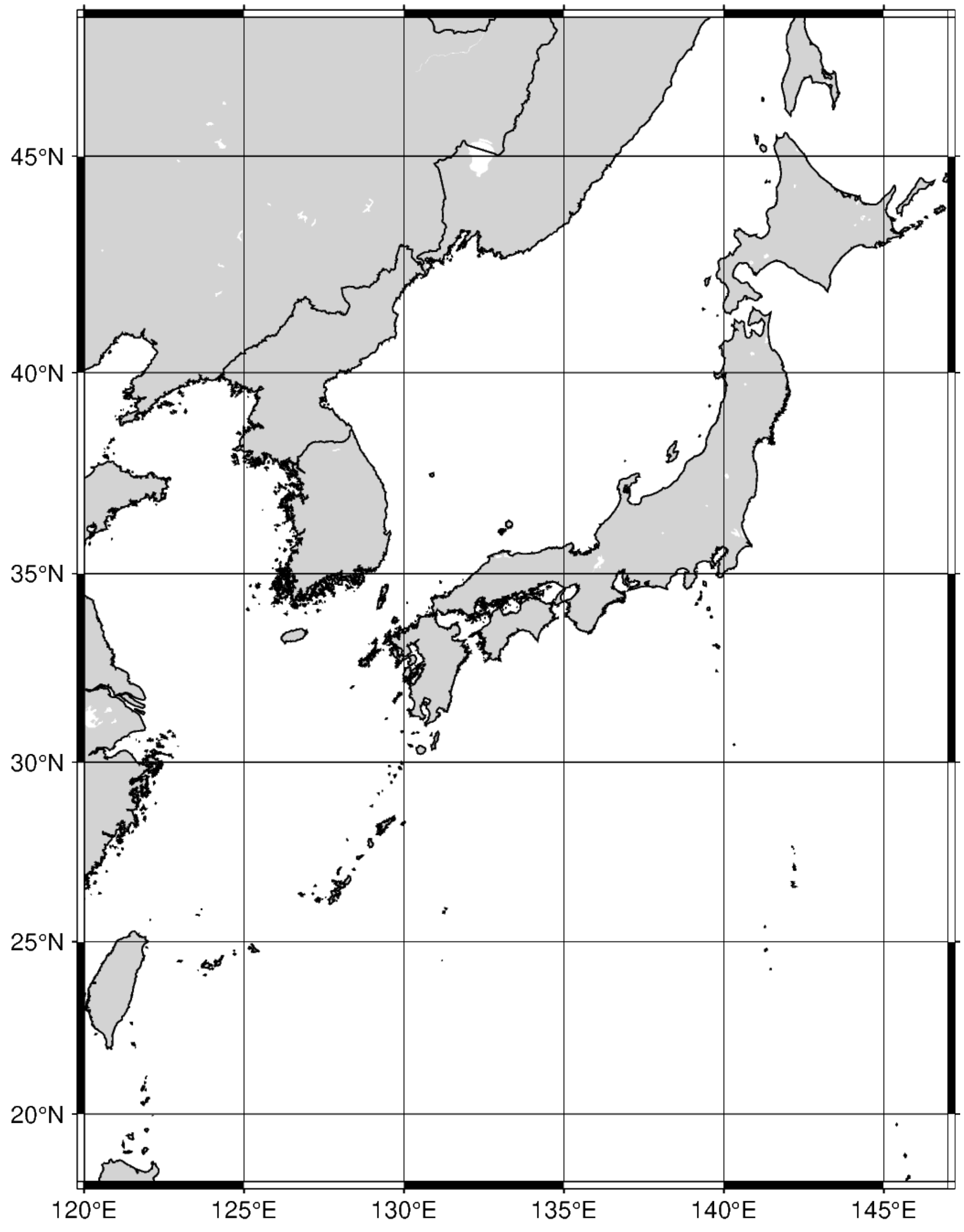
The area encompassed by the ISO code can be expanded by appending **+r***increment* to the ISO code. The *increment* unit is in degrees, and if only one value is added it expands the range of the region in all directions. Using **+r** expands the final region boundaries to be multiples of *increment* .

```
fig = pygmt.Figure()
fig.coast(
    # Expand the region boundaries to be multiples of 3 degrees in all
    # directions
    region="JP+r3",
    projection="M12c",
```

(continues on next page)

(continued from previous page)

```
    land="lightgray",
    water="white",
    borders="1/0.5p",
    shorelines="1/0.5p",
    frame="ag",
)
fig.show()
```

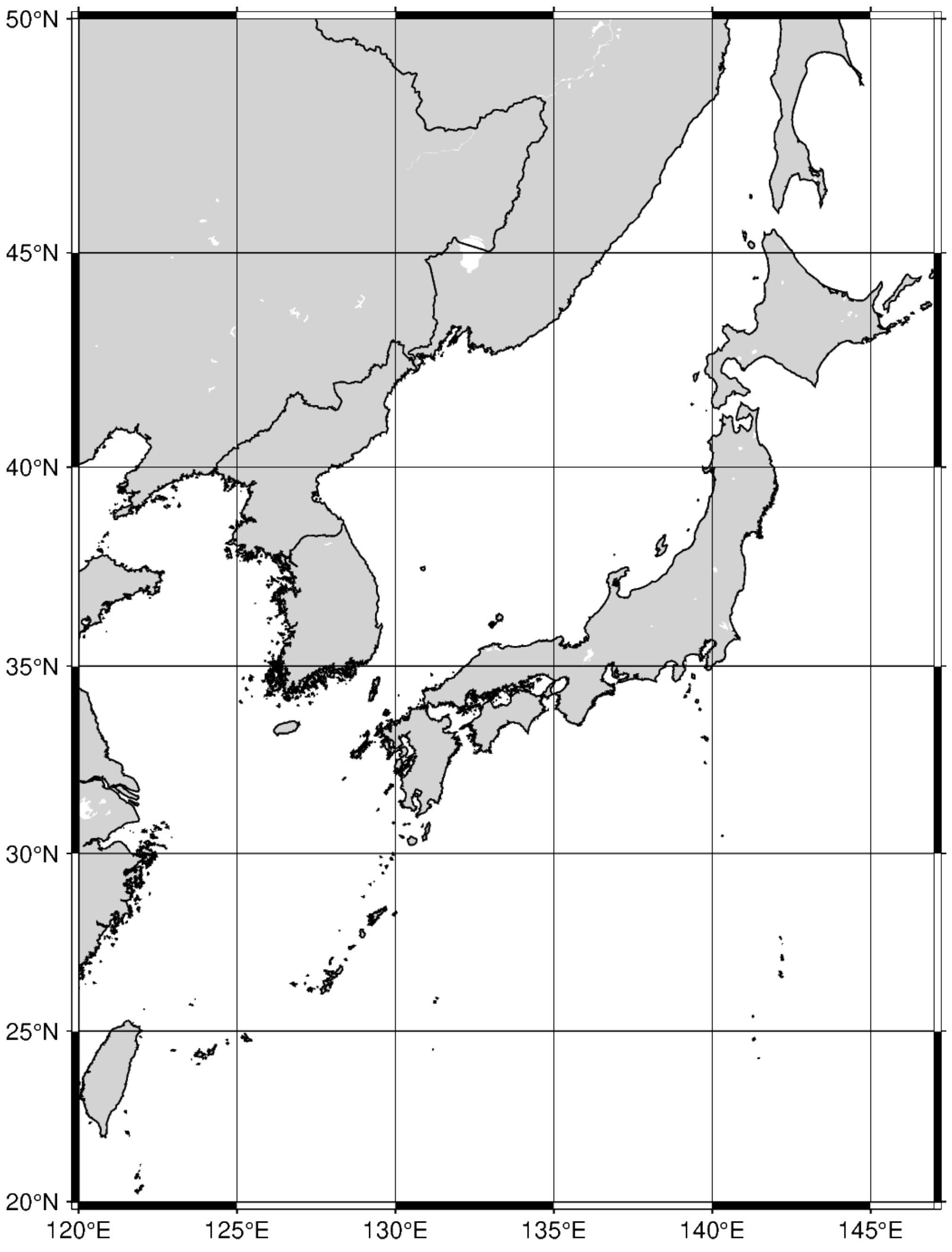


Out:

```
<IPython.core.display.Image object>
```

Instead of expanding the range of the plot uniformly in all directions, two values can be passed to expand differently on each axis. The format is *xinc*/*yinc*.

```
fig = pygmt.Figure()
fig.coast(
    # Expand the region boundaries to be multiples of 3 degrees on the x-axis
    # and 5 degrees on the y-axis.
    region="JP+r3/5",
    projection="M12c",
    land="lightgray",
    water="white",
    borders="1/0.5p",
    shorelines="1/0.5p",
    frame="ag",
)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

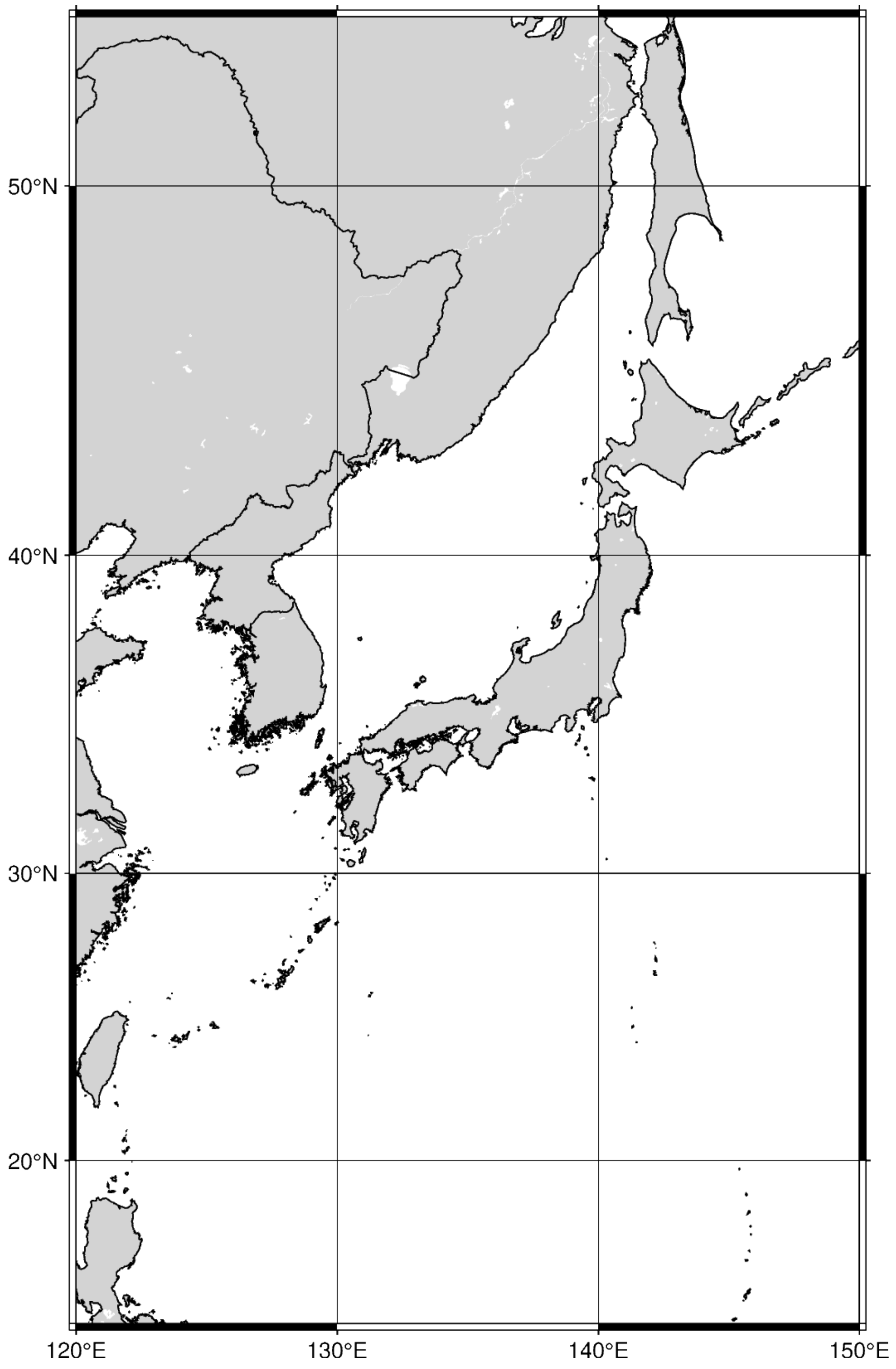
Instead of expanding the range of the plot uniformly in all directions, four values can be passed to expand differently in each direction. The format is *winc/einc/sinc/ninc*, which expands on the west, east, south, and north axes.

```
fig = pygmt.Figure()
fig.coast(
    # Expand the region boundaries to be multiples of 3 degrees to the west, 5
    # degrees to the east, 7 degrees to the south, and 9 degrees to the north.
    region="JP+r3/5/7/9",
```

(continues on next page)

(continued from previous page)

```
    projection="M12c",
    land="lightgray",
    water="white",
    borders="1/0.5p",
    shorelines="1/0.5p",
    frame="ag",
)
fig.show()
```

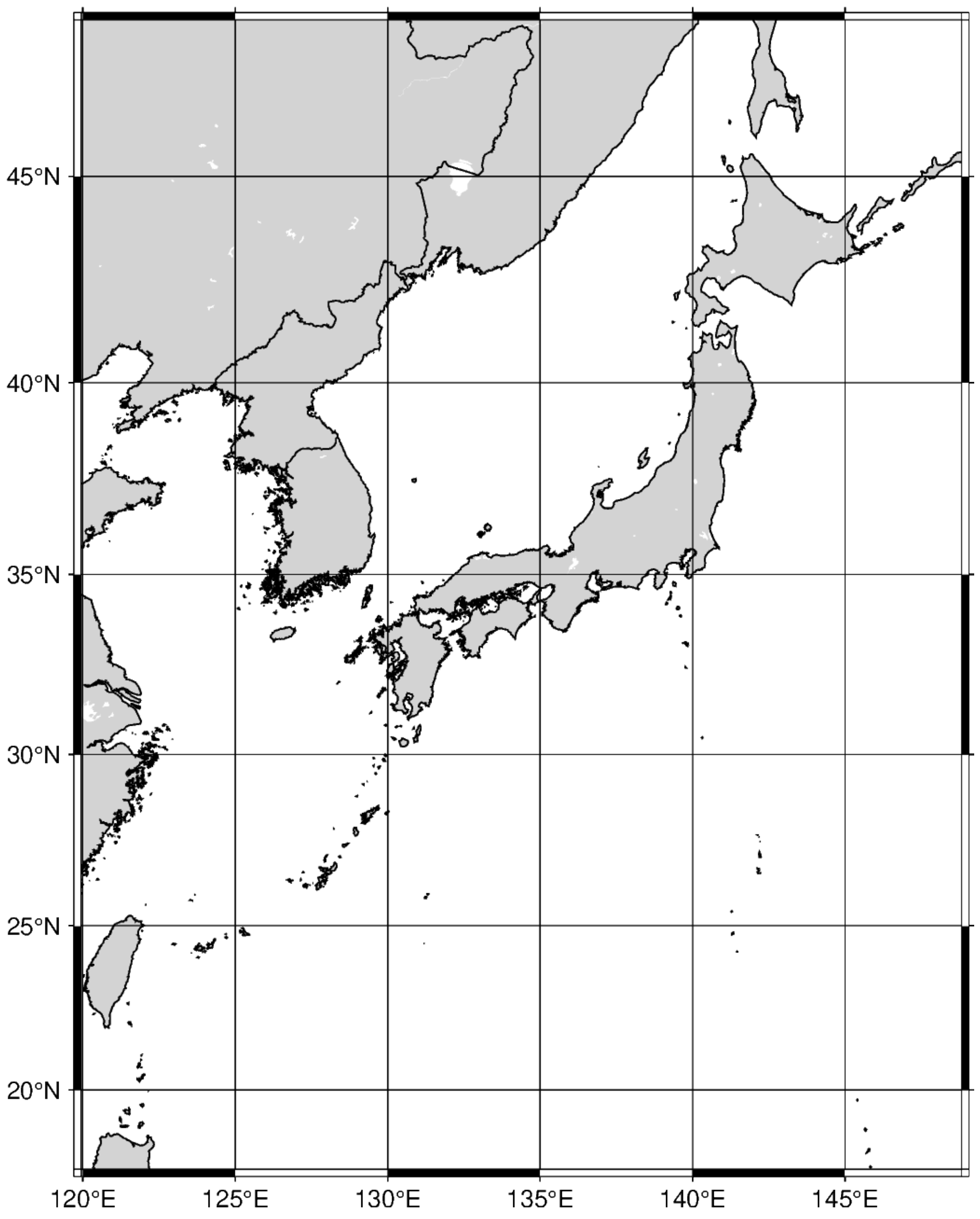


Out:

```
<IPython.core.display.Image object>
```

The `region` increment can be appended with **+R**, which adds the increment without rounding.

```
fig = pygmt.Figure()
fig.coast(
    # Expand the region setting outside the range of Japan by 3 degrees in all
    # directions, without rounding to the nearest increment.
    region="JP+R3",
    projection="M12c",
    land="lightgray",
    water="white",
    borders="1/0.5p",
    shorelines="1/0.5p",
    frame="ag",
)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

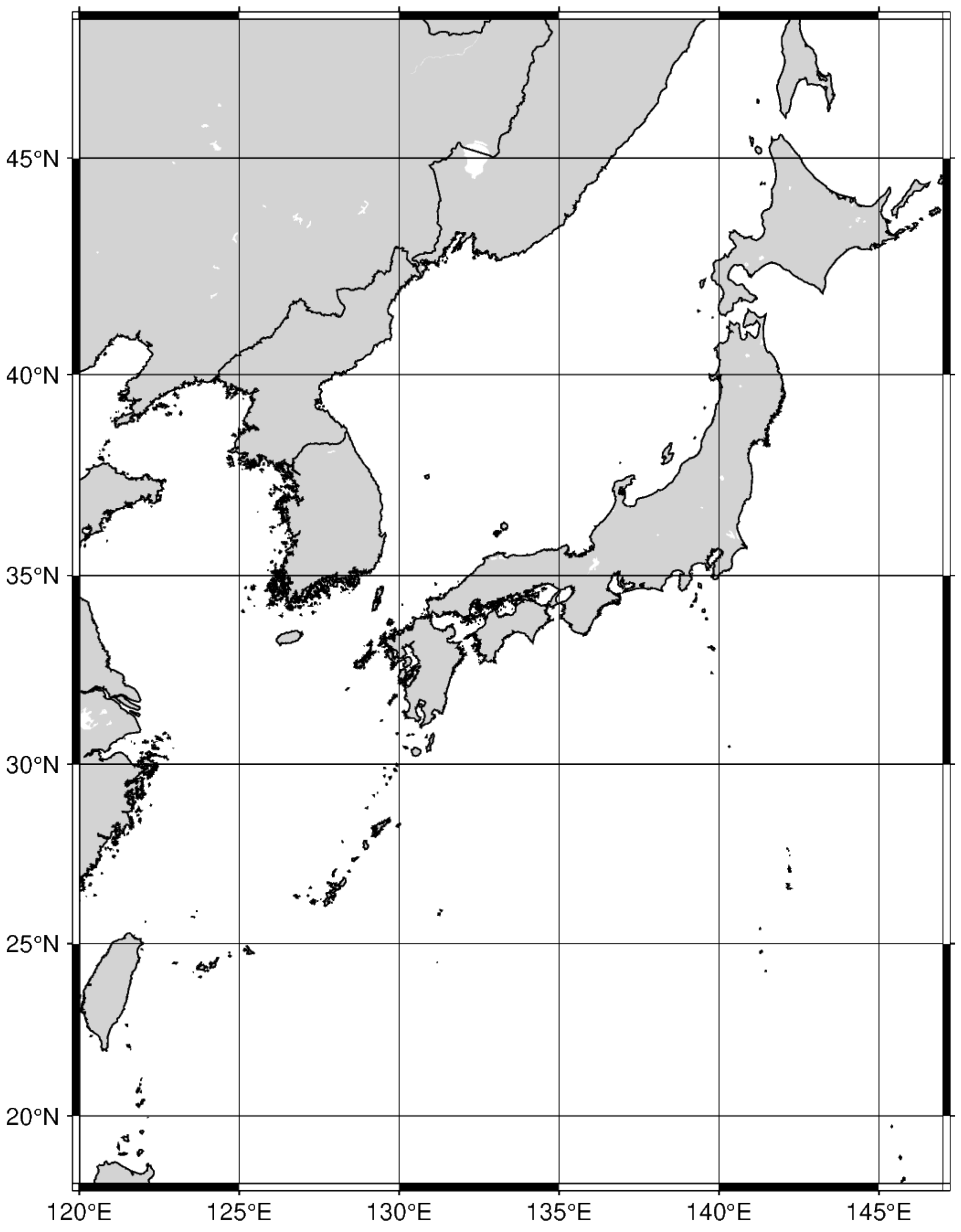
The `region` increment can be appended with **+e**, which is like **+r** and expands the final region boundaries to be multiples of *increment*. However, it ensures that the bounding box extends by at least 0.25 times the increment.

```
fig = pygmt.Figure()
fig.coast(
    # Expand the region boundaries to be multiples of 3 degrees in all
    # directions
    region="JP+e3",
    projection="M12c",
    land="lightgray",
```

(continues on next page)

(continued from previous page)

```
    water="white",
    borders="1/0.5p",
    shorelines="1/0.5p",
    frame="ag",
)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

Total running time of the script: (0 minutes 33.171 seconds)

9.10 Plotting data points

GMT shines when it comes to plotting data on a map. We can use some sample data that is packaged with GMT to try this out. PyGMT provides access to these datasets through the `pygmt.datasets` package. If you don't have the data files already, they are automatically downloaded and saved to a cache directory the first time you use them (usually `~/.gmt/cache`).

```
import pygmt
```

For example, let's load the sample dataset of tsunami generating earthquakes around Japan (*`pygmt.datasets.load_japan_quakes`*). The data is loaded as a `pandas.DataFrame`.

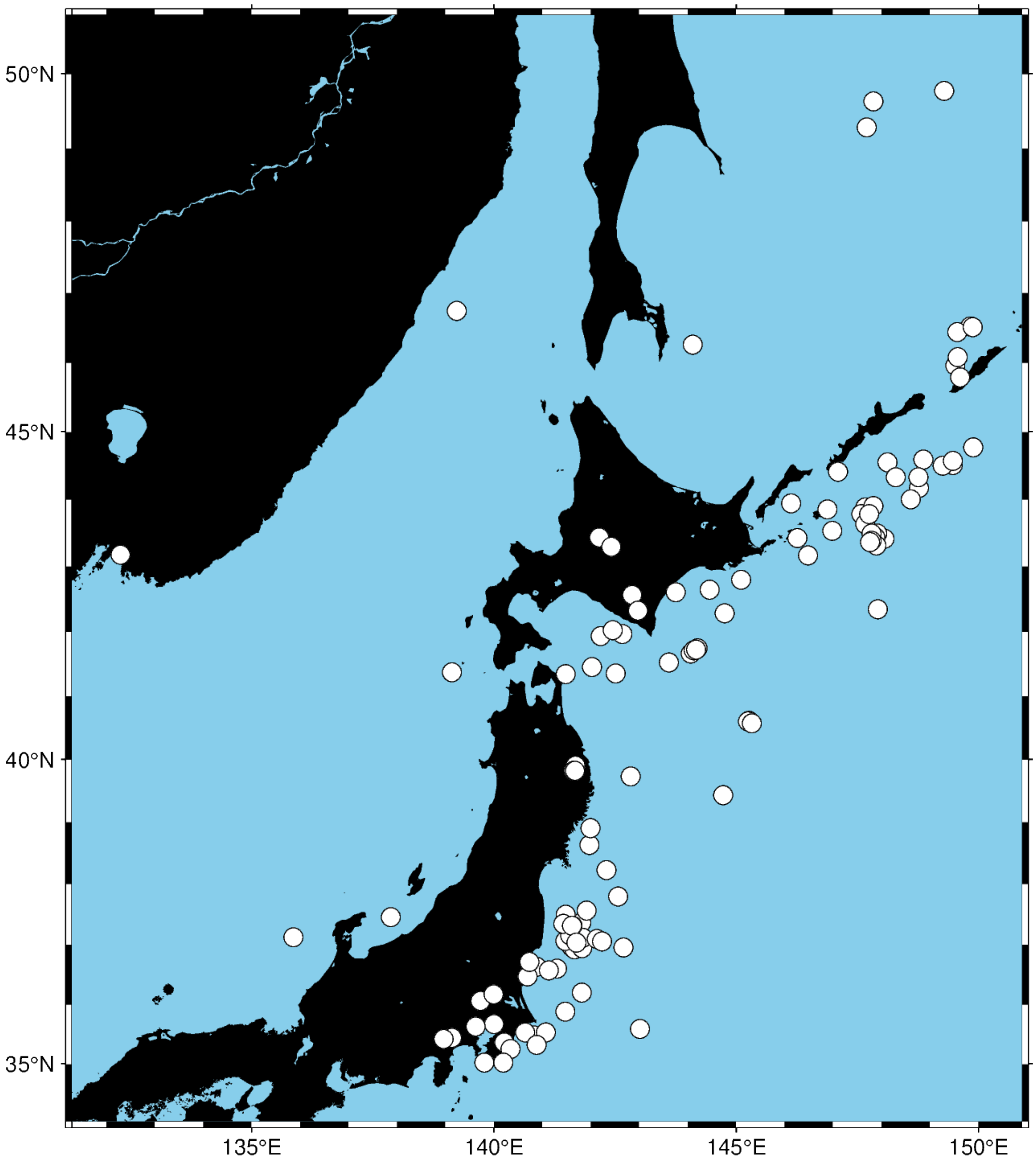
```
data = pygmt.datasets.load_japan_quakes()

# Set the region for the plot to be slightly larger than the data bounds.
region = [
    data.longitude.min() - 1,
    data.longitude.max() + 1,
    data.latitude.min() - 1,
    data.latitude.max() + 1,
]

print(region)
print(data.head())
```

Out:

```
[131.29, 150.89, 34.02, 50.77]
   year  month  day  latitude  longitude  depth_km  magnitude
0  1987      1    4     49.77     149.29       489        4.1
1  1987      1    9     39.90     141.68        67        6.8
2  1987      1    9     39.82     141.64        84        4.0
3  1987      1   14     42.56     142.85       102        6.5
4  1987      1   16     42.79     145.10        54        5.1
```

We'll use the *`pygmt.Figure.plot`* method to plot circles on the earthquake epicenters.

```
fig = pygmt.Figure()
fig.basemap(region=region, projection="M15c", frame=True)
fig.coast(land="black", water="skyblue")
fig.plot(x=data.longitude, y=data.latitude, style="c0.3c", color="white", pen="black")
fig.show()
```



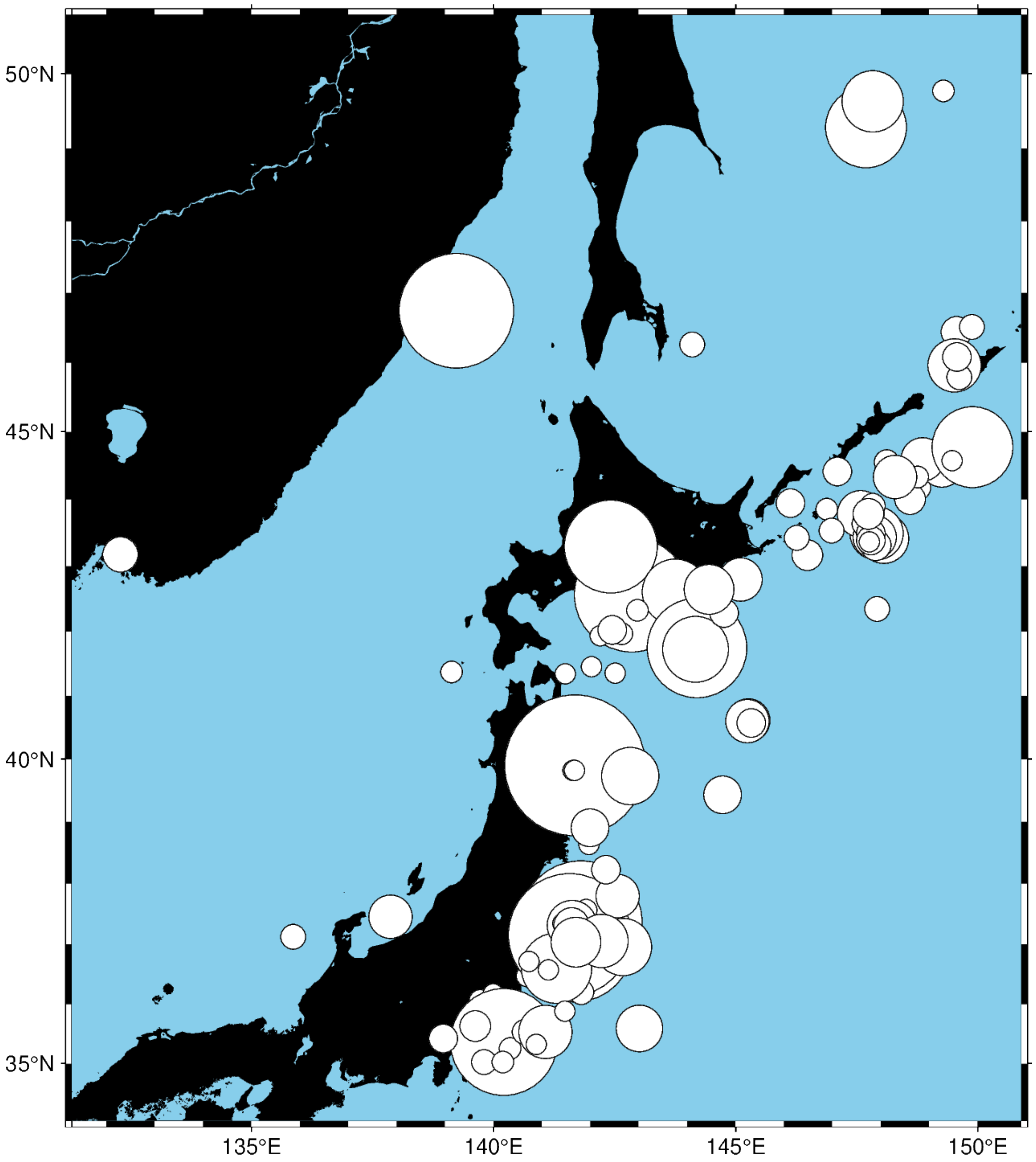
Out:

```
<IPython.core.display.Image object>
```

We used the style `c0.3c` which means "circles of 0.3 centimeter size". The `pen` parameter controls the outline of the symbols and the `color` parameter controls the fill.

We can map the size of the circles to the earthquake magnitude by passing an array to the `size` parameter. Because the magnitude is on a logarithmic scale, it helps to show the differences by scaling the values using a power law.

```
fig = pygmt.Figure()
fig.basemap(region=region, projection="M15c", frame=True)
fig.coast(land="black", water="skyblue")
fig.plot(
    x=data.longitude,
    y=data.latitude,
    size=0.02 * (2 ** data.magnitude),
    style="cc",
    color="white",
    pen="black",
)
fig.show()
```



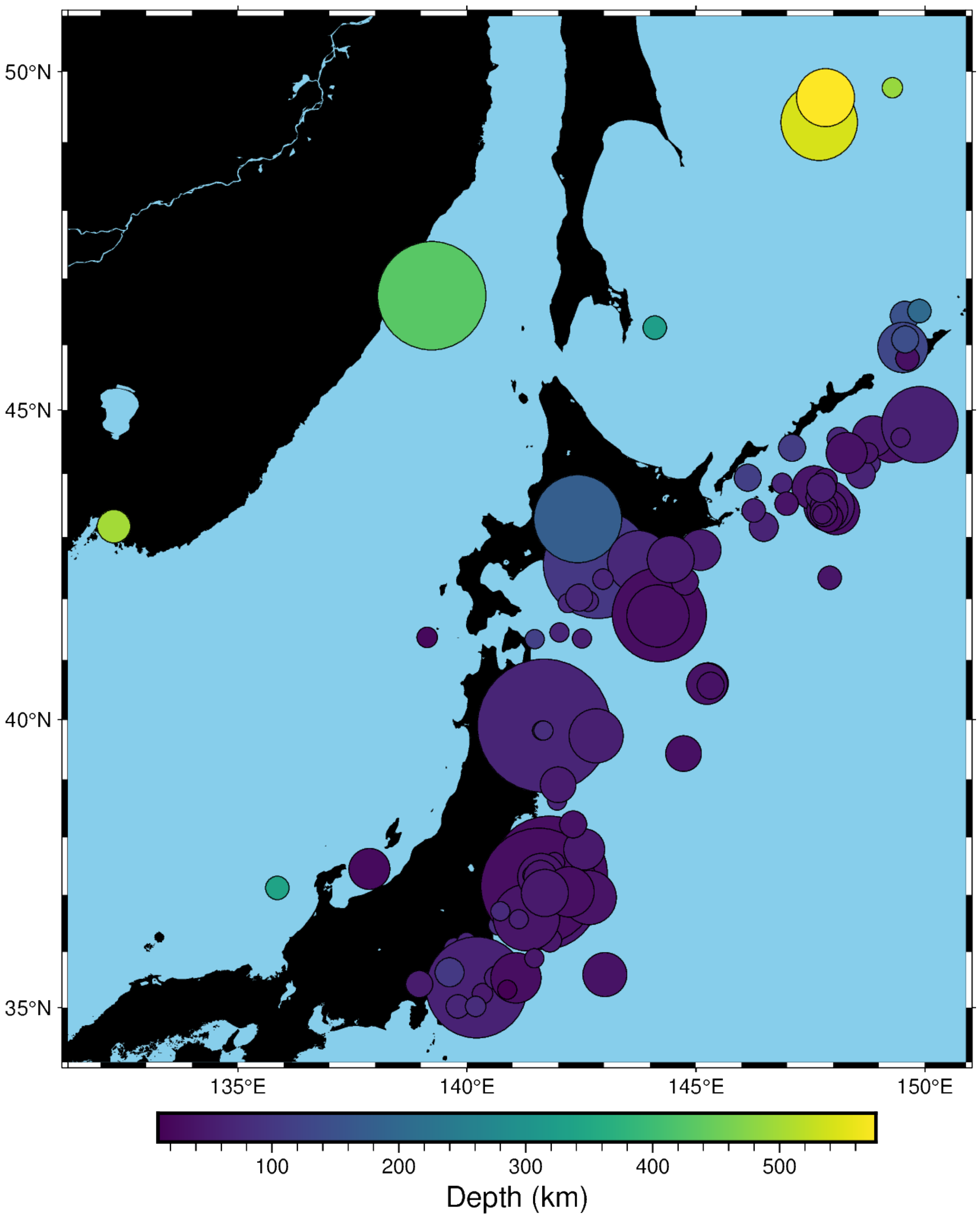
Out:

```
<IPython.core.display.Image object>
```

Notice that we didn't include the size in the `style` parameter this time, just the symbol `c` (circles) and the unit `c` (centimeter). So in this case, the size will be interpreted as being in centimeters.

We can also map the colors of the markers to the depths by passing an array to the `color` parameter and providing a colormap name (`cmap`). We can even use the new matplotlib colormap "viridis". Here, we first create a continuous colormap ranging from the minimum depth to the maximum depth of the earthquakes using *`pygmt.makecpt`*, then set `cmap=True` in *`pygmt.Figure.plot`* to use the colormap. At the end of the plot, we also plot a colorbar showing the colormap used in the plot.

```
fig = pygmt.Figure()
fig.basemap(region=region, projection="M15c", frame=True)
fig.coast(land="black", water="skyblue")
pygmt.makecpt(cmap="viridis", series=[data.depth_km.min(), data.depth_km.max()])
fig.plot(
    x=data.longitude,
    y=data.latitude,
    size=0.02 * 2 ** data.magnitude,
    color=data.depth_km,
    cmap=True,
    style="cc",
    pen="black",
)
fig.colorbar(frame='af+l"Depth (km)"')
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

Total running time of the script: (0 minutes 11.636 seconds)

9.11 Plotting lines

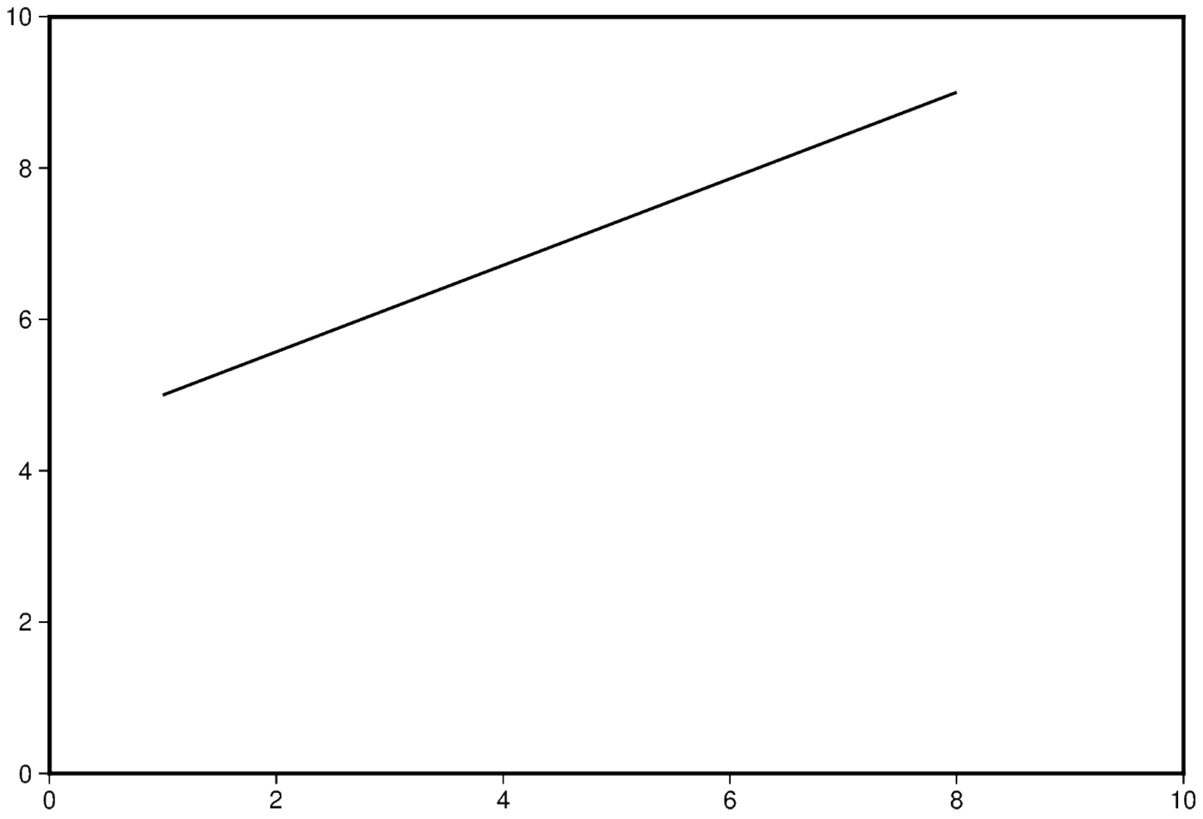
Plotting lines is handled by *pygmt.Figure.plot*.

```
import pygmt
```

9.11.1 Plot lines

Create a Cartesian figure using `projection` parameter and set the axis scales using `region` (in this case, each axis is 0-10). Pass a list of `x` and `y` values to be plotted as a line.

```
fig = pygmt.Figure()
fig.plot(
    region=[0, 10, 0, 10],
    projection="X15c/10c",
    frame="a",
    x=[1, 8],
    y=[5, 9],
    pen="1p,black",
)
fig.show()
```

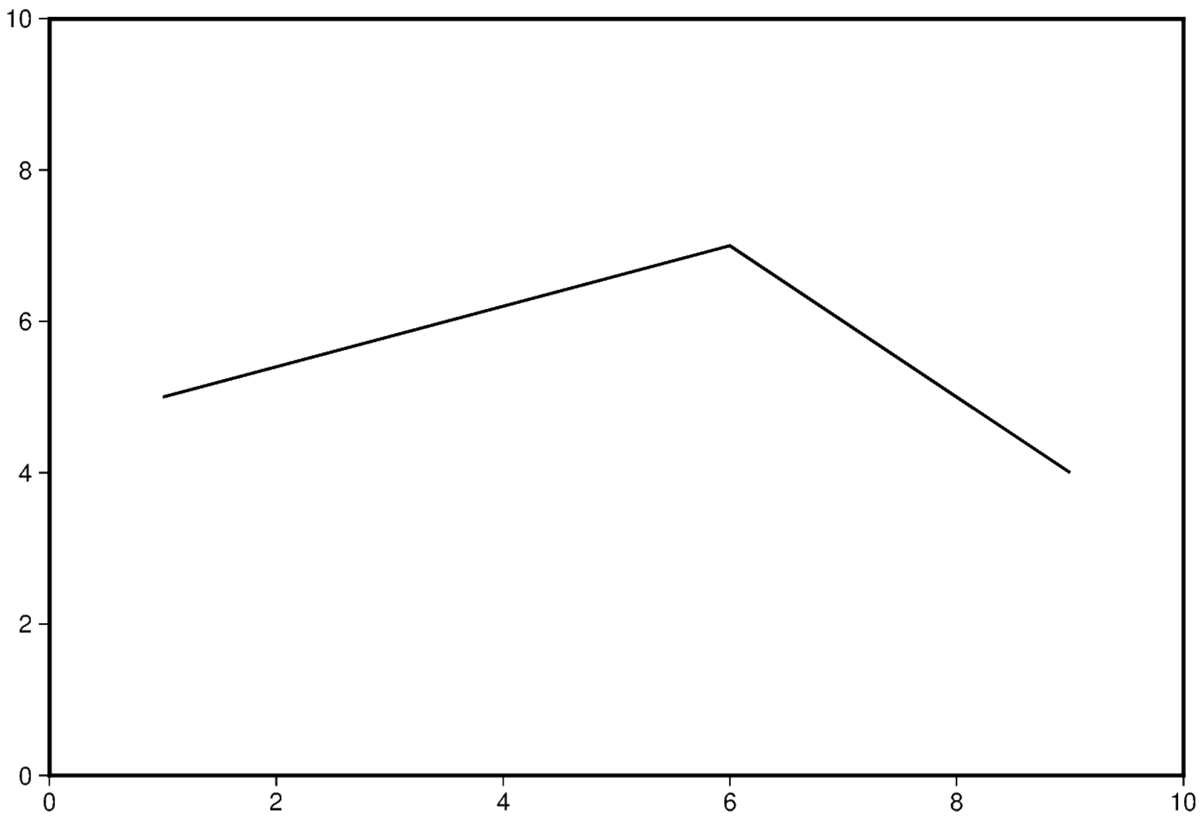


Out:

```
<IPython.core.display.Image object>
```

Additional line segments can be added by including additional values for `x` and `y`.

```
fig = pygmt.Figure()
fig.plot(
    region=[0, 10, 0, 10],
    projection="X15c/10c",
    frame="a",
    x=[1, 6, 9],
    y=[5, 7, 4],
    pen="1p,black",
)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

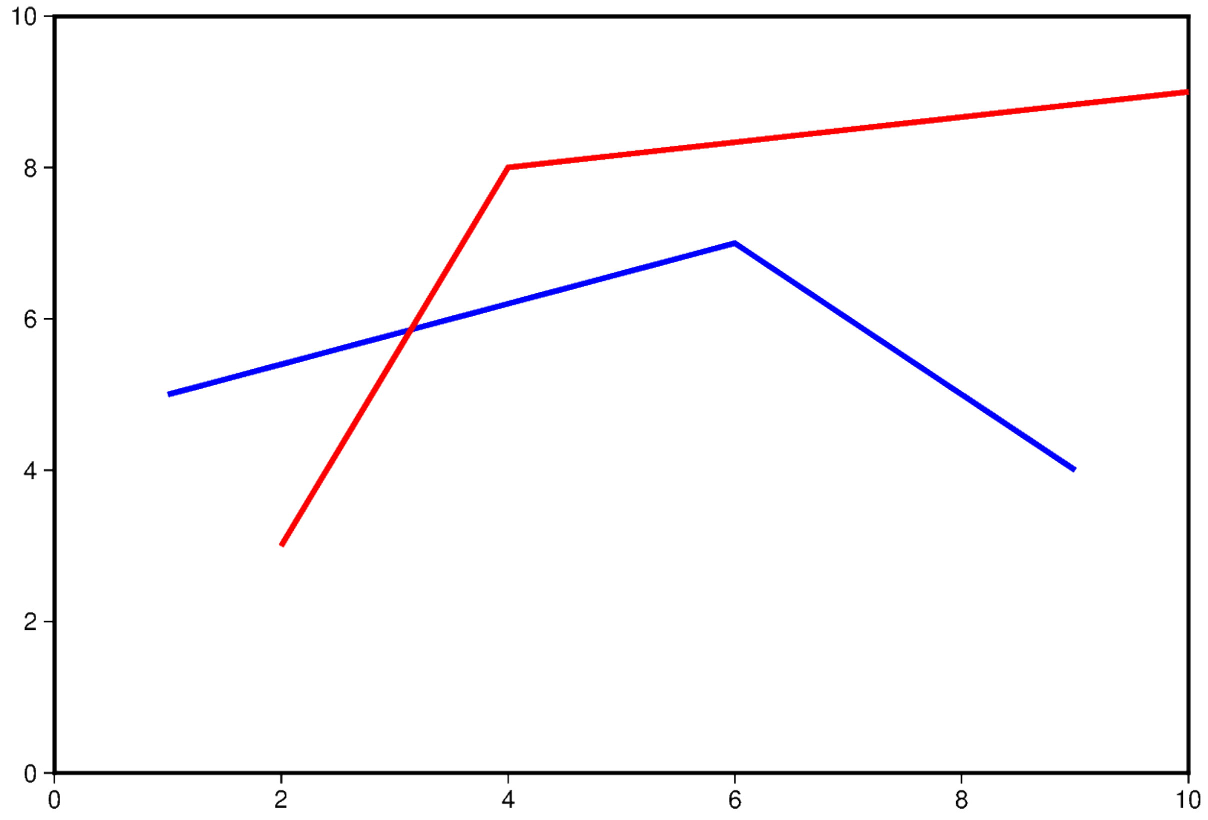
To plot multiple lines, *pygmt.Figure.plot* needs to be used for each additional line. Arguments such as `region`, `projection`, and `frame` do not need to be repeated in subsequent uses.

```
fig = pygmt.Figure()
fig.plot(
    region=[0, 10, 0, 10],
    projection="X15c/10c",
    frame="a",
    x=[1, 6, 9],
    y=[5, 7, 4],
    pen="2p,blue",
)
```

(continues on next page)

(continued from previous page)

```
fig.plot(x=[2, 4, 10], y=[3, 8, 9], pen="2p,red")
fig.show()
```



Out:

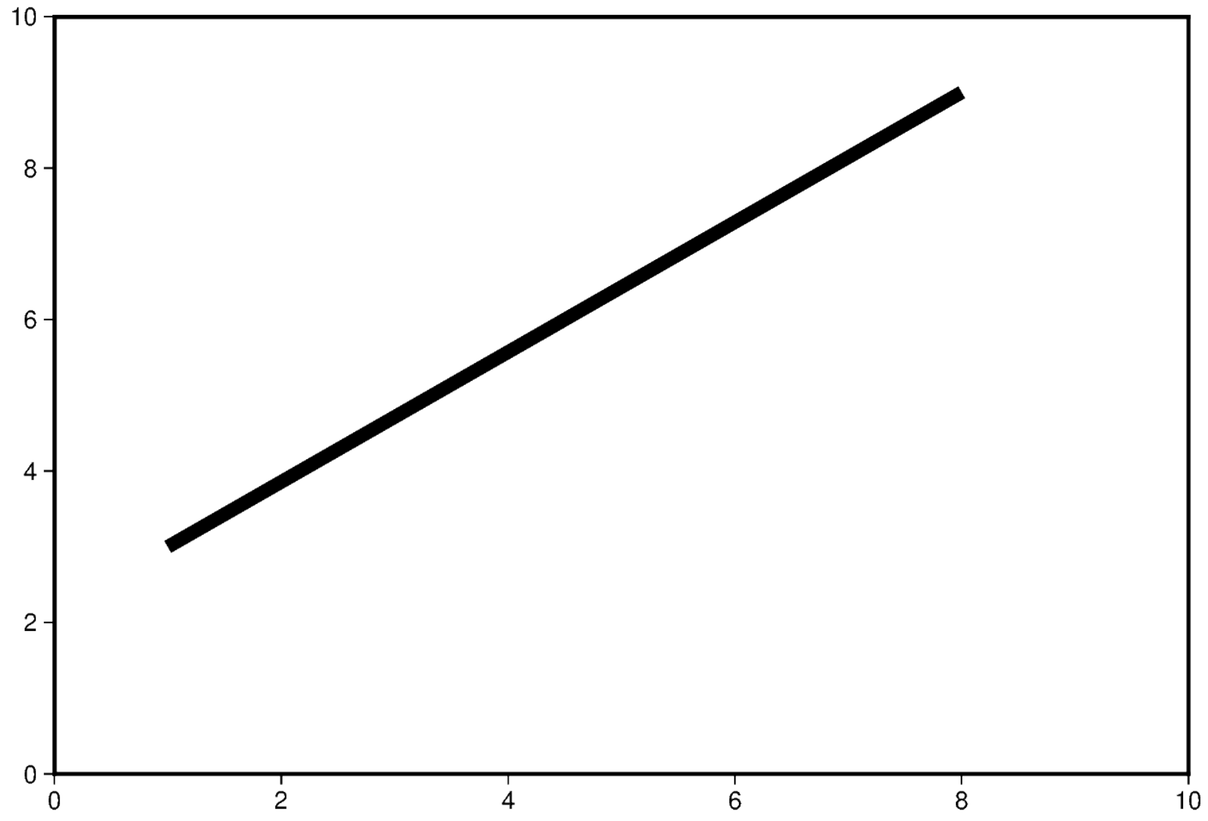
```
<IPython.core.display.Image object>
```

9.11.2 Change line attributes

The line attributes can be set by the `pen` parameter. `pen` takes a string argument with the optional values *width*,*color*,*style*.

In the example below, the pen width is set to `5p`, and with `black` as the default color and `solid` as the default style.

```
fig = pygmt.Figure()
fig.plot(
    region=[0, 10, 0, 10],
    projection="X15c/10c",
    frame="a",
    x=[1, 8],
    y=[3, 9],
    pen="5p",
)
fig.show()
```

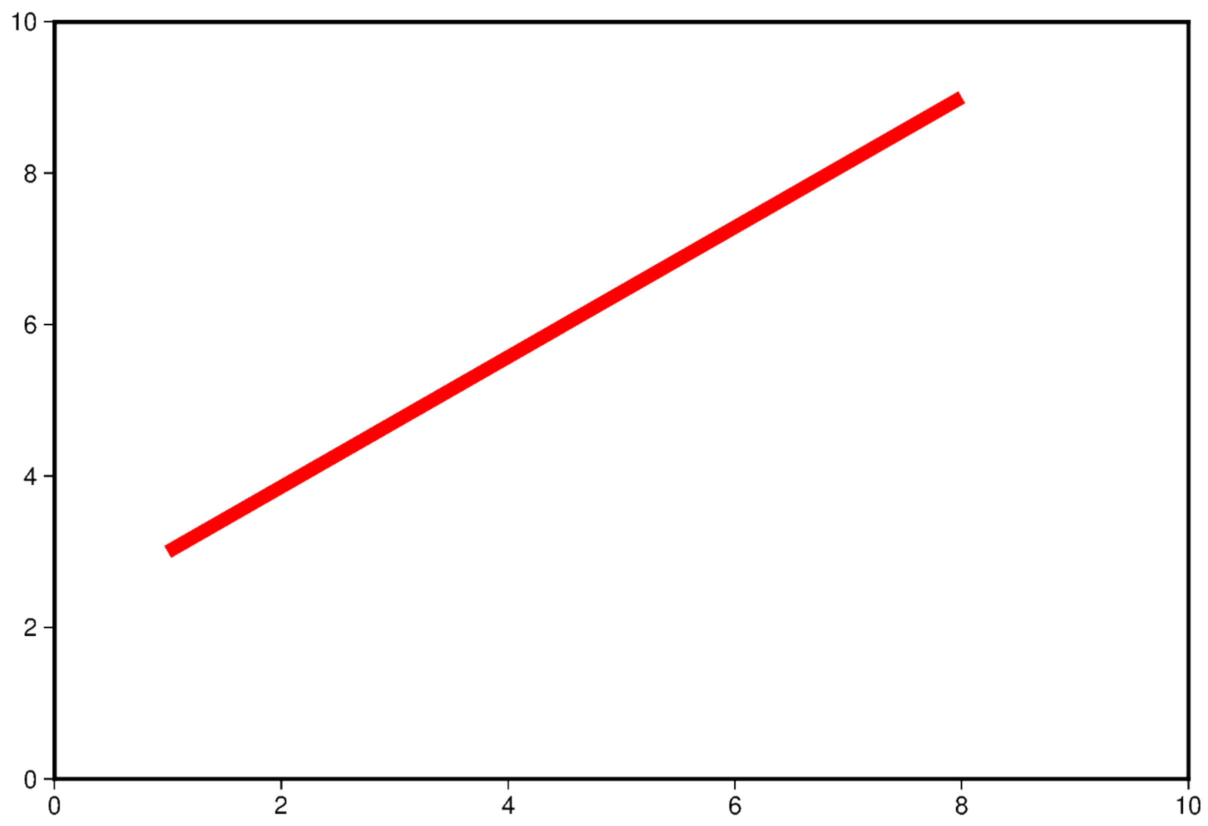


Out:

```
<IPython.core.display.Image object>
```

The line color can be set and is added after the line width to the `pen` parameter. In the example below, the line color is set to `red`.

```
fig = pygmt.Figure()
fig.plot(
    region=[0, 10, 0, 10],
    projection="X15c/10c",
    frame="a",
    x=[1, 8],
    y=[3, 9],
    pen="5p,red",
)
fig.show()
```

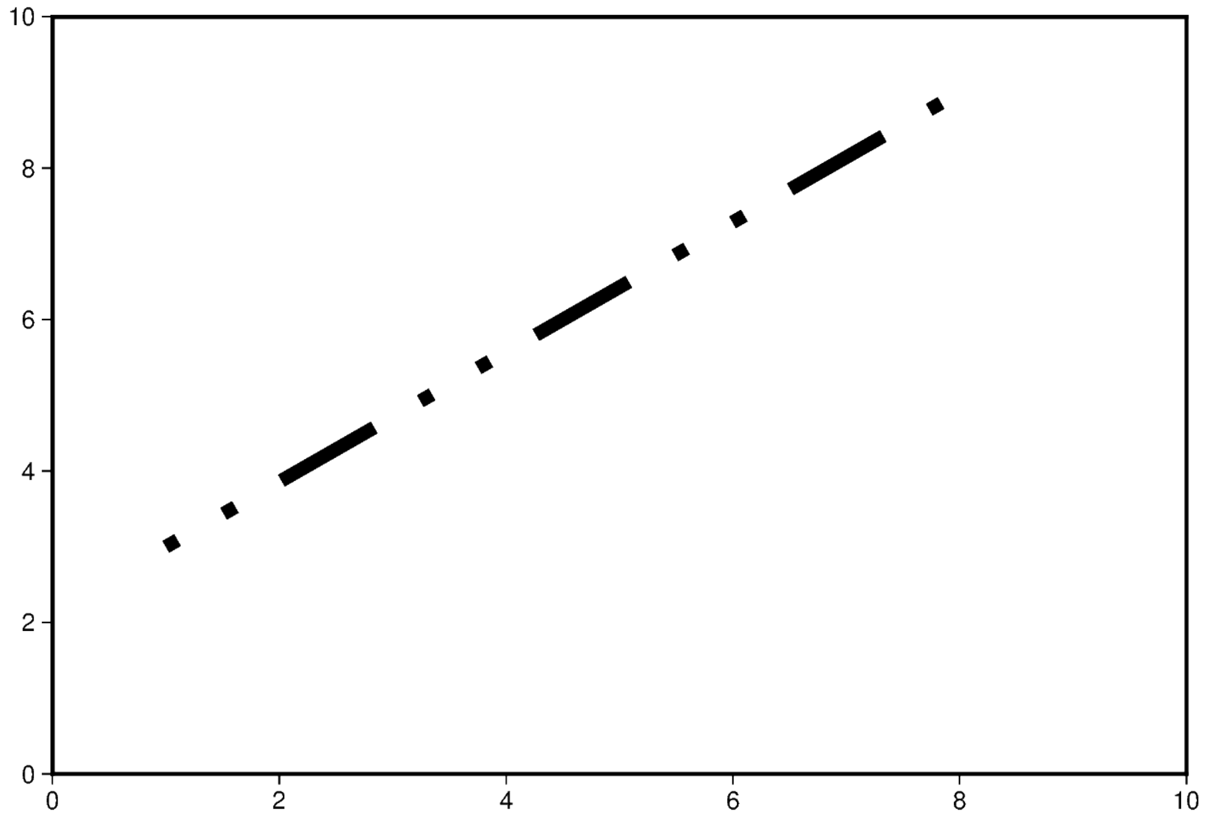


Out:

```
<IPython.core.display.Image object>
```

The line style can be set and is added after the line width or color to the `pen` parameter. In the example below, the line style is set to `..-` (*dot dot dash*), and the default color `black` is used.

```
fig = pygmt.Figure()
fig.plot(
    region=[0, 10, 0, 10],
    projection="X15c/10c",
    frame="a",
    x=[1, 8],
    y=[3, 9],
    pen="5p,..-",
)
fig.show()
```



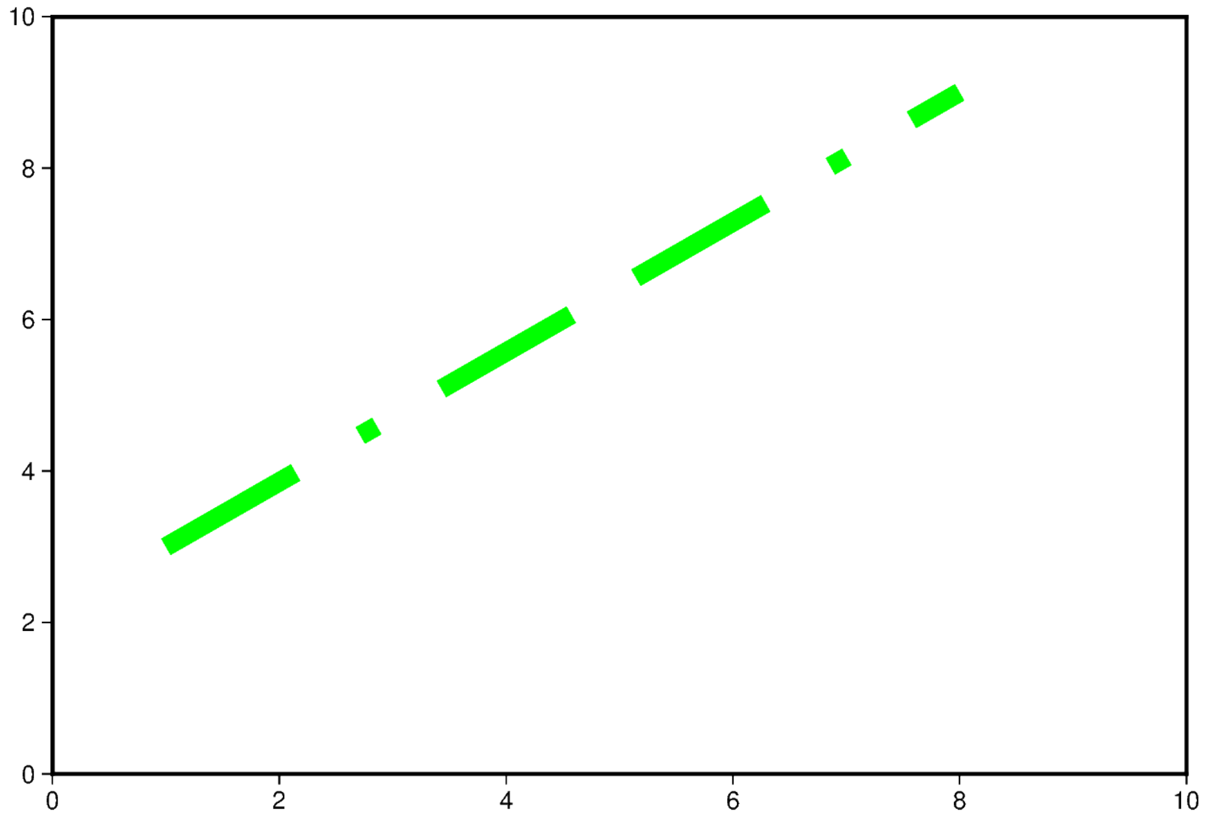
Out:

```
<IPython.core.display.Image object>
```

The line width, color, and style can all be set in the same `pen` parameter. In the example below, the line width is set to `7p`, the color is set to `green`, and the line style is `-.-` (*dash dot dash*).

For a gallery showing other `pen` settings, see *Line styles*.

```
fig = pygmt.Figure()
fig.plot(
    region=[0, 10, 0, 10],
    projection="X15c/10c",
    frame="a",
    x=[1, 8],
    y=[3, 9],
    pen="7p,green,-.-",
)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

Total running time of the script: (0 minutes 14.996 seconds)

9.12 Plotting vectors

Plotting vectors is handled by *pygmt.Figure.plot*.

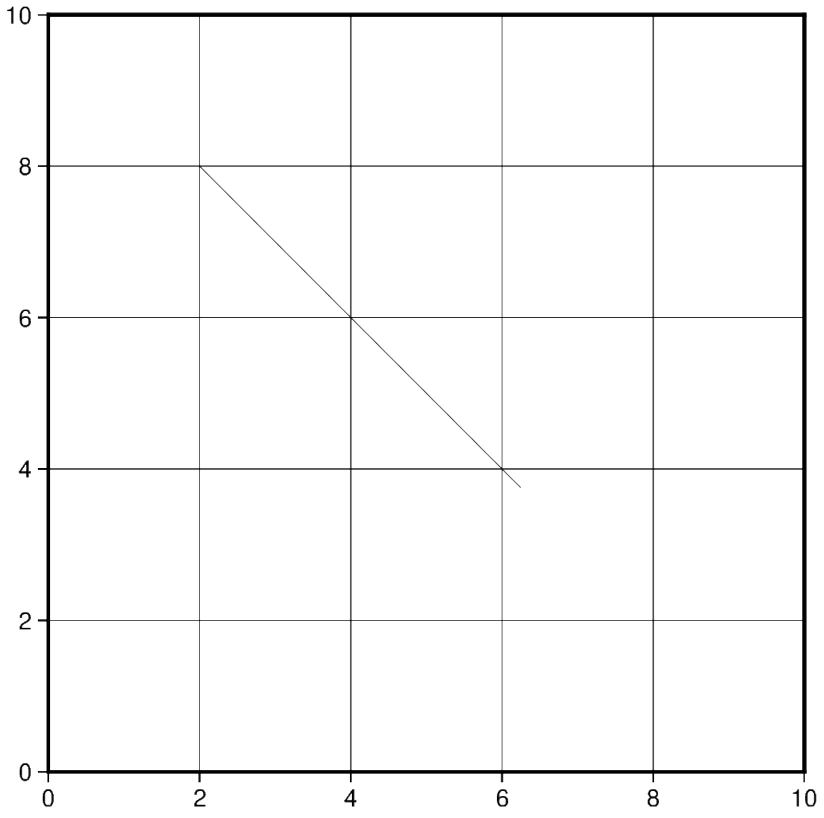
```
import numpy as np
import pygmt
```

9.12.1 Plot Cartesian Vectors

Create a simple Cartesian vector using a starting point through `x`, `y`, and `direction` parameters. On the shown figure, the plot is projected on a 10cm X 10cm region, which is specified by the `projection` parameter. The direction is specified by a list of two 1d arrays structured as `[[angle_in_degrees], [length]]`. The angle is measured in degrees and moves counter-clockwise from the horizontal. The length of the vector uses centimeters by default but could be changed using *pygmt.config* (Check the next examples for unit changes).

Notice that the `v` in the `style` parameter stands for vector; it distinguishes it from regular lines and allows for different customization. `0c` is used to specify the size of the arrow head which explains why there is no arrow on either side of the vector.

```
fig = pygmt.Figure()
fig.plot(
    region=[0, 10, 0, 10],
    projection="X10c/10c",
    frame="ag",
    x=2,
    y=8,
    style="v0c",
    direction=[[-45], [6]],
)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

In this example, we apply the same concept shown previously to plot multiple vectors. Notice that instead of passing int/float to `x` and `y`, a list of all x and y coordinates will be passed. Similarly, the length of direction list will increase accordingly.

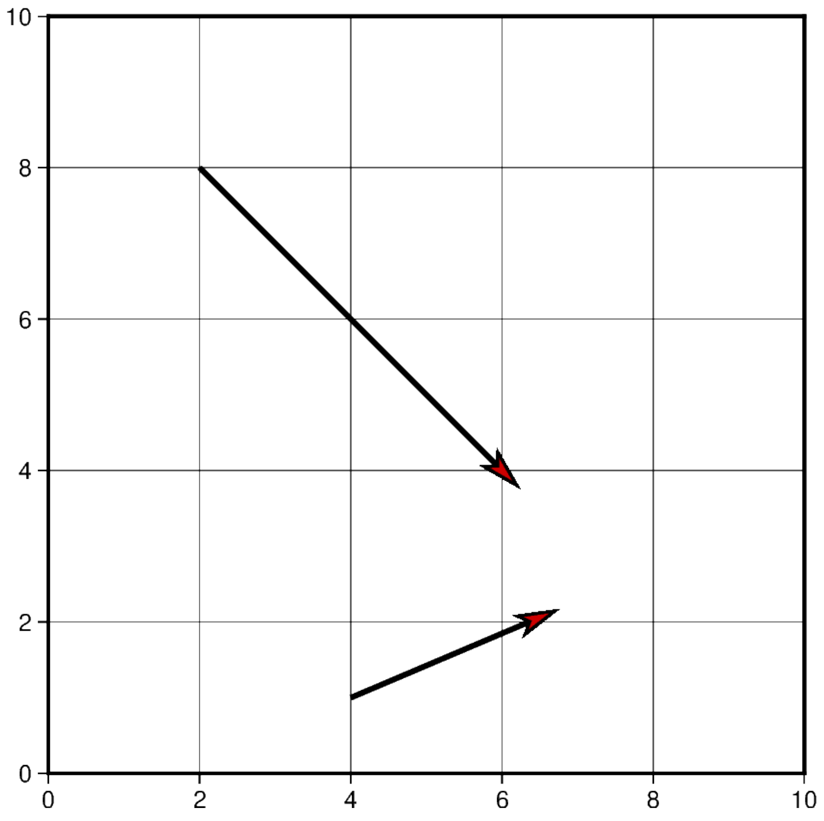
Additionally, we change the style of the vector to include a red arrow head at the end (**+e**) of the vector and increase the thickness (`pen="2p"`) of the vector stem. A list of different styling attributes can be found in *Vector heads and tails*.

```
fig = pygmt.Figure()
fig.plot(
    region=[0, 10, 0, 10],
    projection="X10c/10c",
    frame="ag",
```

(continues on next page)

(continued from previous page)

```
    x=[2, 4],
    y=[8, 1],
    style="v0.6c+e",
    direction=[[-45, 23], [6, 3]],
    pen="2p",
    color="red3",
)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

The default unit of vector length is centimeters, however, this can be changed to inches or points. Note that, in PyGMT, one point is defined as 1/72 inch.

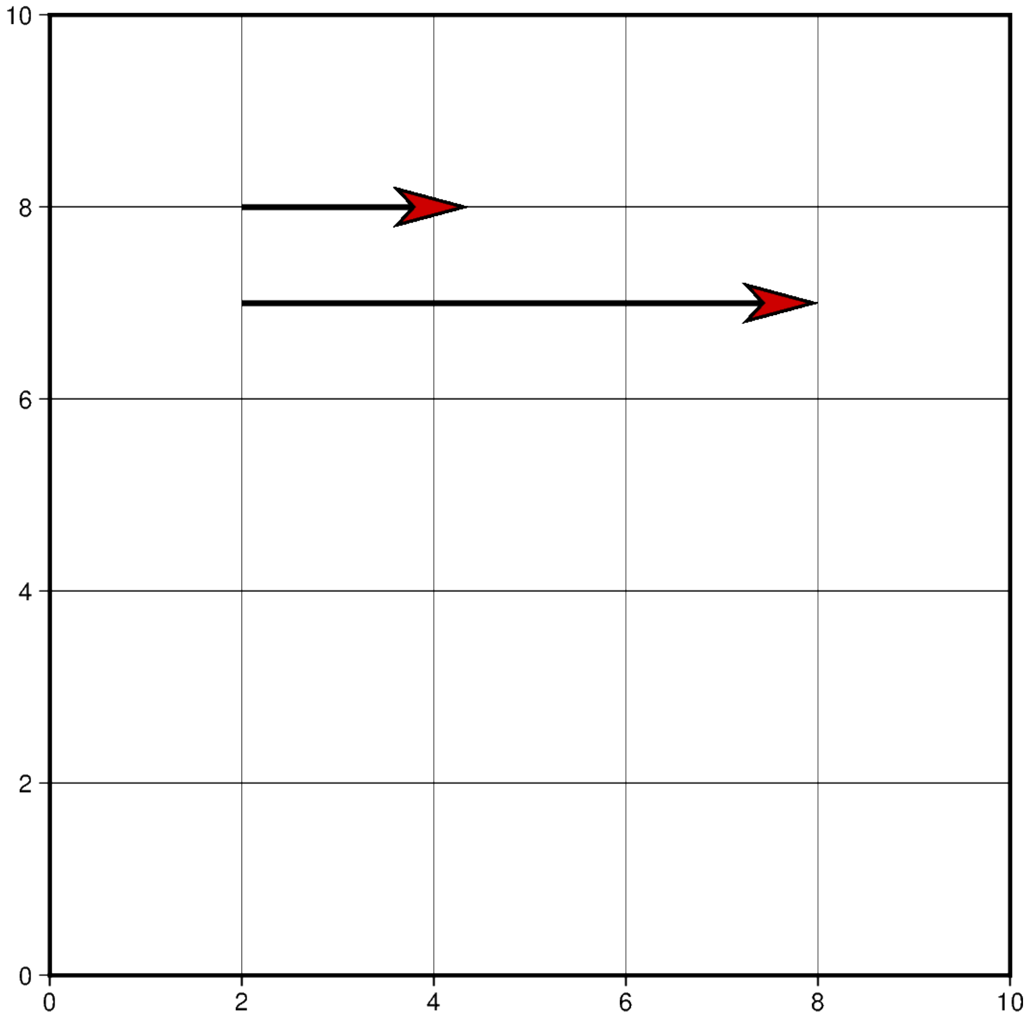
In this example, the graphed region is 5in X 5in, but the length of the first vector is still graphed in centimeters. Using `pygmt.config(PROJ_LENGTH_UNIT="i")`, the default unit can be changed to inches in the second plotted vector.

```
fig = pygmt.Figure()
# Vector 1 with default unit as cm
fig.plot(
    region=[0, 10, 0, 10],
    projection="X5i/5i",
    frame="ag",
    x=2,
    y=8,
```

(continues on next page)

(continued from previous page)

```
    style="v1c+e",
    direction=[[0], [3]],
    pen="2p",
    color="red3",
)
# Vector 2 after changing default unit to inch
with pygmt.config(PROJ_LENGTH_UNIT="i"):
    fig.plot(
        x=2,
        y=7,
        direction=[[0], [3]],
        style="v1c+e",
        pen="2p",
        color="red3",
    )
fig.show()
```



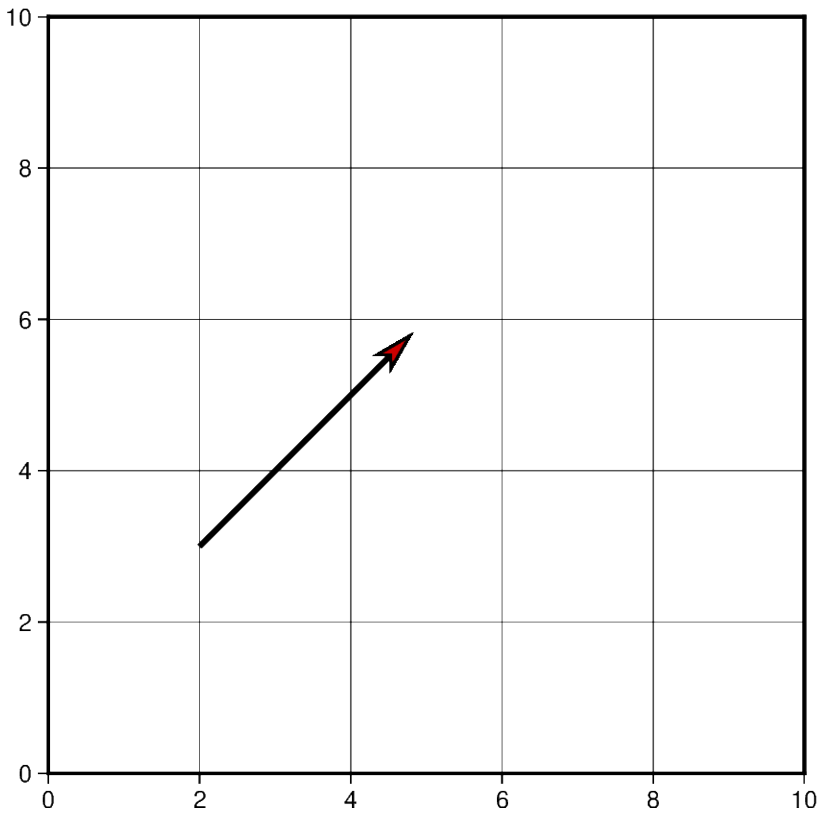
Out:

```
<IPython.core.display.Image object>
```

Vectors can also be plotted by including all the information about a vector in a single list. However, this requires creating a 2D list or numpy array containing all vectors. Each vector list contains the information structured as: `[x_start, y_start, direction_degrees, length]`.

If this approach is chosen, the `data` parameter must be used instead of `x`, `y` and `direction`.

```
# Create a list of lists that include each vector information
vectors = [[2, 3, 45, 4]]

fig = pygmt.Figure()
fig.plot(
    region=[0, 10, 0, 10],
    projection="X10c/10c",
    frame="ag",
    data=vectors,
    style="v0.6c+e",
    pen="2p",
    color="red3",
)
fig.show()
```

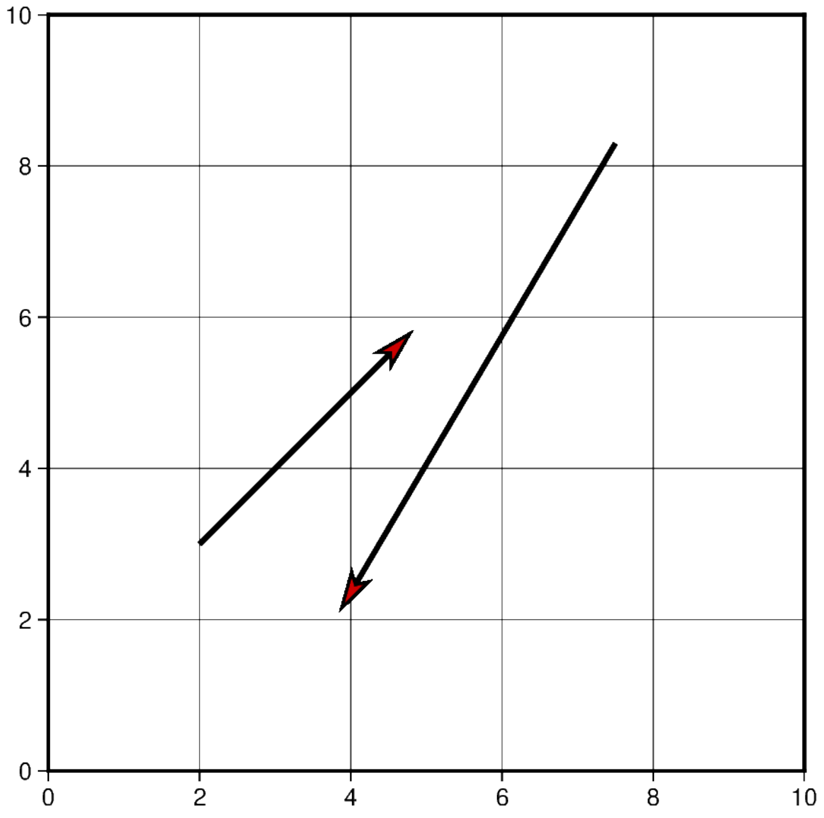


Out:

```
<IPython.core.display.Image object>
```

Using the functionality mentioned in the previous example, multiple vectors can be plotted at the same time. Another vector could be simply added to the 2D list or numpy array object and passed using `data` parameter.

```
# Vector specifications structured as:
# [x_start, y_start, direction_degrees, length]
vector_1 = [2, 3, 45, 4]
vector_2 = [7.5, 8.3, -120.5, 7.2]
# Create a list of lists that include each vector information
vectors = [vector_1, vector_2]
# Vectors structure: [[2, 3, 45, 4], [7.5, 8.3, -120.5, 7.2]]

fig = pygmt.Figure()
fig.plot(
    region=[0, 10, 0, 10],
    projection="X10c/10c",
    frame="ag",
    data=vectors,
    style="v0.6c+e",
    pen="2p",
    color="red3",
)
fig.show()
```



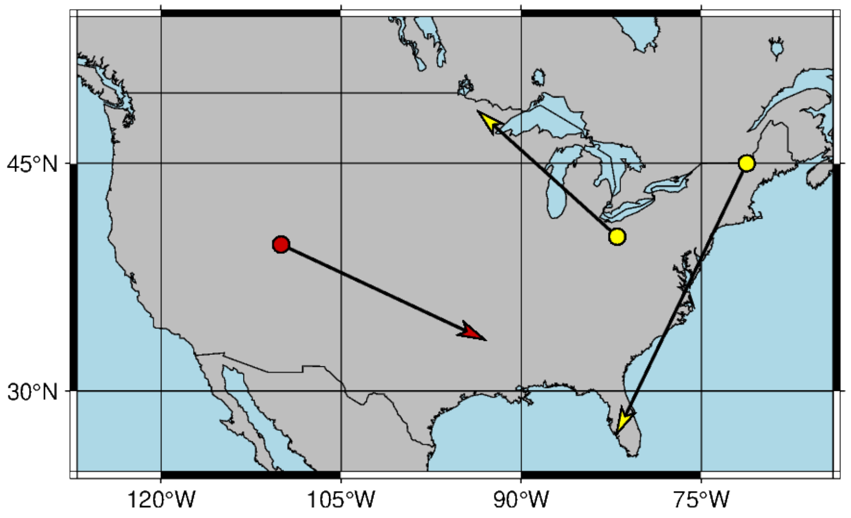
Out:

```
<IPython.core.display.Image object>
```

In this example, cartesian vectors are plotted over a Mercator projection of the continental US. The x values represent the longitude and y values represent the latitude where the vector starts.

This example also shows some of the styles a vector supports. The beginning point of the vector (**+b**) should take the shape of a circle (**c**). Similarly, the end point of the vector (**+e**) should have an arrow shape (**a**) (to draw a plain arrow, use **A** instead). Lastly, the **+a** specifies the angle of the vector head apex (30 degrees in this example).

```
# Create a plot with coast, Mercator projection (M) over the continental US
fig = pygmt.Figure()
fig.coast(
    region=[-127, -64, 24, 53],
    projection="M10c",
    frame="ag",
    borders=1,
    shorelines="0.25p,black",
    area_thresh=4000,
    land="grey",
    water="lightblue",
)

# Plot a vector using the x, y, direction parameters
style = "v0.4c+bc+ea+a30"
fig.plot(
    x=-110,
    y=40,
    style=style,
    direction=[[-25], [3]],
    pen="1p",
    color="red3",
)

# vector specifications structured as:
# [x_start, y_start, direction_degrees, length]
vector_2 = [-82, 40.5, 138, 2.5]
vector_3 = [-71.2, 45, -115.7, 4]
# Create a list of lists that include each vector information
vectors = [vector_2, vector_3]

# Plot vectors using the data parameter.
fig.plot(
    data=vectors,
    style=style,
    pen="1p",
    color="yellow",
)
fig.show()
```

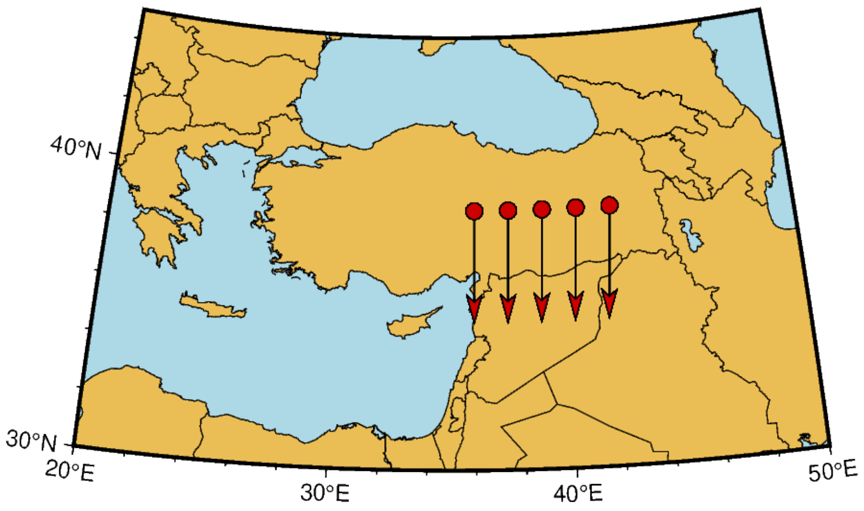


Out:

```
<IPython.core.display.Image object>
```

Another example of plotting cartesian vectors over a coast plot. This time a Transverse Mercator projection is used. Additionally, `numpy.linspace` is used to create 5 vectors with equal stops.

```
x = np.linspace(36, 42, 5)  # x values = [36.   37.5 39.   40.5 42. ]
y = np.linspace(39, 39, 5)  # y values = [39. 39. 39. 39.]
direction = np.linspace(-90, -90, 5)  # direction values = [-90. -90. -90. -90.]
length = np.linspace(1.5, 1.5, 5)  # length values = [1.5 1.5 1.5 1.5]

# Create a plot with coast, Mercator projection (M) over the continental US
fig = pygmt.Figure()
fig.coast(
    region=[20, 50, 30, 45],
    projection="T35/10c",
    frame=True,
    borders=1,
    shorelines="0.25p,black",
    area_thresh=4000,
    land="lightbrown",
    water="lightblue",
)

fig.plot(
    x=x,
    y=y,
    style="v0.4c+ea+bc",
    direction=[direction, length],
    pen="0.6p",
    color="red3",
)

fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

9.12.2 Plot Circular Vectors

When plotting circular vectors, all of the information for a single vector is to be stored in a list. Each circular vector list is structured as: `[x_start, y_start, radius, degree_start, degree_stop]`. The first two values in the vector list represent the origin of the circle that will be plotted. The next value is the radius which is represented on the plot in cm.

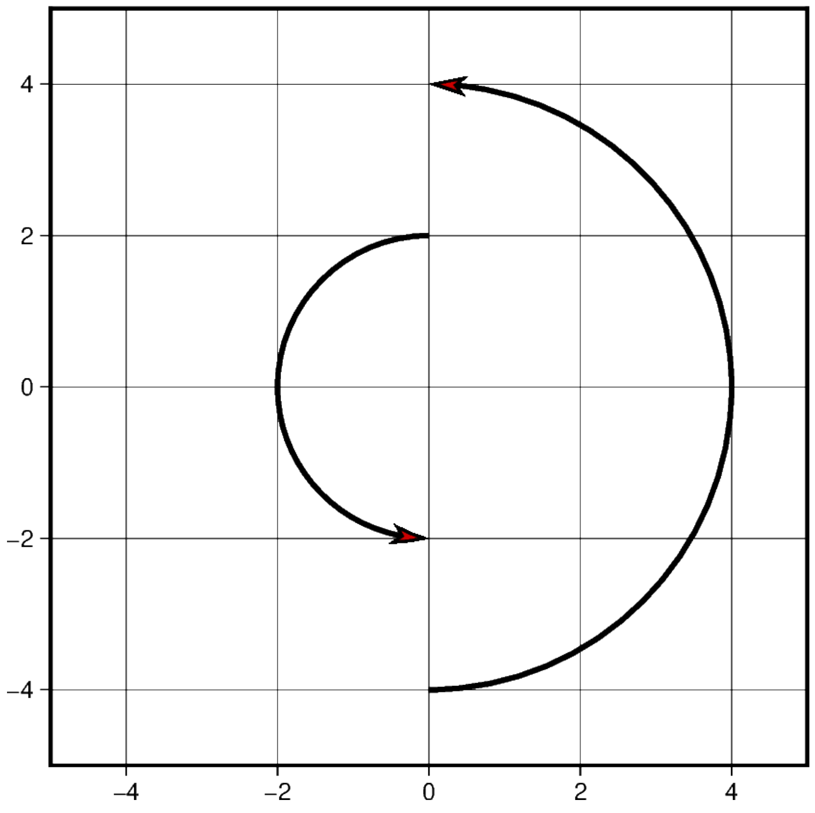
The last two values in the vector list represent the degree at which the plot will start and stop. These values are measured counter-clockwise from the horizontal axis. In this example, the result show is the left half of a circle as the plot starts at 90 degrees and goes until 270. Notice that the `m` in the `style` parameter stands for circular vectors.

```
fig = pygmt.Figure()

circular_vector_1 = [0, 0, 2, 90, 270]
data = [circular_vector_1]
fig.plot(
    region=[-5, 5, -5, 5],
    projection="X10c",
    frame="ag",
    data=data,
    style="m0.5c+ea",
    pen="2p",
    color="red3",
)

# Another example using np.array()
circular_vector_2 = [0, 0, 4, -90, 90]
data = np.array([circular_vector_2])

fig.plot(
    data=data,
    style="m0.5c+ea",
```

(continues on next page)

(continued from previous page)

```
    pen="2p",
    color="red3",
)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

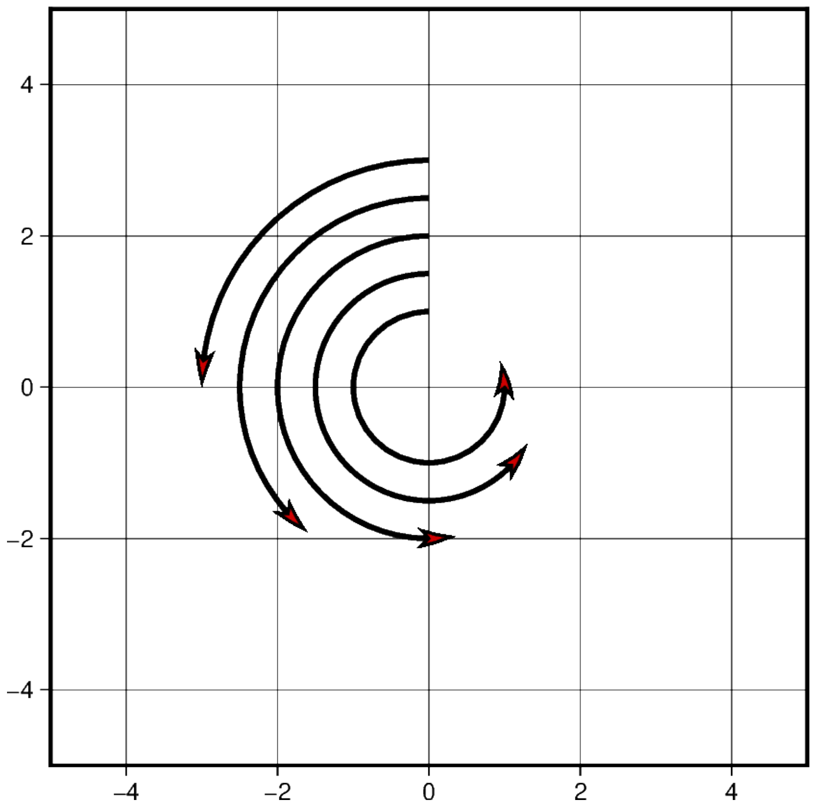
When plotting multiple circular vectors, a two dimensional array or numpy array object should be passed as the `data` parameter. In this example, `numpy.column_stack` is used to generate this two dimensional array. Other numpy objects are used to generate linear values for the `radius` parameter and random values for the `degree_stop` parameter discussed in the previous example. This is the reason in which each vector has a different appearance on the projection.

```
vector_num = 5
radius = 3 - (0.5 * np.arange(0, vector_num))
startdir = np.full(vector_num, 90)
stopdir = 180 + (50 * np.arange(0, vector_num))
data = np.column_stack(
    [np.full(vector_num, 0), np.full(vector_num, 0), radius, startdir, stopdir]
)

fig = pygmt.Figure()
fig.plot(
    region=[-5, 5, -5, 5],
    projection="X10c",
    frame="ag",
```

(continues on next page)

(continued from previous page)

```
    data=data,
    style="m0.5c+ea",
    pen="2p",
    color="red3",
)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

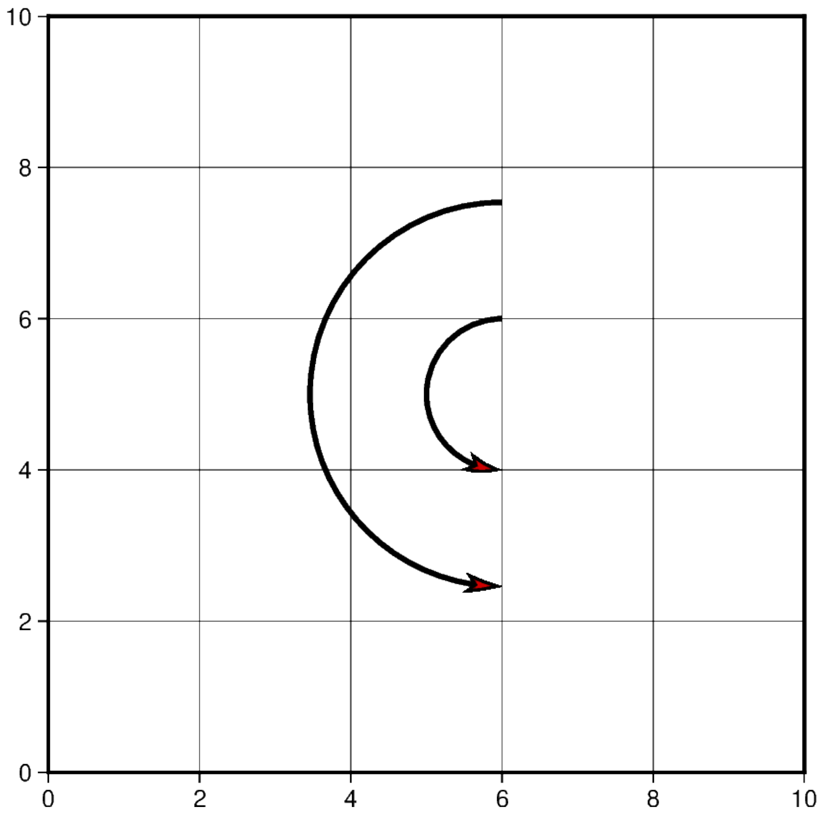
Much like when plotting Cartesian vectors, the default unit used is centimeters. When this is changed to inches, the size of the plot appears larger when the projection units do not change. Below is an example of two circular vectors. One is plotted using the default unit, and the second is plotted using inches. Despite using the same list to plot the vectors, a different measurement unit causes one to be larger than the other.

```
circular_vector = [6, 5, 1, 90, 270]

fig = pygmt.Figure()
fig.plot(
    region=[0, 10, 0, 10],
    projection="X10c",
    frame="ag",
    data=[circular_vector],
    style="m0.5c+ea",
    pen="2p",
    color="red3",
```

(continues on next page)

(continued from previous page)

```
)

with pygmt.config(PROJ_LENGTH_UNIT="i"):
    fig.plot(
        data=[circular_vector],
        style="m0.5c+ea",
        pen="2p",
        color="red3",
    )
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

9.12.3 Plot Geographic Vectors

On this map, `point_1` and `point_2` are coordinate pairs used to set the start and end points of the geographic vector. The geographical vector is going from Idaho to Chicago. To style geographic vectors, use `=` at the beginning of the `style` parameter. Other styling features such as vector stem thickness and head color can be passed into the `pen` and `color` parameters.

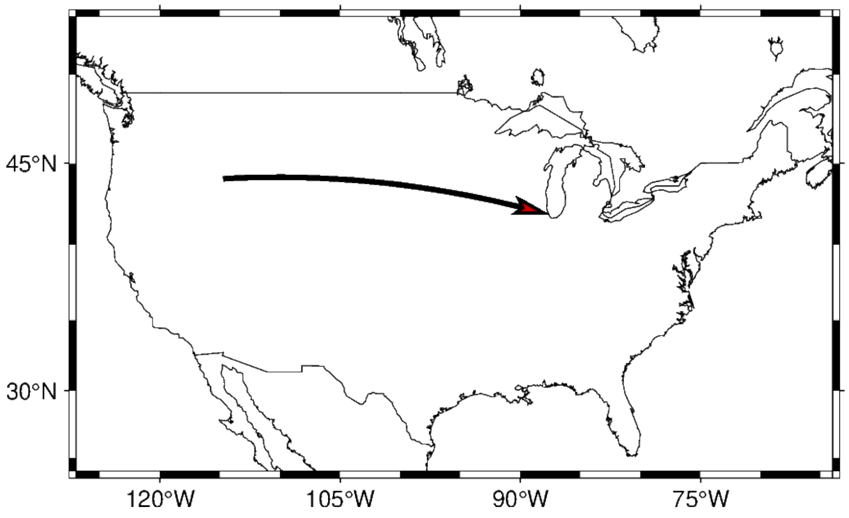
Note that the **+s** is added to use a startpoint and an endpoint to represent the vector instead of input angle and length.

```
point_1 = [-114.7420, 44.0682]
point_2 = [-87.6298, 41.8781]
```

(continues on next page)

(continued from previous page)

```
data = np.array([point_1 + point_2])

fig = pygmt.Figure()
fig.coast(
    region=[-127, -64, 24, 53],
    projection="M10c",
    frame=True,
    borders=1,
    shorelines="0.25p,black",
    area_thresh=4000,
)
fig.plot(
    data=data,
    style="=0.5c+ea+s",
    pen="2p",
    color="red3",
)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

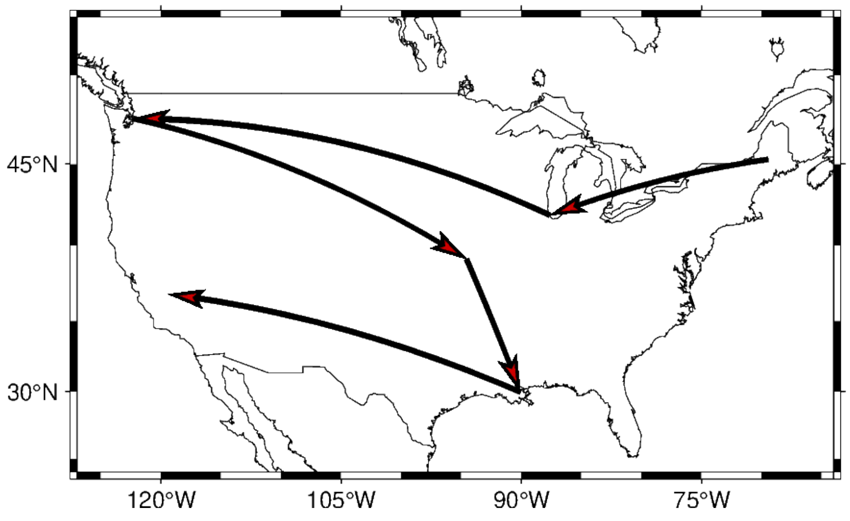
Using the same technique shown in the previous example, multiple vectors can be plotted in a chain where the endpoint of one is the starting point of another. This can be done by adding the coordinate lists together to create this structure: `[[start_latitude, start_longitude, end_latitude, end_longitude]]`. Each list within the 2D list contains the start and end information for each vector.

```
# Coordinate pairs for all the locations used
ME = [-69.4455, 45.2538]
CHI = [-87.6298, 41.8781]
SEA = [-122.3321, 47.6062]
NO = [-90.0715, 29.9511]
KC = [-94.5786, 39.0997]
CA = [-119.4179, 36.7783]

```

(continues on next page)

(continued from previous page)

```
# Add array to piece together the vectors
data = [ME + CHI, CHI + SEA, SEA + KC, KC + NO, NO + CA]

fig = pygmt.Figure()
fig.coast(
    region=[-127, -64, 24, 53],
    projection="M10c",
    frame=True,
    borders=1,
    shorelines="0.25p,black",
    area_thresh=4000,
)
fig.plot(
    data=data,
    style="=0.5c+ea+s",
    pen="2p",
    color="red3",
)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

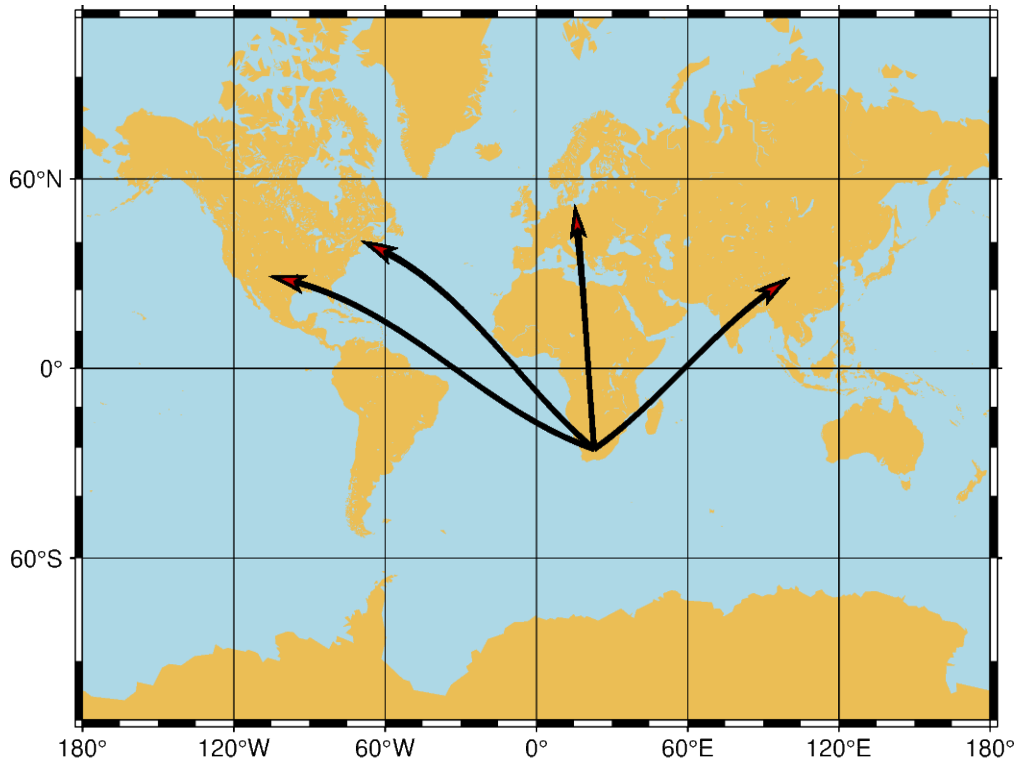
This example plots vectors over a Mercator projection. The starting points are located at SA which is South Africa and going to four different locations.

```
SA = [22.9375, -30.5595]
EUR = [15.2551, 54.5260]
ME = [-69.4455, 45.2538]
AS = [100.6197, 34.0479]
NM = [-105.8701, 34.5199]
data = np.array([SA + EUR, SA + ME, SA + AS, SA + NM])

fig = pygmt.Figure()
fig.coast(
```

(continues on next page)

(continued from previous page)

```
    region=[-180, 180, -80, 80],
    projection="M0/0/12c",
    frame="afg",
    land="lightbrown",
    water="lightblue",
)
fig.plot(
    data=data,
    style="=0.5c+ea+s",
    pen="2p",
    color="red3",
)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

Total running time of the script: (0 minutes 28.221 seconds)

9.13 Plotting datetime charts

PyGMT accepts a variety of datetime objects to plot data and create charts. Aside from the built-in Python `datetime` object, PyGMT supports input using ISO formatted strings, `pandas`, `xarray`, as well as `numpy`. These data types can be used to plot specific points as well as get passed into the `region` parameter to create a range of the data on an axis.

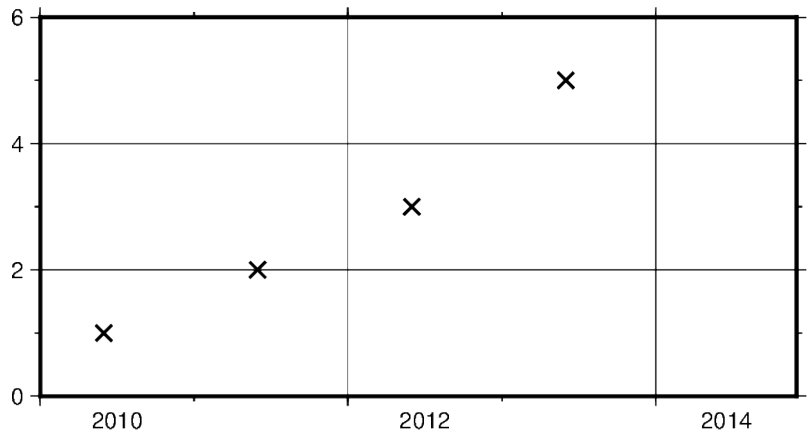
The following examples will demonstrate how to create plots using the different datetime objects.

```python
import datetime

import numpy as np
import pandas as pd
import pygmt
import xarray as xr
```

9.13.1 Using Python's `datetime`

In this example, Python's built-in `datetime` module is used to create data points stored in list `x`. Additionally, dates are passed into the `region` parameter in the format `(x_start, x_end, y_start, y_end)`, where the date range is plotted on the x-axis. An additional notable parameter is `style`, where it's specified that data points are to be plotted in an **X** shape with a size of 0.3 centimeters.

```python
x = [
    datetime.date(2010, 6, 1),
    datetime.date(2011, 6, 1),
    datetime.date(2012, 6, 1),
    datetime.date(2013, 6, 1),
]
y = [1, 2, 3, 5]

fig = pygmt.Figure()
fig.plot(
    projection="X10c/5c",
    region=[datetime.date(2010, 1, 1), datetime.date(2014, 12, 1), 0, 6],
    frame=["WSen", "afg"],
    x=x,
    y=y,
    style="x0.3c",
    pen="1p",
)
fig.show()
```



Out:

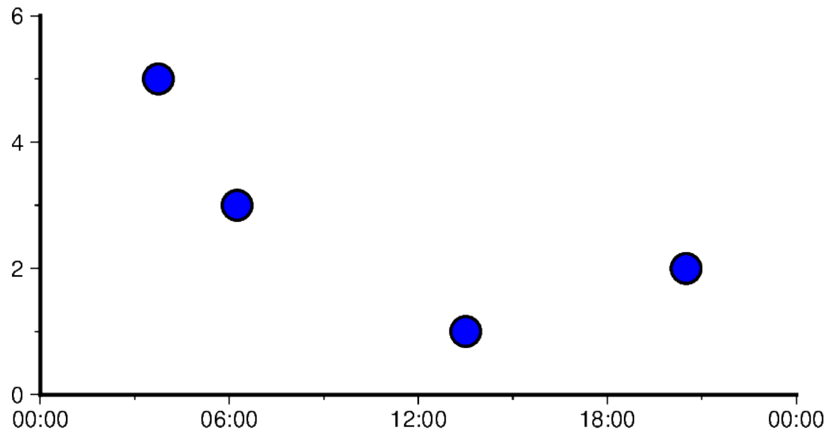
```
<IPython.core.display.Image object>
```

In addition to specifying the date, `datetime` supports the exact time at which the data points were recorded. Using `datetime.datetime` the `region` parameter as well as data points can be created with both date and time information.

Some notable differences to the previous example include

- Modifying `frame` to only include West (left) and South (bottom) borders, and removing grid lines
- Using circles to plot data points defined through `c` in `style` parameter

```
x = [
    datetime.datetime(2021, 1, 1, 3, 45, 1),
    datetime.datetime(2021, 1, 1, 6, 15, 1),
    datetime.datetime(2021, 1, 1, 13, 30, 1),
    datetime.datetime(2021, 1, 1, 20, 30, 1),
]
y = [5, 3, 1, 2]

fig = pygmt.Figure()
fig.plot(
    projection="X10c/5c",
    region=[
        datetime.datetime(2021, 1, 1, 0, 0, 0),
        datetime.datetime(2021, 1, 2, 0, 0, 0),
        0,
        6,
    ],
    frame=["WS", "af"],
    x=x,
    y=y,
    style="c0.4c",
    pen="1p",
    color="blue",
)
fig.show()
```



Out:

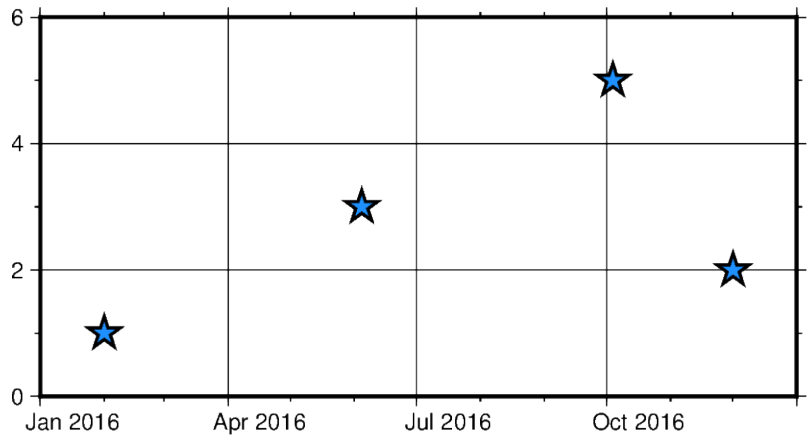
```
<IPython.core.display.Image object>
```

9.13.2 Using ISO Format

In addition to Python's `datetime` library, PyGMT also supports passing times in ISO format. Basic ISO strings are formatted as `YYYY-MM-DD` with each `-` delineated section marking the four digit year value, two digit month value, and two digit day value respectively.

When including time of day into ISO strings, the `T` character is used, as can be seen in the following example. This character is immediately followed by a string formatted as `hh:mm:ss` where each `:` delineated section marking the two digit hour value, two digit minute value, and two digit second value respectively. The figure in the following example is plotted over a horizontal range of one year from 1/1/2016 to 1/1/2017.

```
x = ["2016-02-01", "2016-06-04T14", "2016-10-04T00:00:15", "2016-12-01T05:00:15"]
y = [1, 3, 5, 2]
fig = pygmt.Figure()
fig.plot(
    projection="X10c/5c",
    region=["2016-01-01", "2017-01-1", 0, 6],
    frame=["WSen", "afg"],
    x=x,
    y=y,
    style="a0.45c",
    pen="1p",
    color="dodgerblue",
)
fig.show()
```



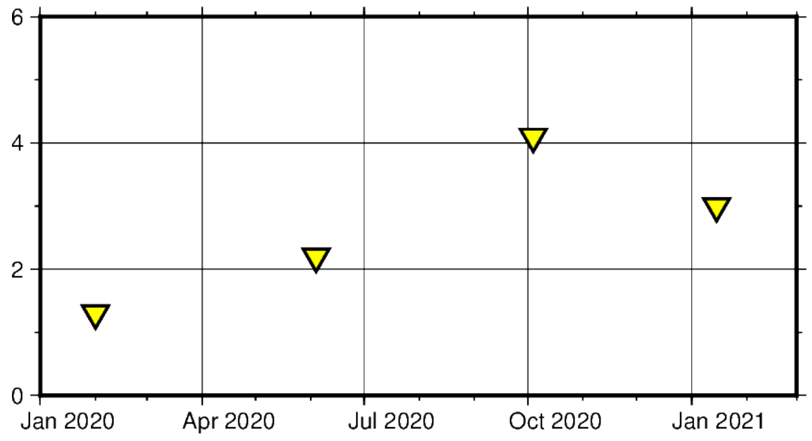
Out:

```
<IPython.core.display.Image object>
```

9.13.3 Mixing and matching Python datetime and ISO dates

The following example provides context on how both `datetime` and ISO date data can be plotted using PyGMT. This can be helpful when dates and times are coming from different sources, meaning conversions do not need to take place between ISO and datetime in order to create valid plots.

```
x = ["2020-02-01", "2020-06-04", "2020-10-04", datetime.datetime(2021, 1, 15)]
y = [1.3, 2.2, 4.1, 3]
fig = pygmt.Figure()
fig.plot(
    projection="X10c/5c",
    region=[datetime.datetime(2020, 1, 1), datetime.datetime(2021, 3, 1), 0, 6],
    frame=["WSen", "afg"],
    x=x,
    y=y,
    style="i0.4c",
    pen="1p",
    color="yellow",
)
fig.show()
```



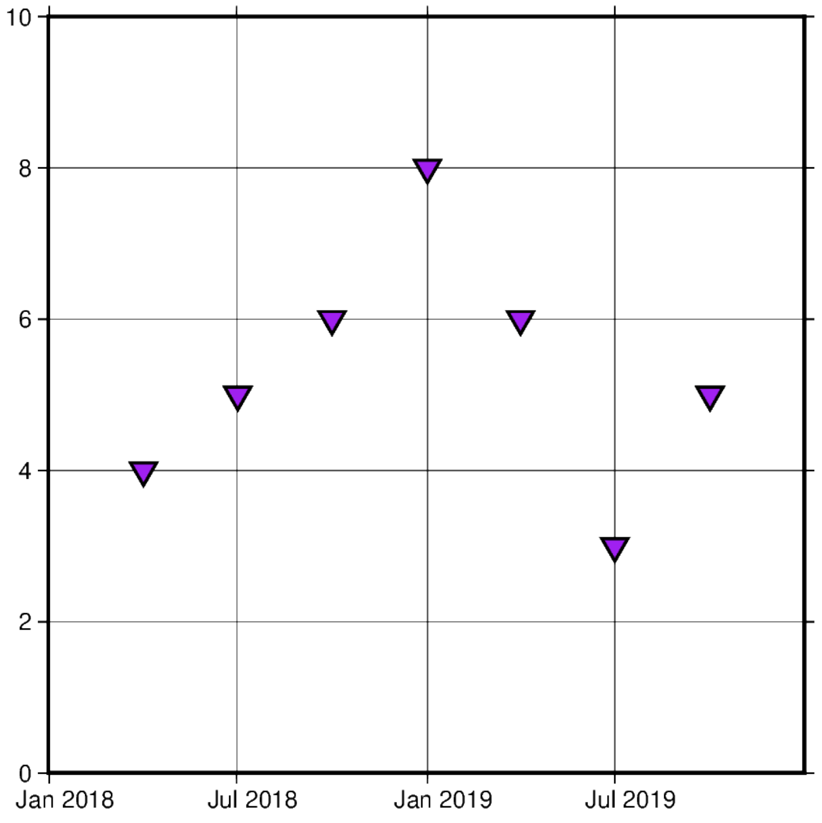
Out:

```
<IPython.core.display.Image object>
```

9.13.4 Using pandas.date_range

In the following example, pandas.date_range produces a list of pandas.DatetimeIndex objects, which gets is used to pass date data to the PyGMT figure. Specifically x contains 7 different pandas.DatetimeIndex objects, with the number being manipulated by the periods parameter. Each period begins at the start of a business quarter as denoted by BQS when passed to the periods parameter. The initial date is the first argument that is passed to pandas.date_range and it marks the first data point in the list x that will be plotted.

```python
x = pd.date_range("2018-03-01", periods=7, freq="BQS")
y = [4, 5, 6, 8, 6, 3, 5]

fig = pygmt.Figure()
fig.plot(
    projection="X10c/10c",
    region=[datetime.datetime(2017, 12, 31), datetime.datetime(2019, 12, 31), 0, 10],
    frame=["WSen", "ag"],
    x=x,
    y=y,
    style="i0.4c",
    pen="1p",
    color="purple",
)
fig.show()
```



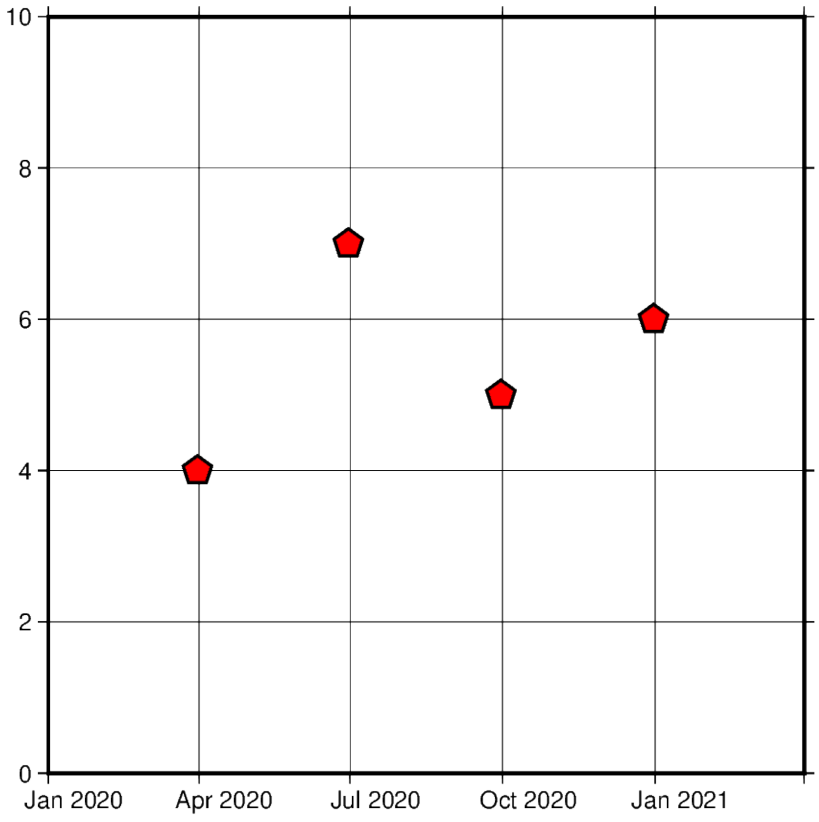
Out:

```
<IPython.core.display.Image object>
```

9.13.5 Using `xarray.DataArray`

In this example, instead of using a `pandas.date_range`, `x` is initialized as a list of `xarray.DataArray` objects. This object provides a wrapper around regular PyData formats. It also allows the data to have labeled dimensions while supporting operations that use various pieces of metadata.The following code uses `pandas.date_range` object to fill the DataArray with data, but this is not essential for the creation of a valid DataArray.

```python
x = xr.DataArray(data=pd.date_range(start="2020-01-01", periods=4, freq="Q"))
y = [4, 7, 5, 6]

fig = pygmt.Figure()
fig.plot(
    projection="X10c/10c",
    region=[datetime.datetime(2020, 1, 1), datetime.datetime(2021, 4, 1), 0, 10],
    frame=["WSen", "ag"],
    x=x,
    y=y,
    style="n0.4c",
    pen="1p",
    color="red",
)
fig.show()
```



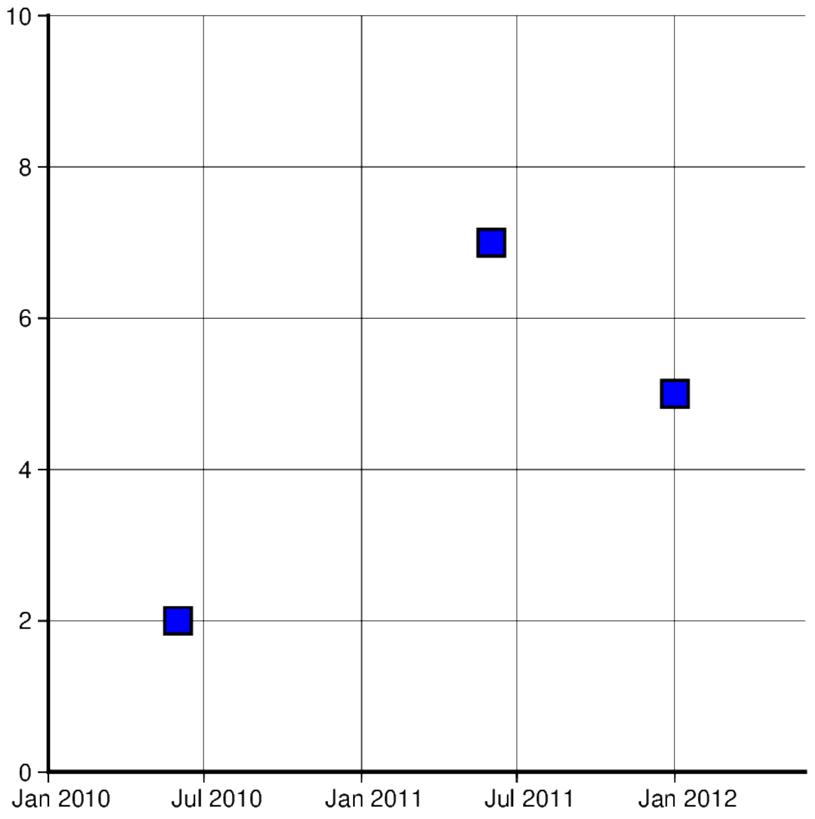
Out:

```
<IPython.core.display.Image object>
```

9.13.6 Using `numpy.datetime64`

In this example, instead of using a `pd.date_range`, `x` is initialized as an `np.array` object. Similar to `xarray.DataArray` this wraps the dataset before passing it as a parameter. However, `np.array` objects use less memory and allow developers to specify datatypes.

```
x = np.array(["2010-06-01", "2011-06-01T12", "2012-01-01T12:34:56"], dtype="datetime64")
y = [2, 7, 5]

fig = pygmt.Figure()
fig.plot(
    projection="X10c/10c",
    region=[datetime.datetime(2010, 1, 1), datetime.datetime(2012, 6, 1), 0, 10],
    frame=["WS", "ag"],
    x=x,
    y=y,
    style="s0.5c",
    pen="1p",
    color="blue",
)
fig.show()
```



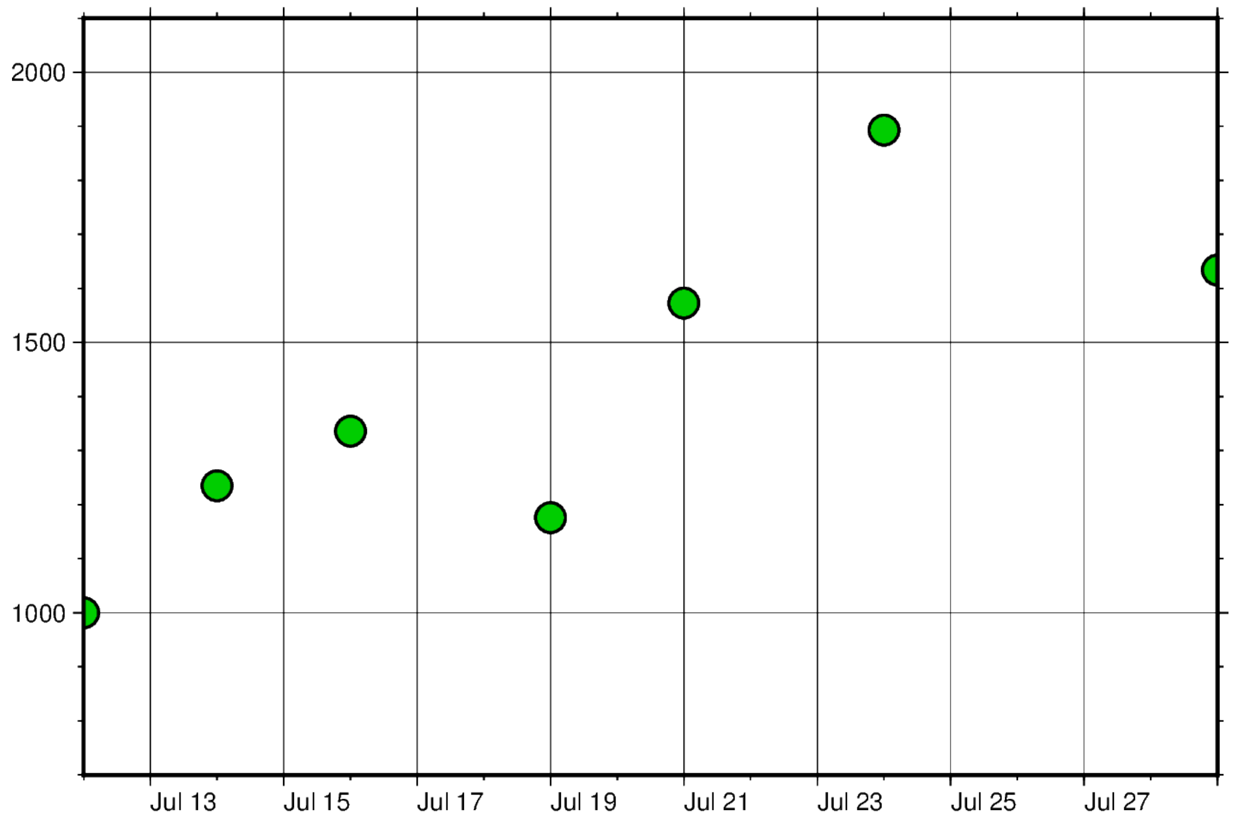
Out:

```
<IPython.core.display.Image object>
```

9.13.7 Generating an automatic region

Another way of creating charts involving datetime data can be done by automatically generating the region of the plot. This can be done by passing the dataframe to *pygmt.info*, which will find maximum and minimum values for each column and create a list that could be passed as region. Additionally, the `spacing` argument can be passed to increase the range past the maximum and minimum data points.

```
data = [
    ["20200712", 1000],
    ["20200714", 1235],
    ["20200716", 1336],
    ["20200719", 1176],
    ["20200721", 1573],
    ["20200724", 1893],
    ["20200729", 1634],
]
df = pd.DataFrame(data, columns=["Date", "Score"])
df.Date = pd.to_datetime(df["Date"], format="%Y%m%d")

fig = pygmt.Figure()
region = pygmt.info(
    data=df[["Date", "Score"]], per_column=True, spacing=(700, 700), coltypes="T"
)

fig.plot(
    region=region,
    projection="X15c/10c",
    frame=["WSen", "afg"],
    x=df.Date,
    y=df.Score,
    style="c0.4c",
    pen="1p",
    color="green3",
)

fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

9.13.8 Setting Primary and Secondary Time Axes

This example focuses on labeling the axes and setting intervals at which the labels are expected to appear. All of these modifications are added to the `frame` parameter and each item in that list modifies a specific section of the plot.

Starting off with `WS`, adding this string means that only Western/Left (**W**) and Southern/Bottom (**S**) borders of the plot will be shown. For more information on this, please refer to *frame instructions*.

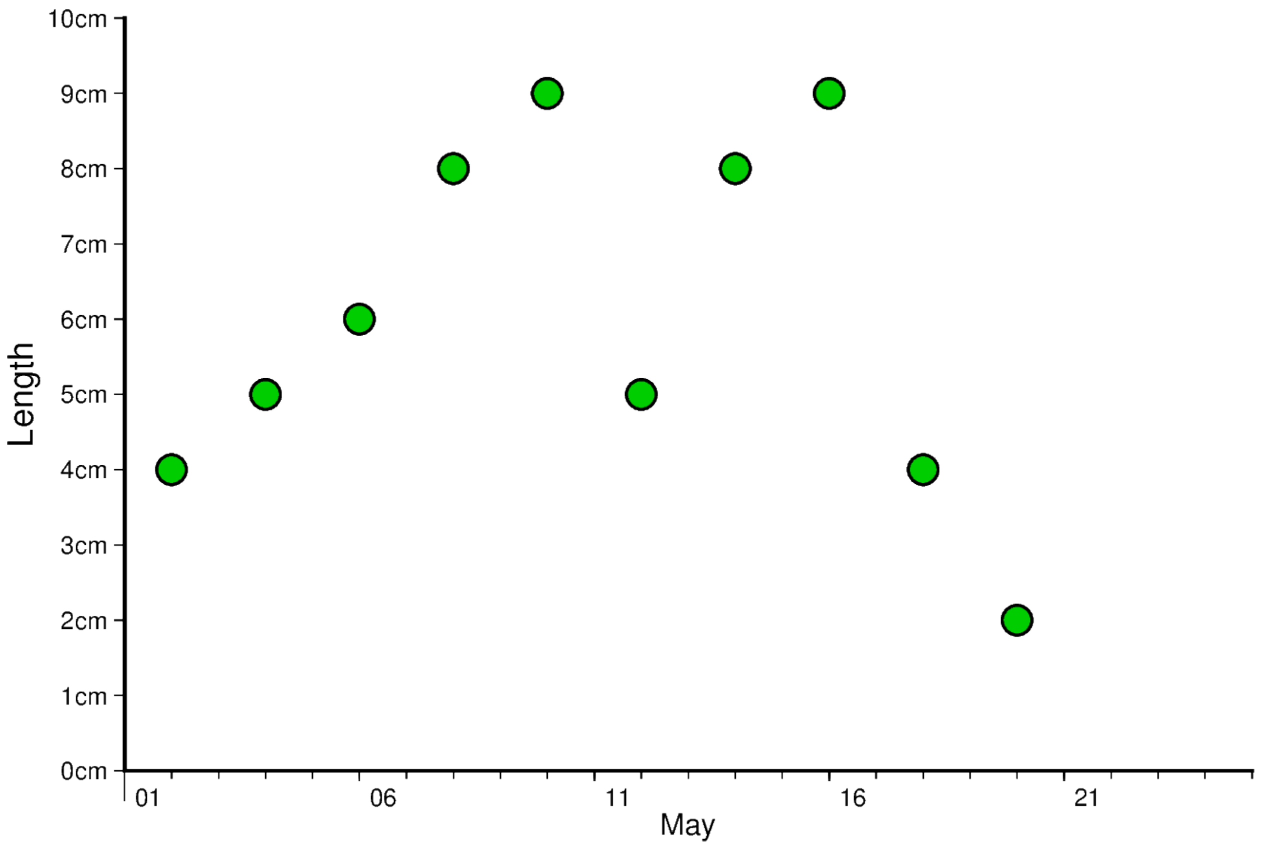
The other important item in the `frame` list is `"sxa1Of1D"`. This string modifies the secondary labeling (**s**) of the x-axis (**x**). Specifically, it sets the main annotation and major tick spacing interval to one month (**a1O**) (capital letter o, not zero). Additionally, it sets the minor tick spacing interval to 1 day (**f1D**). The labeling of this axis can be modified by setting FORMAT_DATE_MAP to 'o' to use the month's name instead of its number. More information about configuring date formats can be found on the official GMT documentation page.

```python
x = pd.date_range("2013-05-02", periods=10, freq="2D")
y = [4, 5, 6, 8, 9, 5, 8, 9, 4, 2]

fig = pygmt.Figure()
with pygmt.config(FORMAT_DATE_MAP="o"):
    fig.plot(
        projection="X15c/10c",
        region=[datetime.datetime(2013, 5, 1), datetime.datetime(2013, 5, 25), 0, 10],
        frame=["WS", "sxa1Of1D", "pxa5d", "sy+lLength", "pya1+ucm"],
```

(continues on next page)

(continued from previous page)

```
        x=x,
        y=y,
        style="c0.4c",
        pen="1p",
        color="green3",
    )

fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

The same concept shown above can be applied to smaller as well as larger intervals. In this example, data is plotted for different times throughout two days. Primary x-axis labels are modified to repeat every 6 hours and secondary x-axis label repeats every day and shows the day of the week.

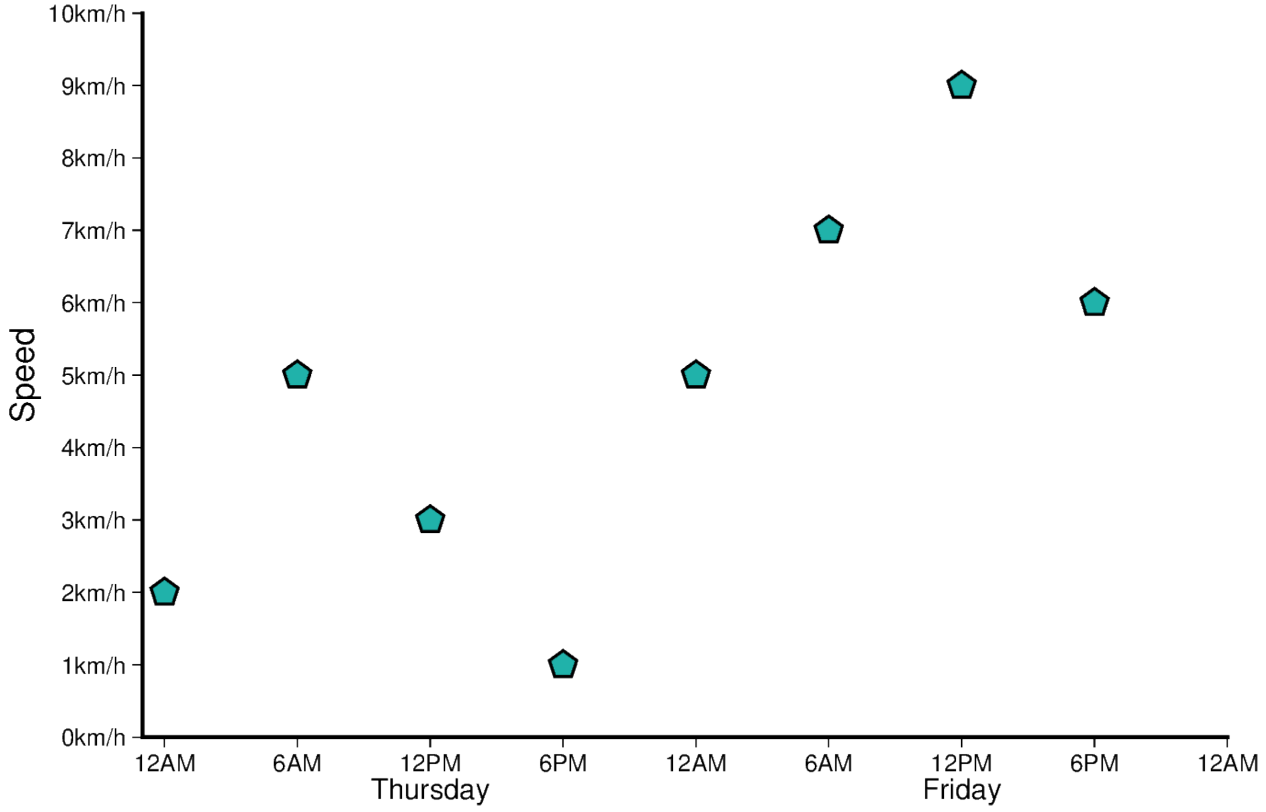
Another notable mention in this example is setting FORMAT_CLOCK_MAP to "-hhAM" which specifies the format used for time. In this case, leading zeros are removed using (**-**), and only hours are displayed. Additionally, an AM/PM system is being used instead of a 24-hour system. More information about configuring time formats can be found on the official GMT documentation page.

```
x = pd.date_range("2021-04-15", periods=8, freq="6H")
y = [2, 5, 3, 1, 5, 7, 9, 6]

fig = pygmt.Figure()
```

(continues on next page)

(continued from previous page)

```
with pygmt.config(FORMAT_CLOCK_MAP="-hhAM"):
    fig.plot(
        projection="X15c/10c",
        region=[
            datetime.datetime(2021, 4, 14, 23, 0, 0),
            datetime.datetime(2021, 4, 17),
            0,
            10,
        ],
        frame=["WS", "sxa1K", "pxa6H", "sy+lSpeed", "pya1+ukm/h"],
        x=x,
        y=y,
        style="n0.4c",
        pen="1p",
        color="lightseagreen",
    )
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

Total running time of the script: (0 minutes 21.408 seconds)

9.14 Plotting text

It is often useful to add annotations to a map plot. This is handled by *pygmt.Figure.text*.

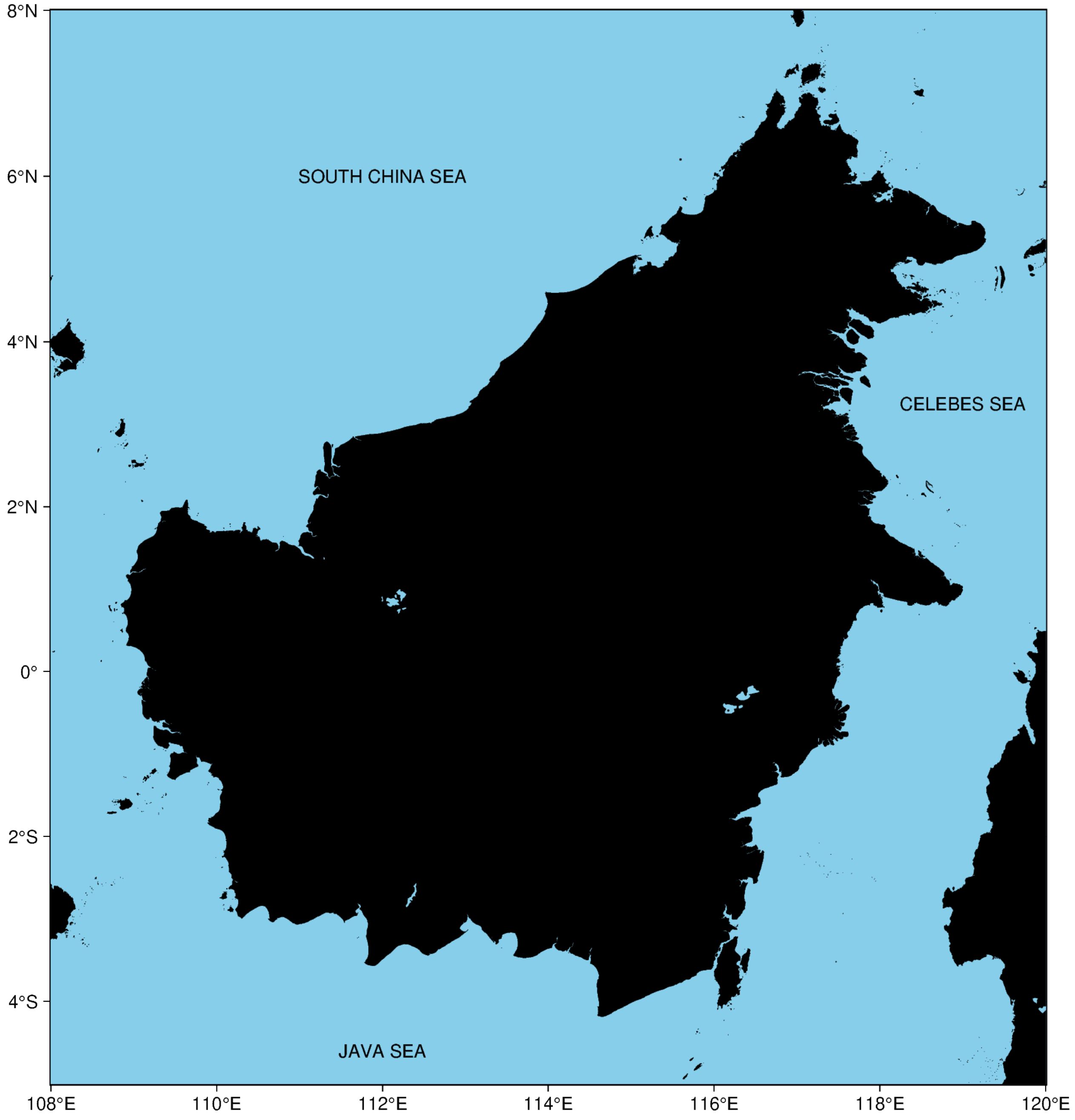
```
import os

import pygmt
```

9.14.1 Basic map annotation

Text annotations can be added to a map using the *pygmt.Figure.text* method of the *pygmt.Figure* class.

Here we create a simple map and add an annotation using the `text`, `x`, and `y` parameters to specify the annotation text and position in the projection frame. `text` accepts *str* types, while `x`, and `y` accepts either *int* or *float* numbers, or a list/array of numbers.

```
fig = pygmt.Figure()
with pygmt.config(MAP_FRAME_TYPE="plain"):
    fig.basemap(region=[108, 120, -5, 8], projection="M20c", frame="a")
fig.coast(land="black", water="skyblue")

# Plot text annotations using a single element
fig.text(text="SOUTH CHINA SEA", x=112, y=6)

# Plot text annotations using lists of elements
fig.text(text=["CELEBES SEA", "JAVA SEA"], x=[119, 112], y=[3.25, -4.6])

fig.show()
```


8°N
6°N
4°N
2°N
0°
2°S
4°S
SOUTH CHINA SEA
CELEBES SEA
JAVA SEA
108°E
110°E
112°E
114°E
116°E
118°E
120°E

Out:

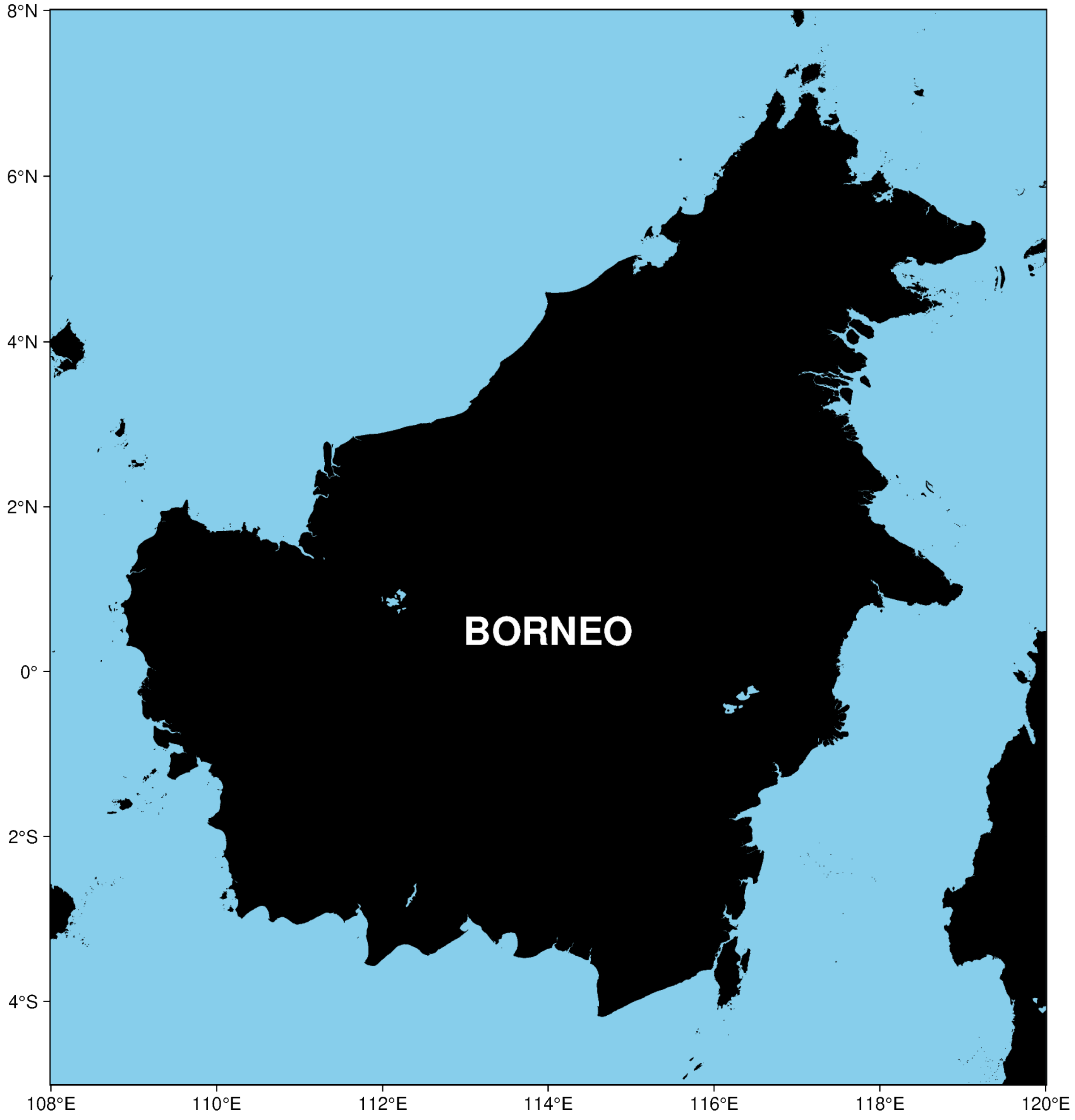
```
<IPython.core.display.Image object>
```

9.14.2 Changing font style

The size, family/weight, and color of an annotation can be specified using the `font` parameter.

A list of all recognized fonts can be found at PostScript Fonts Used by GMT, including details of how to use non-default fonts.

```
fig = pygmt.Figure()
with pygmt.config(MAP_FRAME_TYPE="plain"):
    fig.basemap(region=[108, 120, -5, 8], projection="M20c", frame="a")
fig.coast(land="black", water="skyblue")

# Customize the font style
fig.text(text="BORNEO", x=114.0, y=0.5, font="22p,Helvetica-Bold,white")

fig.show()
```



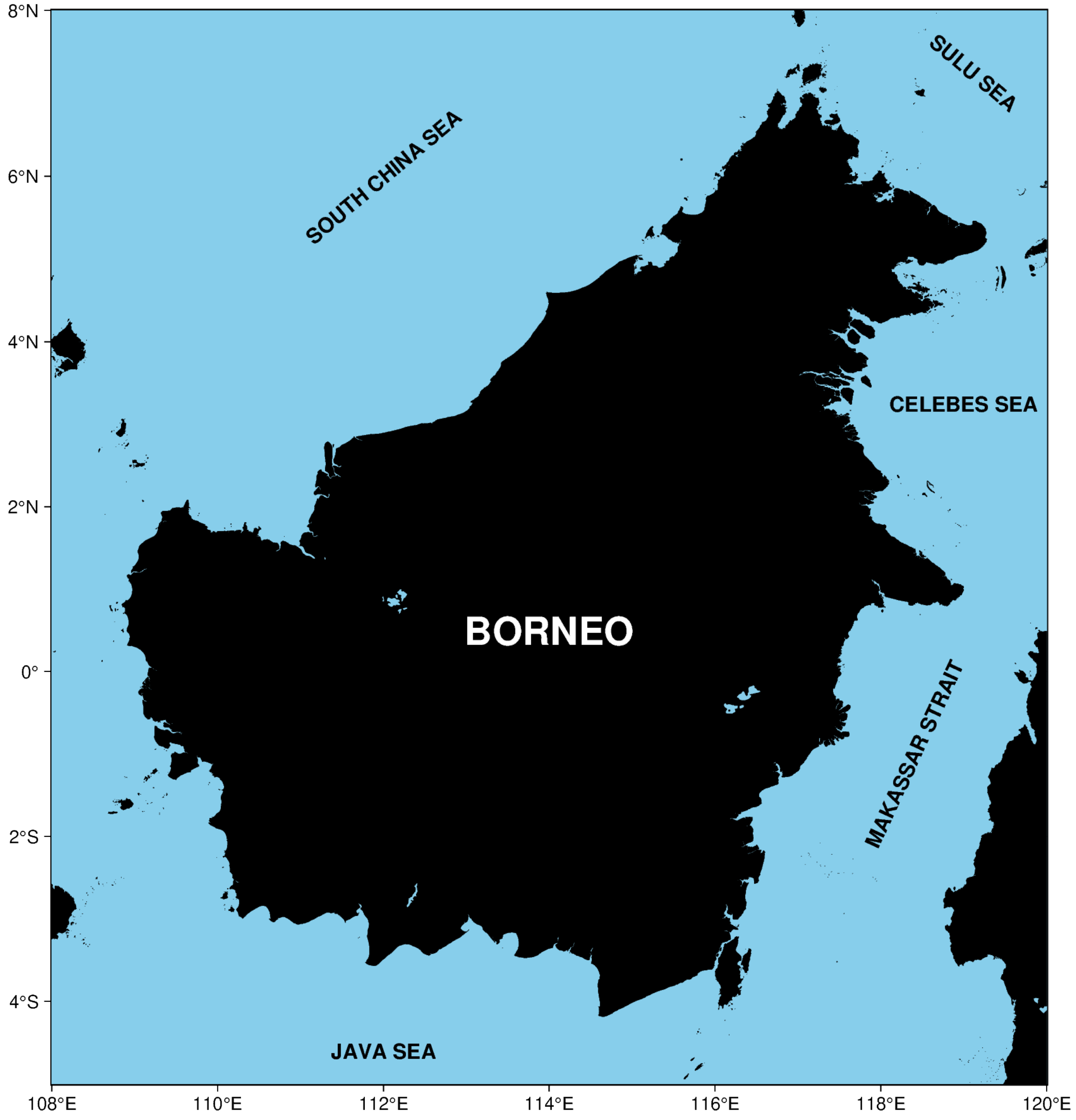
Out:

```
<IPython.core.display.Image object>
```

9.14.3 Plotting from a text file

It is also possible to add annotations from a file containing `x`, `y`, and `text` fields. Here we give a complete example.

```
fig = pygmt.Figure()
with pygmt.config(MAP_FRAME_TYPE="plain"):
    fig.basemap(region=[108, 120, -5, 8], projection="M20c", frame="a")
fig.coast(land="black", water="skyblue")

# Create space-delimited file
with open("examples.txt", "w") as f:
    f.write("114 0.5 0 22p,Helvetica-Bold,white CM BORNEO\n")
    f.write("119 3.25 0 12p,Helvetica-Bold,black CM CELEBES SEA\n")
    f.write("112 -4.6 0 12p,Helvetica-Bold,black CM JAVA SEA\n")
    f.write("112 6 40 12p,Helvetica-Bold,black CM SOUTH CHINA SEA\n")
    f.write("119.12 7.25 -40 12p,Helvetica-Bold,black CM SULU SEA\n")
    f.write("118.4 -1 65 12p,Helvetica-Bold,black CM MAKASSAR STRAIT\n")

# Plot region names / sea names from a text file, where
# the longitude (x) and latitude (y) coordinates are in the first two columns.
# Setting angle/font/justify to True will indicate that those columns are
# present in the text file too (Note: must be in that order!).
# Finally, the text to be printed will be in the last column
fig.text(textfiles="examples.txt", angle=True, font=True, justify=True)

# Cleanups
os.remove("examples.txt")

fig.show()
```


SOUTH CHINA SEA
SULU SEA
CELEBES SEA
BORNEO
MAKASSAR STRAIT
JAVA SEA
8°N
6°N
4°N
2°N
0°
2°S
4°S
108°E
110°E
112°E
114°E
116°E
118°E
120°E

Out:

```
<IPython.core.display.Image object>
```

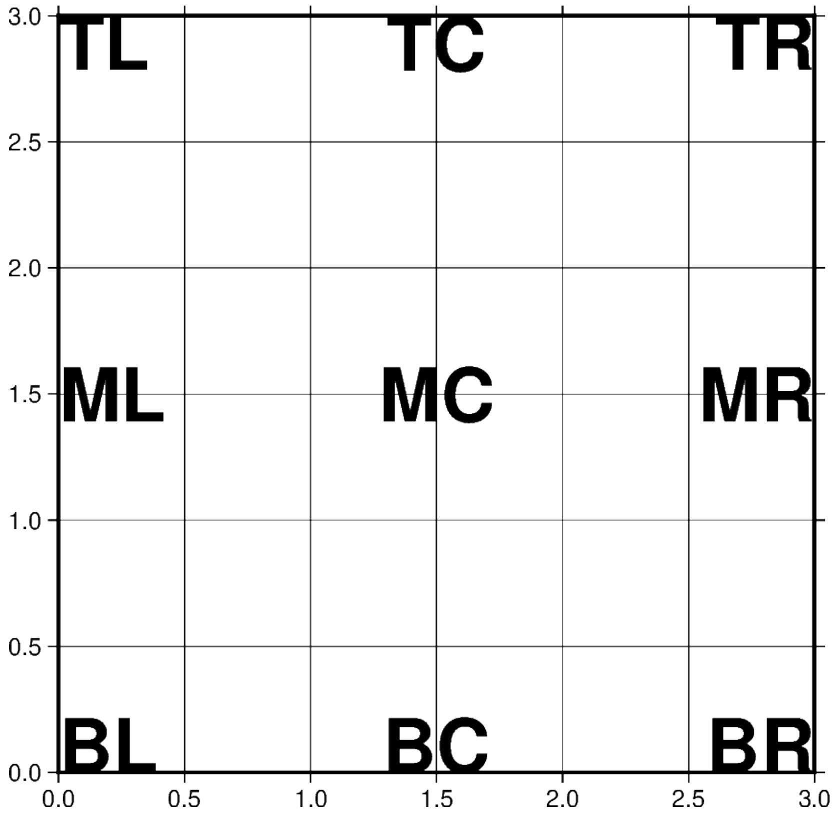
9.14.4 `justify` parameter

`justify` is used to define the anchor point for the bounding box for text being added to a plot. The following code segment demonstrates the positioning of the anchor point relative to the text.

The anchor is specified with a two letter (order independent) code, chosen from:

- Vertical anchor: **T**(op), **M**(iddle), **B**(ottom)
- Horizontal anchor: **L**(eft), **C**(entre), **R**(ight)

```
fig = pygmt.Figure()
fig.basemap(region=[0, 3, 0, 3], projection="X10c", frame=["WSne", "af0.5g"])
for position in ("TL", "TC", "TR", "ML", "MC", "MR", "BL", "BC", "BR"):
    fig.text(
        text=position,
        position=position,
        font="28p,Helvetica-Bold,black",
        justify=position,
    )
fig.show()
```



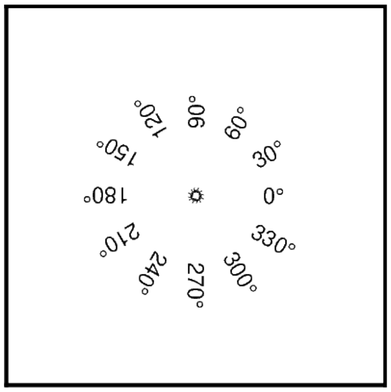
Out:

```
<IPython.core.display.Image object>
```

9.14.5 `angle` parameter

`angle` is an optional parameter used to specify the clockwise rotation of the text from the horizontal.

```
fig = pygmt.Figure()
fig.basemap(region=[0, 4, 0, 4], projection="X5c", frame="WSen")
for i in range(0, 360, 30):
    fig.text(text=f"`          {i}@.", x=2, y=2, justify="LM", angle=i)
fig.show()
```



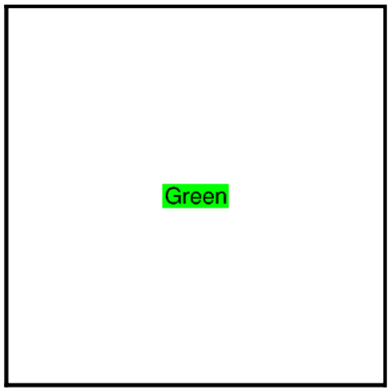
Out:

```
<IPython.core.display.Image object>
```

9.14.6 `fill` parameter

`fill` is used to set the fill color of the area surrounding the text.

```
fig = pygmt.Figure()
fig.basemap(region=[0, 1, 0, 1], projection="X5c", frame="WSen")
fig.text(text="Green", x=0.5, y=0.5, fill="green")
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

9.14.7 Advanced configuration

For crafting more advanced styles, be sure to check out the GMT documentation at https://docs.generic-mapping-tools.org/latest/text.html and also the cookbook at https://docs.generic-mapping-tools.org/latest/cookbook/features.html#placement-of-text. Good luck!

Total running time of the script: (0 minutes 25.336 seconds)

9.15 Creating a map with contour lines

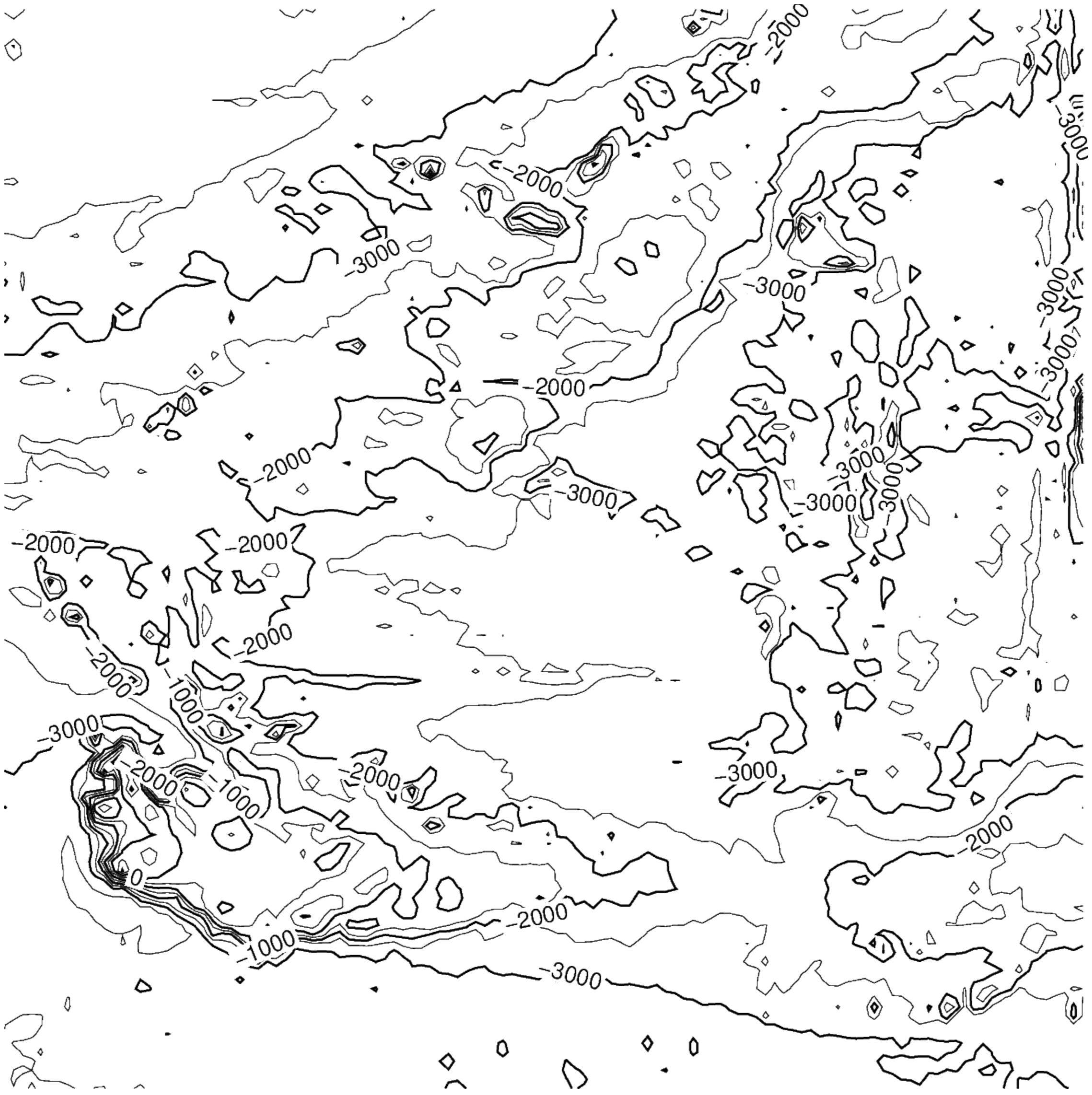
Plotting a contour map is handled by *pygmt.Figure.grdcontour*.

```
import pygmt

# Load sample earth relief data
grid = pygmt.datasets.load_earth_relief(resolution="05m", region=[-92.5, -82.5, -3, 7])
```

9.15.1 Create contour plot

The *pygmt.Figure.grdcontour* method takes the grid input. It plots annotated contour lines, which are thicker and have the elevation/depth written on them, and unannotated contour lines. In the example below, the default contour line intervals are 500 meters, with an annotated contour line every 1000 meters. By default, it plots the map with the equidistant cylindrical projection and with no frame.

```
fig = pygmt.Figure()
fig.grdcontour(grid=grid)
fig.show()
```



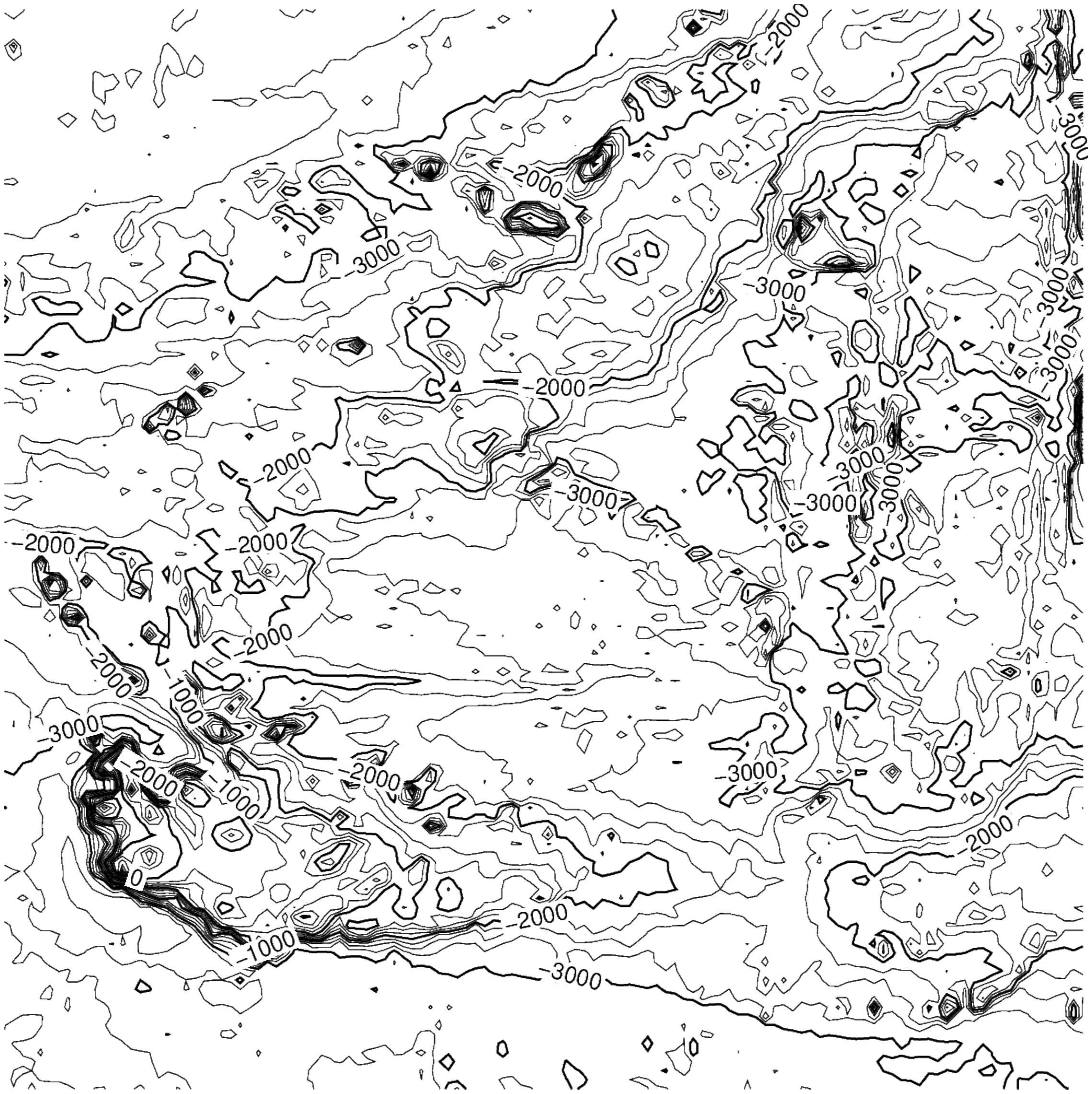
Out:

```
<IPython.core.display.Image object>
```

9.15.2 Contour line settings

Use the `annotation` and `interval` arguments to adjust contour line intervals. In the example below, there are contour intervals every 250 meters and annotated contour lines every 1,000 meters.

```
fig = pygmt.Figure()
fig.grdcontour(
    annotation=1000,
    interval=250,
    grid=grid,
)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

9.15.3 Contour limits

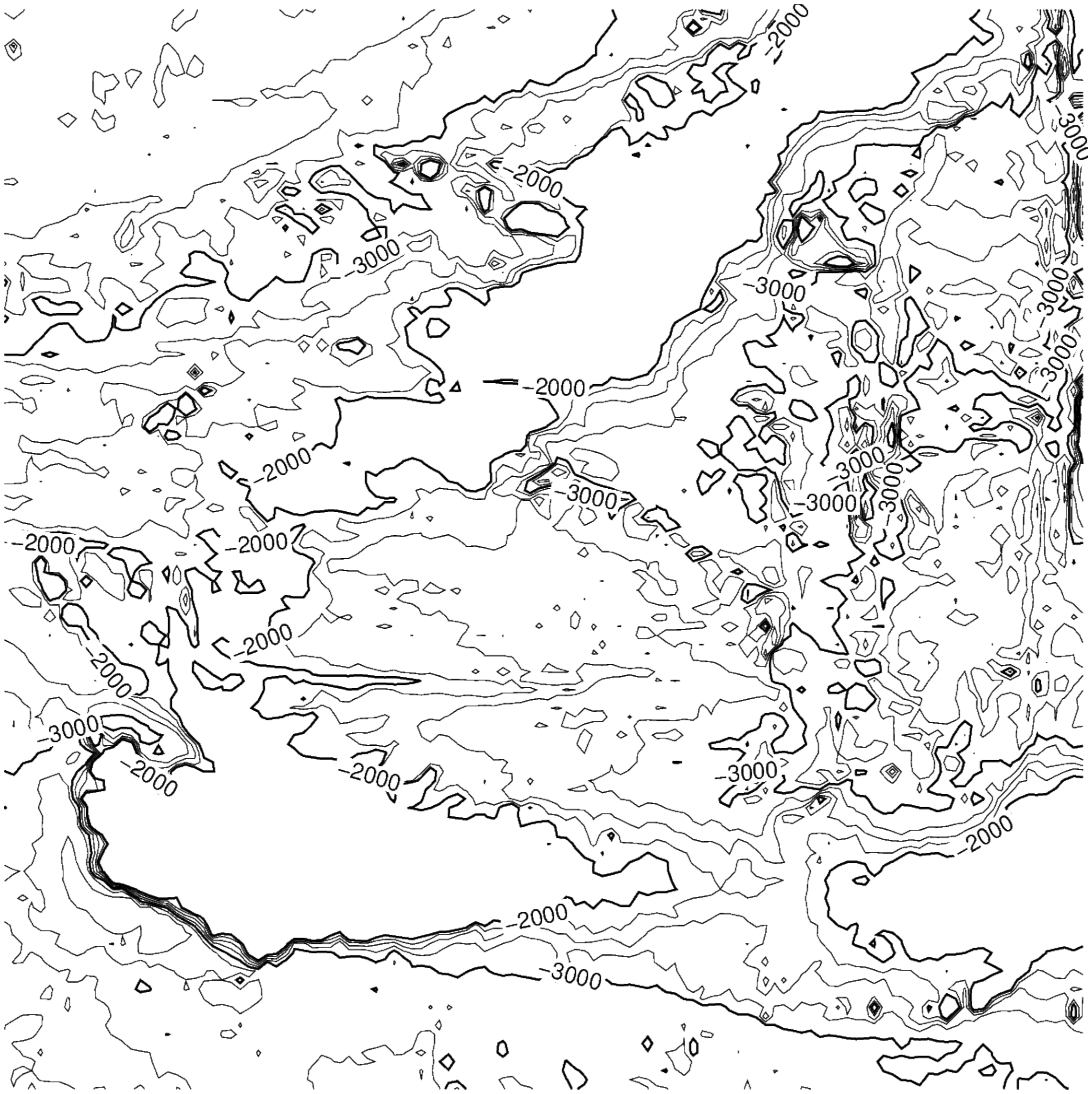
The `limit` argument sets the minimum and maximum values for the contour lines. The argument takes the low and high values, and is either a list (as below) or a string `limit="-4000/-2000"`.

```
fig = pygmt.Figure()
fig.grdcontour(
    annotation=1000,
    interval=250,
    grid=grid,
    limit=[-4000, -2000],
)
```

(continues on next page)

(continued from previous page)

```
fig.show()
```



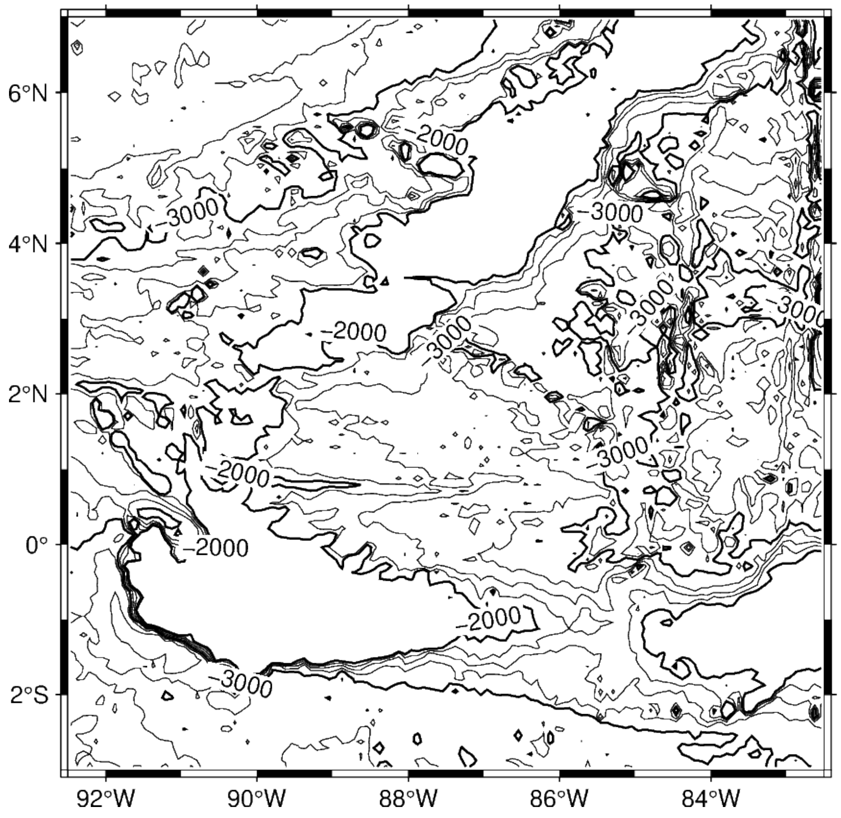
Out:

```
<IPython.core.display.Image object>
```

9.15.4 Map settings

The *pygmt.Figure.grdcontour* method accepts additional arguments, including setting the projection and frame.

```
fig = pygmt.Figure()
fig.grdcontour(
    annotation=1000,
    interval=250,
    grid=grid,
    limit=[-4000, -2000],
    projection="M10c",
    frame=True,
)
fig.show()
```



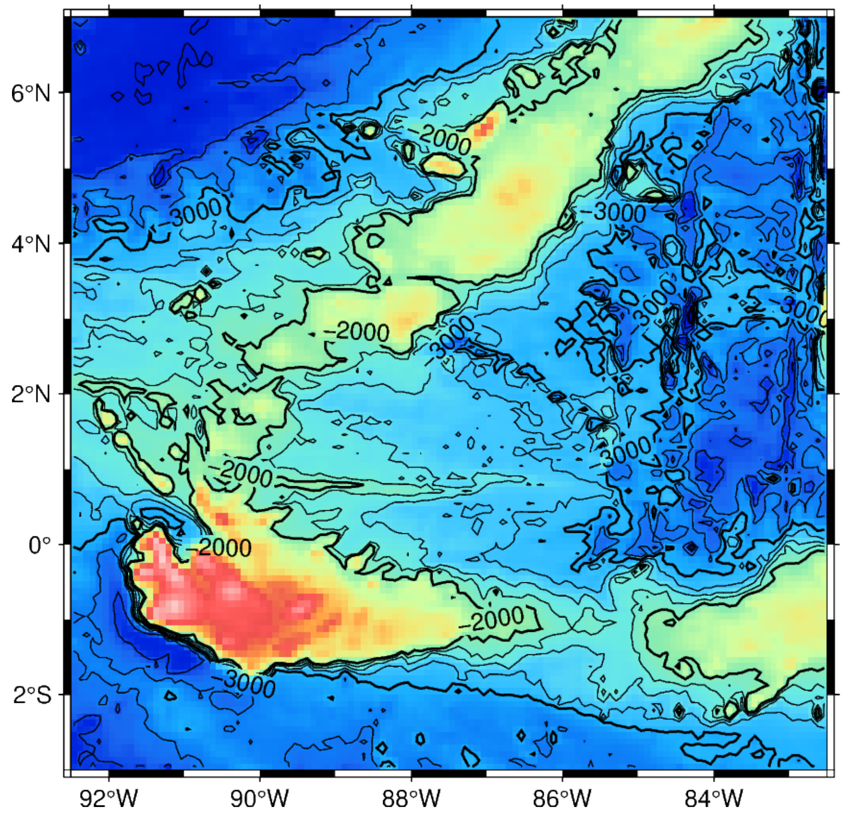
Out:

```
<IPython.core.display.Image object>
```

9.15.5 Adding a colormap

The *pygmt.Figure.grdimage* method can be used to add a colormap to the contour map. It must be called prior to *pygmt.Figure.grdcontour* to keep the contour lines visible on the final map. If the `projection` argument is specified in the *pygmt.Figure.grdimage* method, it does not need to be repeated in the *pygmt.Figure.grdcontour* method.

```
fig = pygmt.Figure()
fig.grdimage(
    grid=grid,
    cmap="haxby",
    projection="M10c",
    frame=True,
)
fig.grdcontour(
    annotation=1000,
    interval=250,
    grid=grid,
    limit=[-4000, -2000],
)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

Total running time of the script: (0 minutes 13.517 seconds)

9.16 Plotting Earth relief

Plotting a map of Earth relief can use the data accessed by the *pygmt.datasets.load_earth_relief* method. The data can then be plotted using the *pygmt.Figure.grdimage* method.

```
import pygmt
```

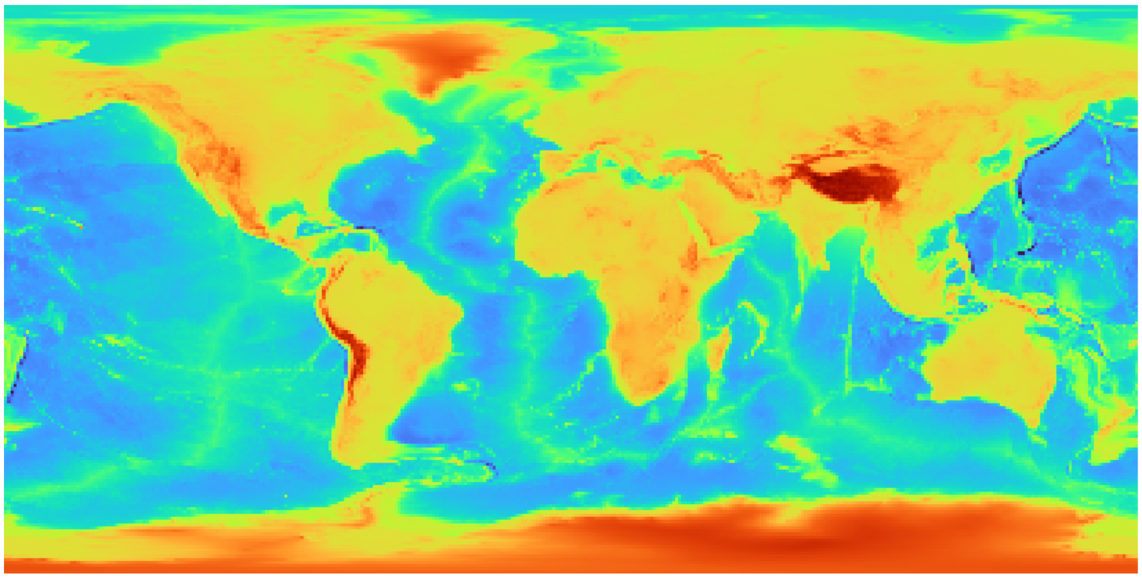
Load sample Earth relief data for the entire globe at a resolution of 1 arc degree. The other available resolutions are show at https://docs.generic-mapping-tools.org/latest/datasets/remote-data.html#global-earth-relief-grids.

```
grid = pygmt.datasets.load_earth_relief(resolution="01d")
```

9.16.1 Create a plot

The *pygmt.Figure.grdimage* method takes the `grid` input to create a figure. It creates and applies a color palette to the figure based upon the z-values of the data. By default, it plots the map with the *turbo* CPT, an equidistant cylindrical projection, and with no frame.

```
fig = pygmt.Figure()
fig.grdimage(grid=grid)
fig.show()
```

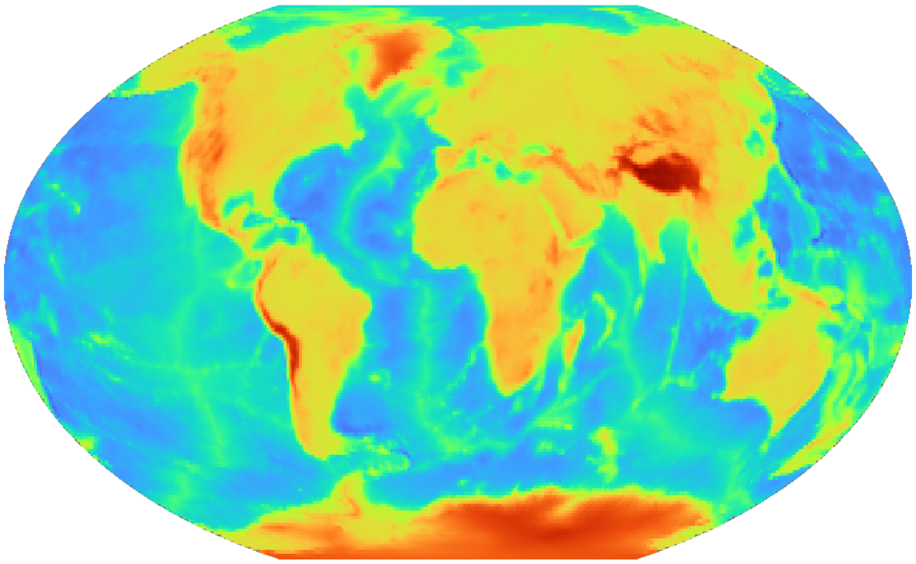


Out:

```
<IPython.core.display.Image object>
```

pygmt.Figure.grdimage can take the optional argument `projection` for the map. In the example below, the `projection` is set as `R12c` for 12 centimeter figure with a Winkel Tripel projection. For a list of available projections, see https://docs.generic-mapping-tools.org/latest/cookbook/map-projections.html.

```
fig = pygmt.Figure()
fig.grdimage(grid=grid, projection="R12c")
fig.show()
```



Out:

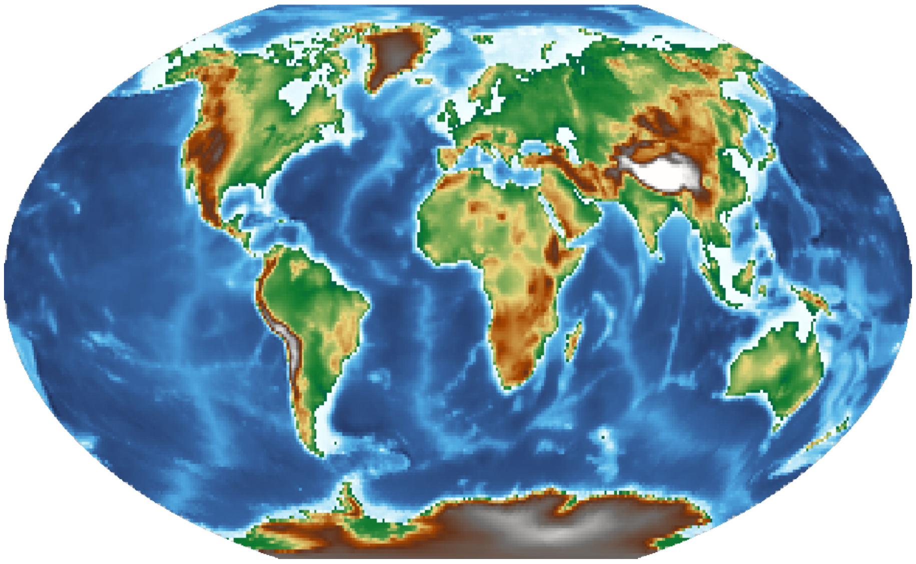
```
<IPython.core.display.Image object>
```

9.16.2 Set a color map

pygmt.Figure.grdimage takes the cmap argument to set the CPT of the figure. Examples of common CPTs for Earth relief are shown below. A full list of CPTs can be found at https://docs.generic-mapping-tools.org/latest/cookbook/cpts.html.

Using the *geo* CPT:

```
fig = pygmt.Figure()
fig.grdimage(grid=grid, projection="R12c", cmap="geo")
fig.show()
```

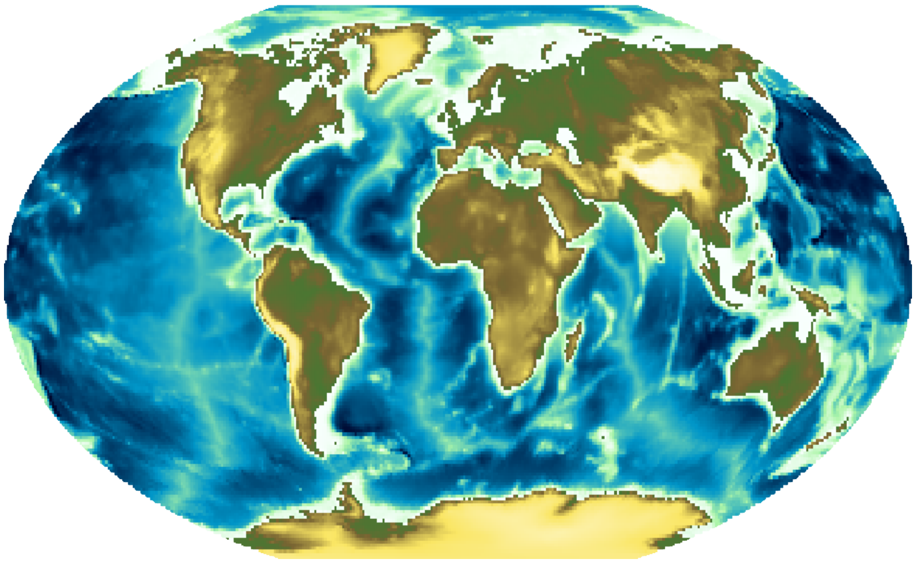


Out:

```
<IPython.core.display.Image object>
```

Using the *relief* CPT:

```
fig = pygmt.Figure()
fig.grdimage(grid=grid, projection="R12c", cmap="relief")
fig.show()
```



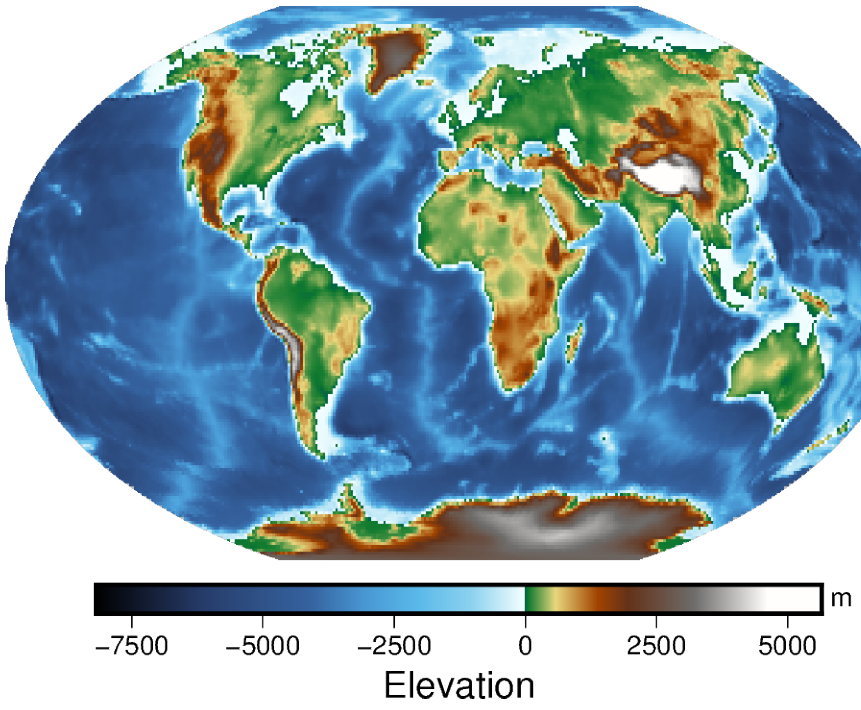
Out:

```
<IPython.core.display.Image object>
```

9.16.3 Add a color bar

The *pygmt.Figure.colorbar* method displays the CPT and the associated Z-values of the figure, and by default uses the same CPT set by the cmap argument for *pygmt.Figure.grdimage*. The frame argument for *pygmt.Figure.colorbar* can be used to set the axis intervals and labels. A list is used to pass multiple arguments to frame. In the example below, a2500 sets the axis interval to 2,500, x+lElevation sets the x-axis label, and y+lm sets the y-axis label.

```
fig = pygmt.Figure()
fig.grdimage(grid=grid, projection="R12c", cmap="geo")
fig.colorbar(frame=["a2500", "x+lElevation", "y+lm"])
fig.show()
```



Out:

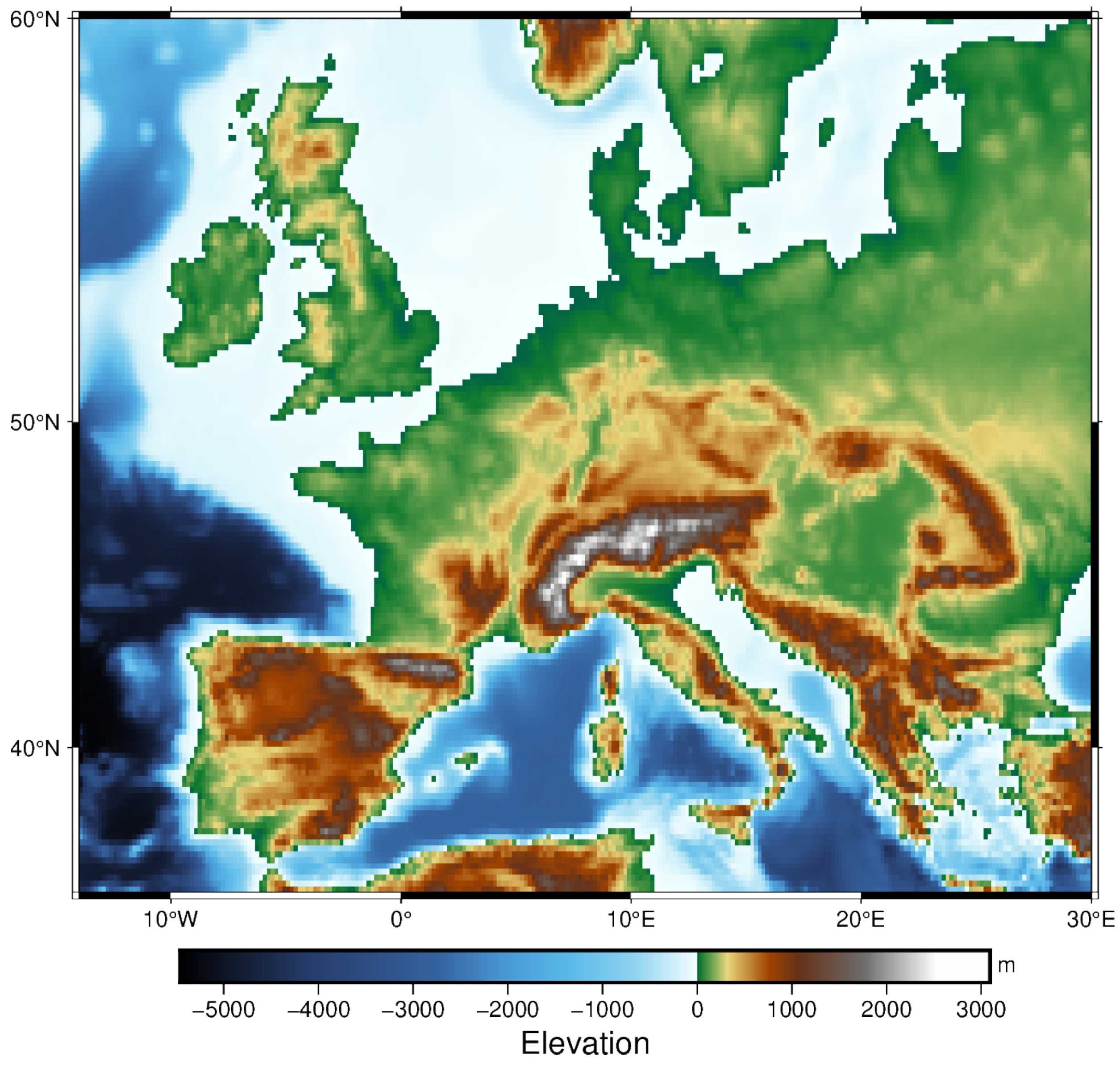
```
<IPython.core.display.Image object>
```

9.16.4 Create a region map

In addition to providing global data, the `region` argument for *`pygmt.datasets.load_earth_relief`* can be used to provide data for a specific area. The `region` argument is required for resolutions at 5 arc minutes or higher, and accepts a list (as in the example below) or a string. The geographic ranges are passed as *xmin*/*xmax*/*ymin*/*ymax*.

The example below uses data with a 10 arc minute resolution, and plots it on a 15 centimeter figure with a Mercator projection and a CPT set to *geo*. `frame="a"` is used to add a frame to the figure.

```
grid = pygmt.datasets.load_earth_relief(resolution="10m", region=[-14, 30, 35, 60])
fig = pygmt.Figure()
fig.grdimage(grid=grid, projection="M15c", frame="a", cmap="geo")
fig.colorbar(frame=["a1000", "x+lElevation", "y+lm"])
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

Total running time of the script: (0 minutes 20.008 seconds)

9.17 Creating a 3D perspective image

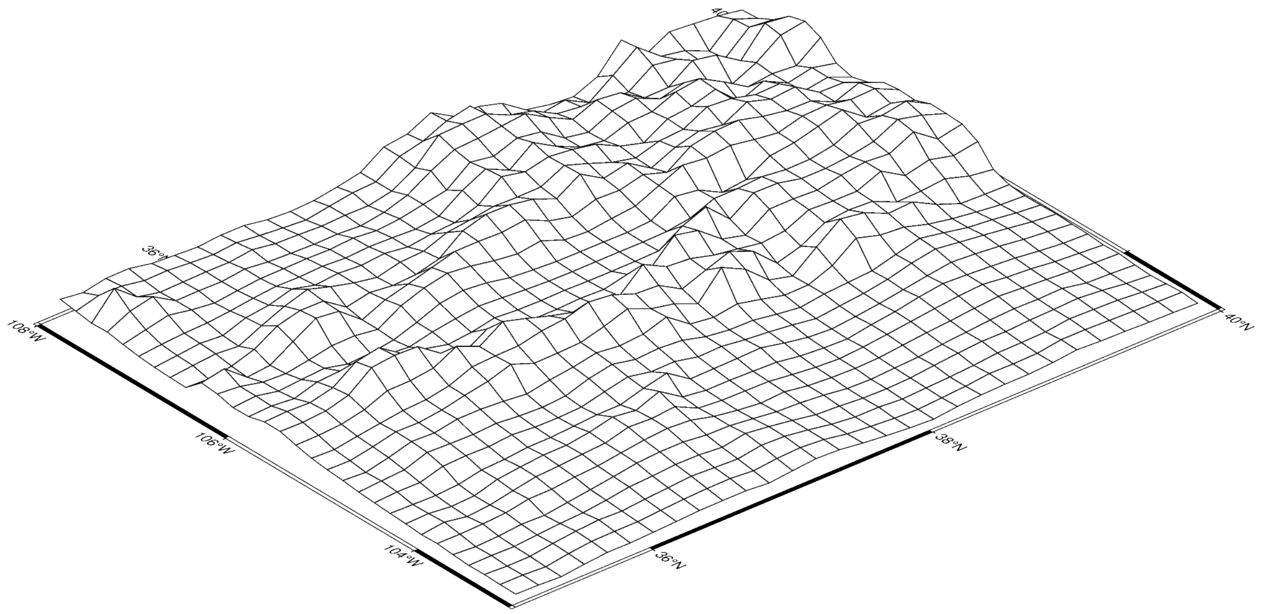
Create 3-D perspective image or surface mesh from a grid using *pygmt.Figure.grdview*.

```
import pygmt

# Load sample earth relief data
grid = pygmt.datasets.load_earth_relief(resolution="10m", region=[-108, -103, 35, 40])
```

The *pygmt.Figure.grdview* method takes the `grid` input. The `perspective` parameter changes the azimuth and elevation of the viewpoint; the default is [180, 90], which is looking directly down on the figure and north is "up". The `zsize` parameter sets how tall the three-dimensional portion appears.

The default grid surface type is *mesh plot*.

```
fig = pygmt.Figure()
fig.grdview(
    grid=grid,
    # Sets the view azimuth as 130 degrees, and the view elevation as 30
    # degrees
    perspective=[130, 30],
    # Sets the x- and y-axis labels, and annotates the west, south, and east
    # axes
    frame=["xa", "ya", "WSnE"],
    # Sets a Mercator projection on a 15-centimeter figure
    projection="M15c",
    # Sets the height of the three-dimensional relief at 1.5 centimeters
    zsize="1.5c",
)
fig.show()
```

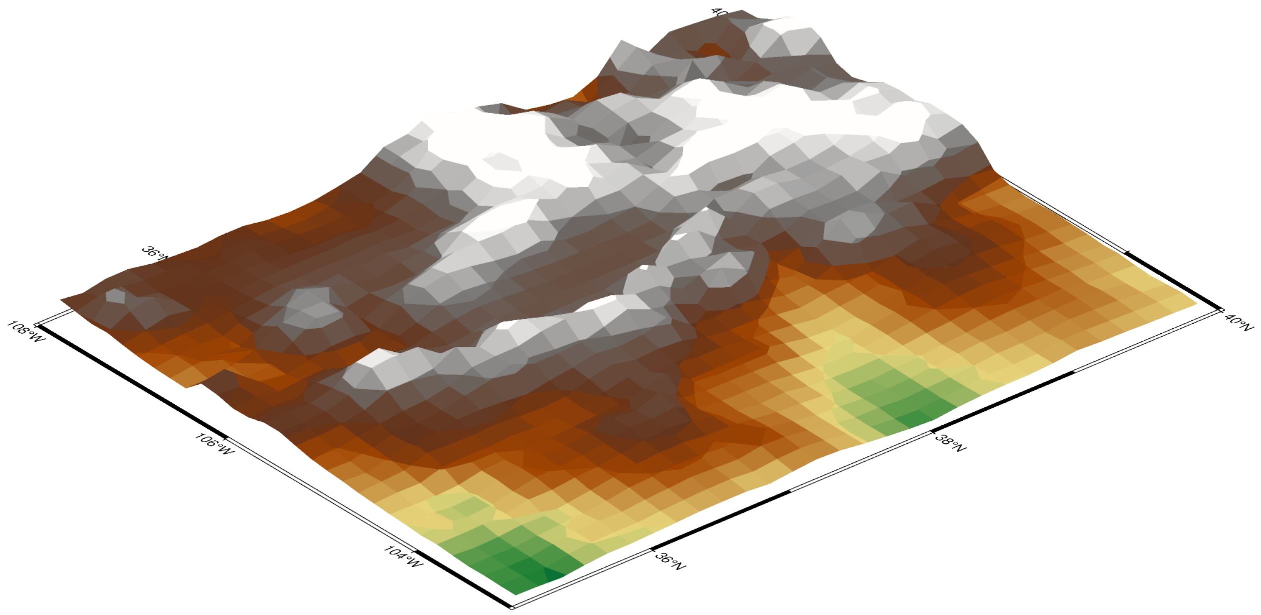


Out:

```
<IPython.core.display.Image object>
```

The grid surface type can be set with the `surftype` parameter. The default CPT is *turbo* and can be customized with the `cmap` parameter.

```
fig = pygmt.Figure()
fig.grdview(
    grid=grid,
    perspective=[130, 30],
    frame=["xa", "yaf", "WSnE"],
    projection="M15c",
    zsize="1.5c",
    # Set the surftype to "surface"
    surftype="s",
    # Set the CPT to "geo"
    cmap="geo",
)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

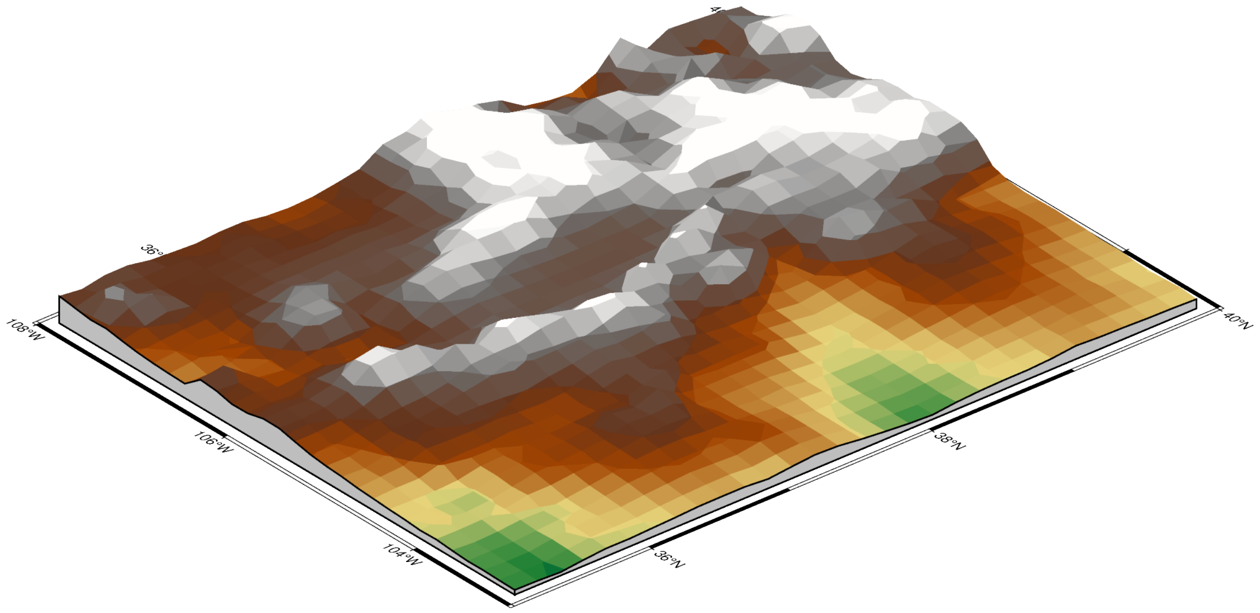
The `plane` parameter sets the elevation and color of a plane that provides a fill below the surface relief.

```
fig = pygmt.Figure()
fig.grdview(
    grid=grid,
    perspective=[130, 30],
    frame=["xa", "yaf", "WSnE"],
    projection="M15c",
    zsize="1.5c",
    surftype="s",
    cmap="geo",
    # Set the plane elevation to 1,000 meters and make the fill "gray"
    plane="1000+ggray",
```

(continues on next page)

(continued from previous page)

```
)
fig.show()
```

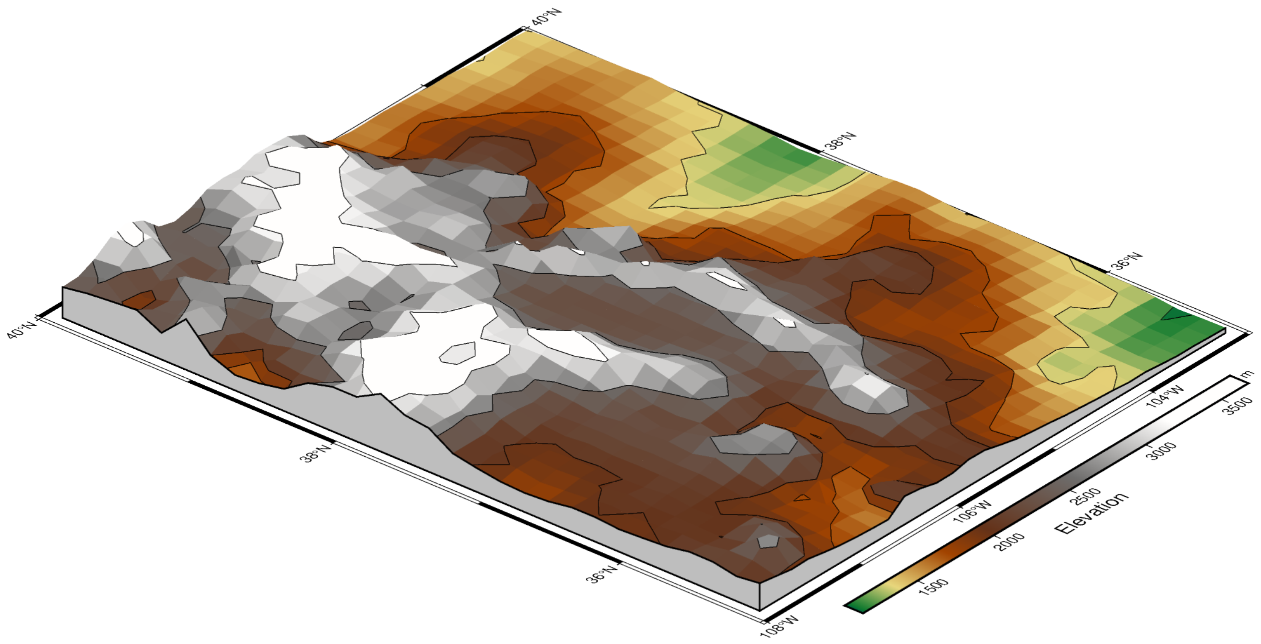


Out:

```
<IPython.core.display.Image object>
```

The `perspective` azimuth can be changed to set the direction that is "up" in the figure. The `contourpen` parameter sets the pen used to draw contour lines on the surface. *`pygmt.Figure.colorbar`* can be used to add a color bar to the figure. The `cmap` parameter does not need to be passed again. To keep the color bar's alignment similar to the figure, use `True` as the `perspective` parameter.

```
fig = pygmt.Figure()
fig.grdview(
    grid=grid,
    # Set the azimuth to -130 (230) degrees and the elevation to 30 degrees
    perspective=[-130, 30],
    frame=["xaf", "yaf", "WSnE"],
    projection="M15c",
    zsize="1.5c",
    surftype="s",
    cmap="geo",
    plane="1000+ggrey",
    # Set the contour pen thickness to "0.1p"
    contourpen="0.1p",
)
fig.colorbar(perspective=True, frame=["a500", "x+lElevation", "y+lm"])
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

Total running time of the script: (0 minutes 10.773 seconds)

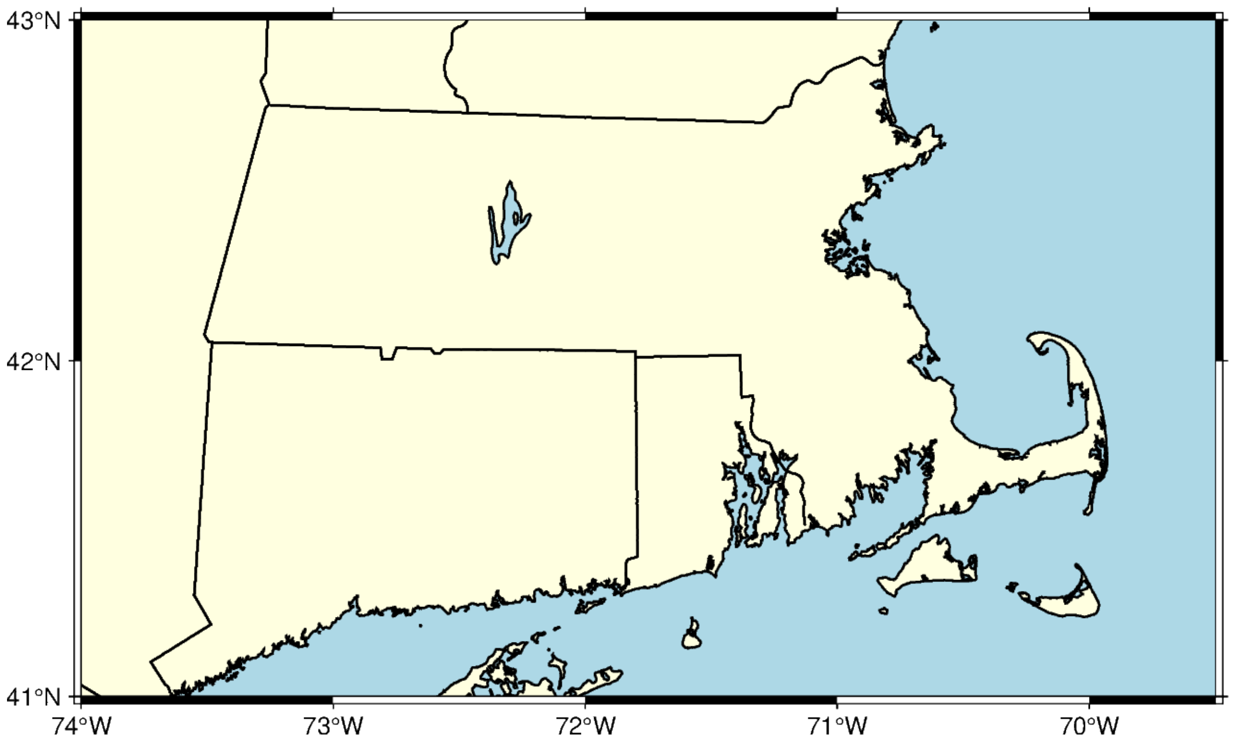
9.18 Adding an inset to the figure

To plot an inset figure inside another larger figure, we can use the *pygmt.Figure.inset* method. After a large figure has been created, call `inset` using a `with` statement, and new plot elements will be added to the inset figure instead of the larger figure.

```
import pygmt
```

Prior to creating an inset figure, a larger figure must first be plotted. In the example below, *pygmt.Figure.coast* is used to create a map of the US state of Massachusetts.

```
fig = pygmt.Figure()
fig.coast(
    region=[-74, -69.5, 41, 43],  # Set bounding box of the large figure
    borders="2/thin",  # Plot state boundaries with thin lines
    shorelines="thin",  # Plot coastline with thin lines
    projection="M15c",  # Set Mercator projection and size of 15 centimeter
    land="lightyellow",  # Color land areas light yellow
    water="lightblue",  # Color water areas light blue
    frame="a",  # Set frame with annotation and major tick spacing
)
fig.show()
```

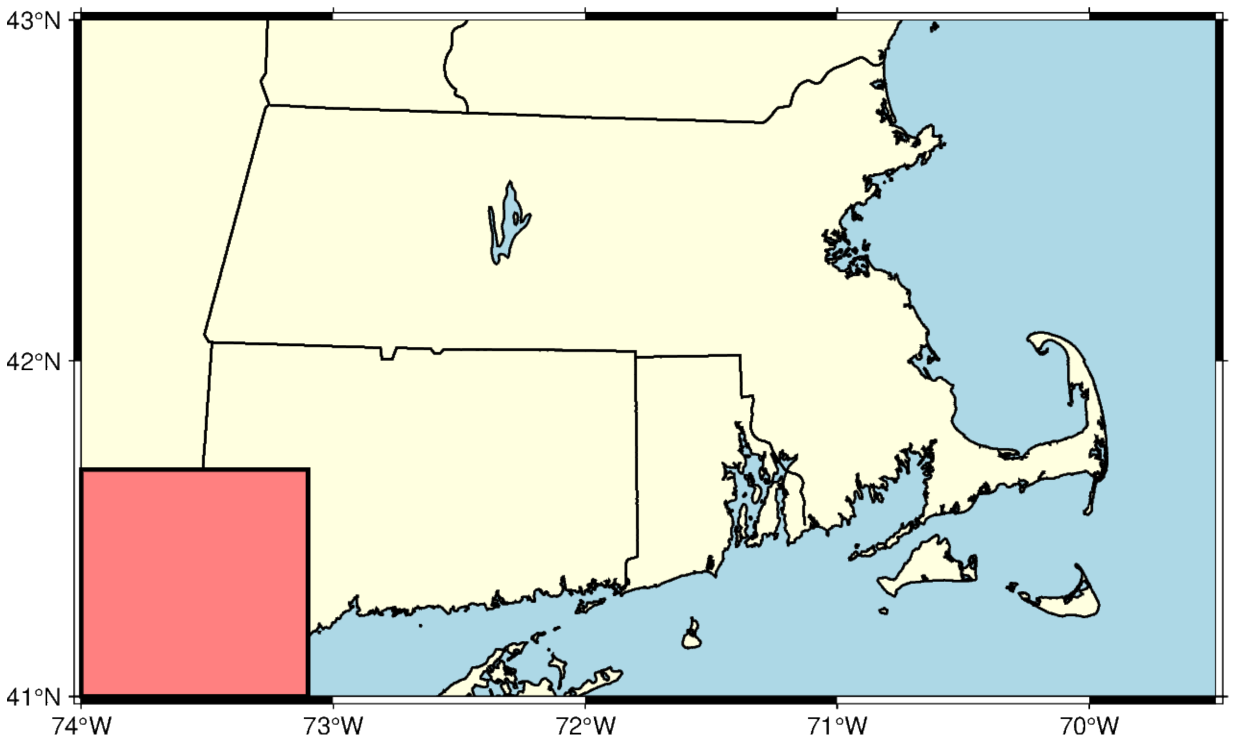


Out:

```
<IPython.core.display.Image object>
```

The *pygmt.Figure.inset* method uses a context manager, and is called using a `with` statement. The `position` parameter, including the inset width, is required to plot the inset. Using the **j** argument, the location of the inset is set to one of the 9 anchors (bottom-middle-top and left-center-right). In the example below, BL sets the inset to the bottom left. The `box` parameter can set the fill and border of the inset. In the example below, `+pblack` sets the border color to black and `+gred` sets the fill to red.

```
fig = pygmt.Figure()
fig.coast(
    region=[-74, -69.5, 41, 43],
    borders="2/thin",
    shorelines="thin",
    projection="M15c",
    land="lightyellow",
    water="lightblue",
    frame="a",
)
with fig.inset(position="jBL+w3c", box="+pblack+glightred"):
    # pass is used to exit the with statement as no plotting functions are
    # called
    pass
fig.show()
```

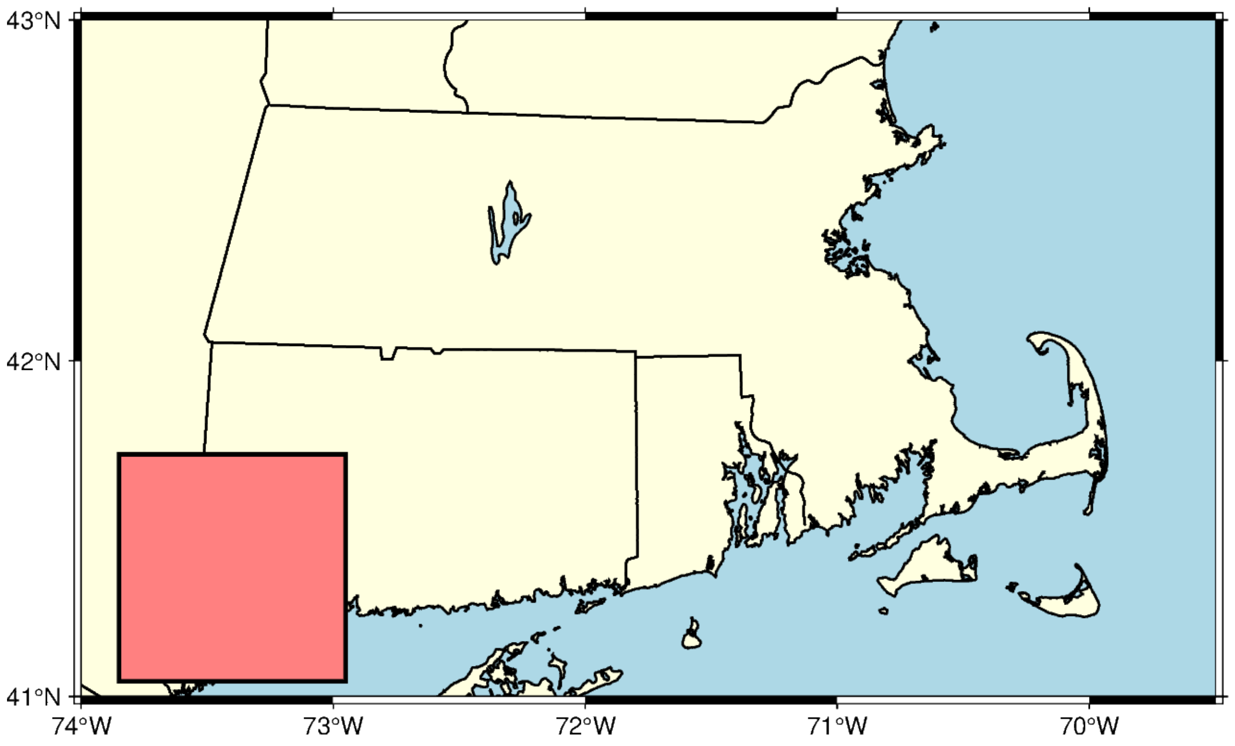


Out:

```
<IPython.core.display.Image object>
```

When using **j** to set the anchor of the inset, the default location is in contact with the nearby axis or axes. The offset of the inset can be set with **+o**, followed by the offsets along the x- and y-axis. If only one offset is passed, it is applied to both axes. Each offset can have its own unit. In the example below, the inset is shifted 0.5 centimeters on the x-axis and 0.2 centimeters on the y-axis.

```python
fig = pygmt.Figure()
fig.coast(
    region=[-74, -69.5, 41, 43],
    borders="2/thin",
    shorelines="thin",
    projection="M15c",
    land="lightyellow",
    water="lightblue",
    frame="a",
)
with fig.inset(position="jBL+w3c+o0.5c/0.2c", box="+pblack+glightred"):
    pass
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

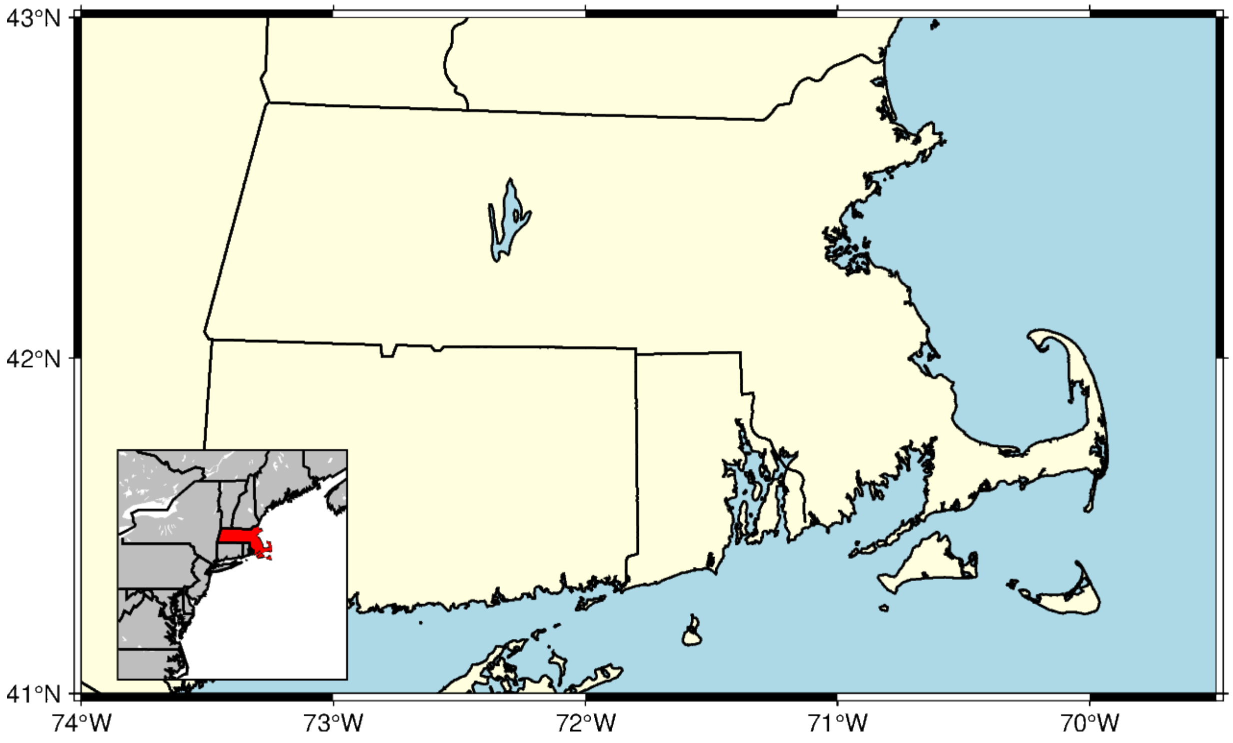
Standard plotting functions can be called from within the `inset` context manager. The example below uses *`pygmt.Figure.coast`* to plot a zoomed out map that selectively paints the state of Massachusetts to shows its location relative to other states.

```python
fig = pygmt.Figure()
fig.coast(
    region=[-74, -69.5, 41, 43],
    borders="2/thin",
    shorelines="thin",
    projection="M15c",
    land="lightyellow",
    water="lightblue",
    frame="a",
)
# This does not include an inset fill as it is covered by the inset figure
with fig.inset(position="jBL+w3c+o0.5c/0.2c", box="+pblack"):
    # Use a plotting function to create a figure inside the inset
    fig.coast(
        region=[-80, -65, 35, 50],
        projection="M3c",
        land="gray",
        borders=[1, 2],
        shorelines="1/thin",
        water="white",
        # Use dcw to selectively highlight an area
        dcw="US.MA+gred",
```

(continues on next page)

(continued from previous page)

```
    )
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

Total running time of the script: (0 minutes 11.010 seconds)

9.19 Making subplots

When you're preparing a figure for a paper, there will often be times when you'll need to put many individual plots into one large figure, and label them 'abcd'. These individual plots are called subplots.

There are two main ways to create subplots in GMT:

- Use `pygmt.Figure.shift_origin` to manually move each individual plot to the right position.
- Use `pygmt.Figure.subplot` to define the layout of the subplots.

The first method is easier to use and should handle simple cases involving a couple of subplots. For more advanced subplot layouts, however, we recommend the use of `pygmt.Figure.subplot` which offers finer grained control, and this is what the tutorial below will cover.

```
import pygmt
```

Let's start by initializing a `pygmt.Figure` instance.

```
fig = pygmt.Figure()
```

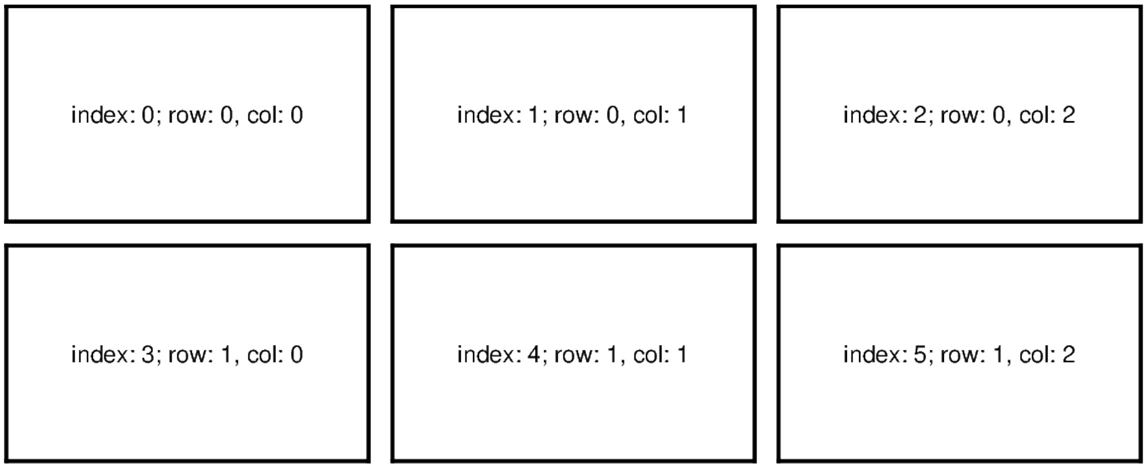
9.19.1 Define subplot layout

The *pygmt.Figure.subplot* function is used to set up the layout, size, and other attributes of the figure. It divides the whole canvas into regular grid areas with *n* rows and *m* columns. Each grid area can contain an individual subplot. For example:

```
with fig.subplot(nrows=2, ncols=3, figsize=("15c", "6c"), frame="lrtb"):
    ...
```

will define our figure to have a 2 row and 3 column grid layout. `figsize=("15c", "6c")` defines the overall size of the figure to be 15 cm wide by 6 cm high. Using `frame="lrtb"` allows us to customize the map frame for all subplots instead of setting them individually. The figure layout will look like the following:

```
with fig.subplot(nrows=2, ncols=3, figsize=("15c", "6c"), frame="lrtb"):
    for i in range(2):  # row number starting from 0
        for j in range(3):  # column number starting from 0
            index = i * 3 + j  # index number starting from 0
            with fig.set_panel(panel=index):  # sets the current panel
                fig.text(
                    position="MC",
                    text=f"index: {index}; row: {i}, col: {j}",
                    region=[0, 1, 0, 1],
                )
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

The *pygmt.Figure.set_panel* function activates a specified subplot, and all subsequent plotting functions will take place in that subplot panel. This is similar to matplotlib's `plt.sca` method. In order to specify a subplot, you will need to provide the identifier for that subplot via the `panel` parameter. Pass in either the *index* number, or a tuple/list like (*row*, *col*) to `panel`.

Note: The row and column numbering starts from 0. So for a subplot layout with N rows and M columns, row numbers will go from 0 to N-1, and column numbers will go from 0 to M-1.

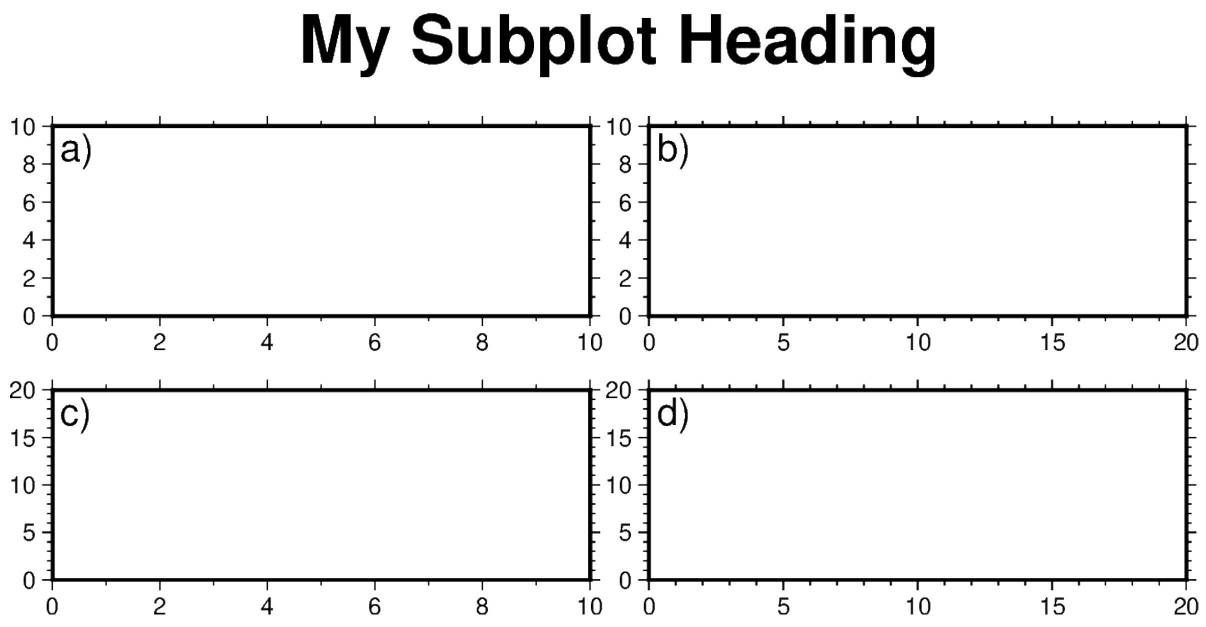
For example, to activate the subplot on the top right corner (index: 2) at *row*=0 and *col*=2, so that all subsequent plotting commands happen there, you can use the following command:

```
with fig.set_panel(panel=[0, 2]):
    ...
```

9.19.2 Making your first subplot

Next, let's use what we learned above to make a 2 row by 2 column subplot figure. We'll also pick up on some new parameters to configure our subplot.

```
fig = pygmt.Figure()
with fig.subplot(
    nrows=2,
    ncols=2,
    figsize=("15c", "6c"),
    autolabel=True,
    frame=["af", "WSne"],
    margins=["0.1c", "0.2c"],
    title="My Subplot Heading",
):
    fig.basemap(region=[0, 10, 0, 10], projection="X?", panel=[0, 0])
    fig.basemap(region=[0, 20, 0, 10], projection="X?", panel=[0, 1])
    fig.basemap(region=[0, 10, 0, 20], projection="X?", panel=[1, 0])
    fig.basemap(region=[0, 20, 0, 20], projection="X?", panel=[1, 1])
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

In this example, we define a 2-row, 2-column (2x2) subplot layout using *pygmt.Figure.subplot*. The overall figure dimensions is set to be 15 cm wide and 6 cm high (`figsize=["15c", "6c"]`). In addition, we use some optional parameters to fine-tune some details of the figure creation:

- `autolabel=True`: Each subplot is automatically labelled abcd

- `margins=["0.1c", "0.2c"]`: adjusts the space between adjacent subplots. In this case, it is set as 0.1 cm in the X direction and 0.2 cm in the Y direction.
- `title="My Subplot Heading"`: adds a title on top of the whole figure.

Notice that each subplot was set to use a linear projection `"X?"`. Usually, we need to specify the width and height of the map frame, but it is also possible to use a question mark `"?"` to let GMT decide automatically on what is the most appropriate width/height for the each subplot's map frame.

Tip: In the above example, we used the following commands to activate the four subplots explicitly one after another:

```
fig.basemap(..., panel=[0, 0])
fig.basemap(..., panel=[0, 1])
fig.basemap(..., panel=[1, 0])
fig.basemap(..., panel=[1, 1])
```

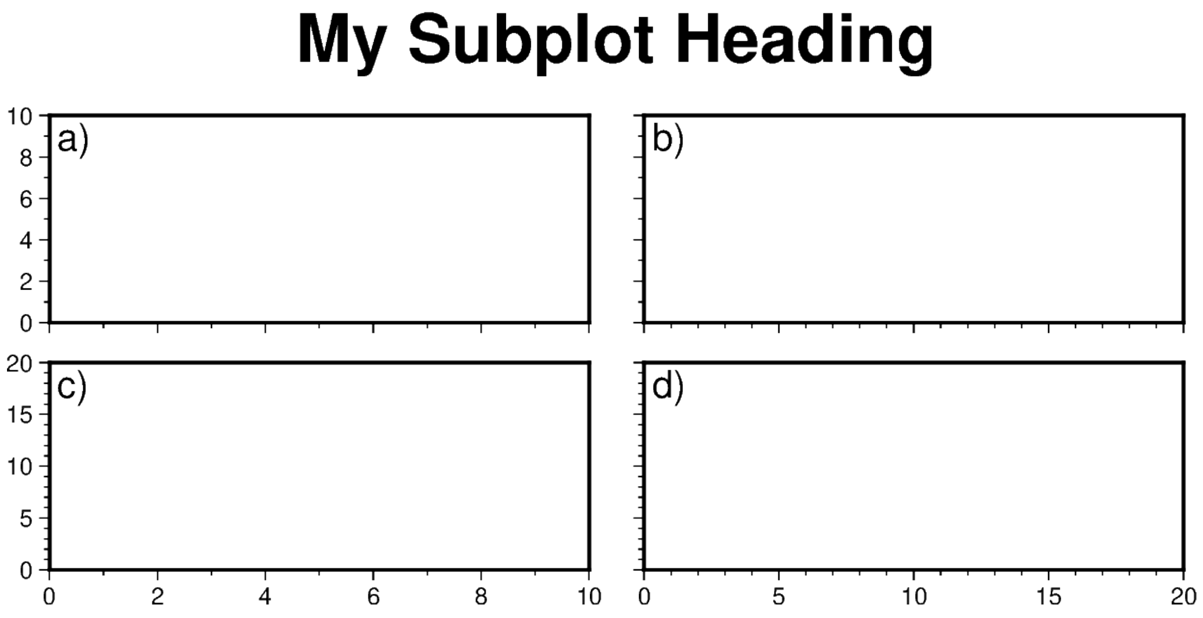
In fact, we can just use `fig.basemap(..., panel=True)` without specifying any subplot index number, and GMT will automatically activate the next subplot panel.

Note: All plotting functions (e.g. *pygmt.Figure.coast*, *pygmt.Figure.text*, etc) are able to use `panel` parameter when in subplot mode. Once a panel is activated using `panel` or *pygmt.Figure.set_panel*, subsequent plotting commands that don't set a `panel` will have their elements added to the same panel as before.

9.19.3 Shared X and Y axis labels

In the example above with the four subplots, the two subplots for each row have the same Y-axis range, and the two subplots for each column have the same X-axis range. You can use the `sharex`/`sharey` parameters to set a common X and/or Y axis between subplots.

```
fig = pygmt.Figure()
with fig.subplot(
    nrows=2,
    ncols=2,
    figsize=("15c", "6c"),  # width of 15 cm, height of 6 cm
    autolabel=True,
    margins=["0.3c", "0.2c"],  # horizontal 0.3 cm and vertical 0.2 cm margins
    title="My Subplot Heading",
    sharex="b",  # shared x-axis on the bottom side
    sharey="l",  # shared y-axis on the left side
    frame="WSrt",
):
    fig.basemap(region=[0, 10, 0, 10], projection="X?", panel=True)
    fig.basemap(region=[0, 20, 0, 10], projection="X?", panel=True)
    fig.basemap(region=[0, 10, 0, 20], projection="X?", panel=True)
    fig.basemap(region=[0, 20, 0, 20], projection="X?", panel=True)
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

`sharex="b"` indicates that subplots in a column will share the x-axis, and only the **b**ottom axis is displayed. `sharey="l"` indicates that subplots within a row will share the y-axis, and only the **l**eft axis is displayed.

Of course, instead of using the `sharex`/`sharey` option, you can also set a different `frame` for each subplot to control the axis properties individually for each subplot.

9.19.4 Advanced subplot layouts

Nested subplot are currently not supported. If you want to create more complex subplot layouts, some manual adjustments are needed.

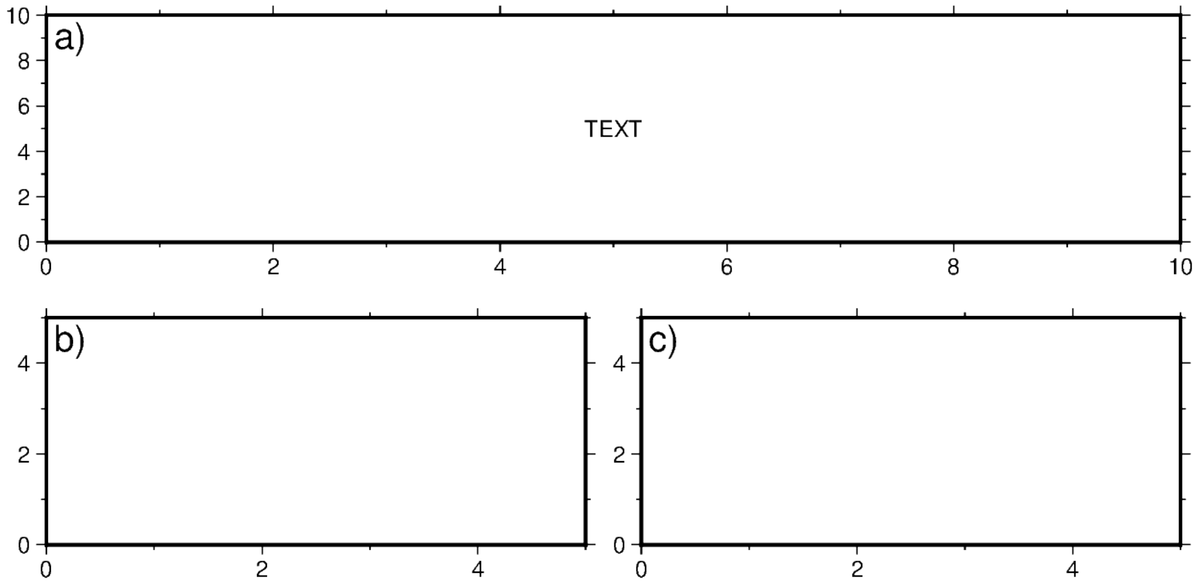
The following example draws three subplots in a 2-row, 2-column layout, with the first subplot occupying the first row.

```python
fig = pygmt.Figure()
# Bottom row, two subplots
with fig.subplot(nrows=1, ncols=2, figsize=("15c", "3c"), autolabel="b)"):
    fig.basemap(
        region=[0, 5, 0, 5], projection="X?", frame=["af", "WSne"], panel=[0, 0]
    )
    fig.basemap(
        region=[0, 5, 0, 5], projection="X?", frame=["af", "WSne"], panel=[0, 1]
    )
# Move plot origin by 1 cm above the height of the entire figure
fig.shift_origin(yshift="h+1c")
# Top row, one subplot
with fig.subplot(nrows=1, ncols=1, figsize=("15c", "3c"), autolabel="a)"):
    fig.basemap(
        region=[0, 10, 0, 10], projection="X?", frame=["af", "WSne"], panel=[0, 0]
    )
    fig.text(text="TEXT", x=5, y=5)
```

(continues on next page)

(continued from previous page)

```
fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

We start by drawing the bottom two subplots, setting `autolabel="b)"` so that the subplots are labelled 'b)' and 'c)'. Next, we use *`pygmt.Figure.shift_origin`* to move the plot origin 1 cm above the **h**eight of the entire figure that is currently plotted (i.e. the bottom row subplots). A single subplot is then plotted on the top row. You may need to adjust the `yshift` parameter to make your plot look nice. This top row uses `autolabel="a)"`, and we also plotted some text inside. Note that `projection="X?"` was used to let GMT automatically determine the size of the subplot according to the size of the subplot area.

You can also manually override the `autolabel` for each subplot using for example, `fig.set_panel(..., fixedlabel="b) Panel 2")` which would allow you to manually label a single subplot as you wish. This can be useful for adding a more descriptive subtitle to individual subplots.

Total running time of the script: (0 minutes 15.502 seconds)

9.20 Configuring PyGMT defaults

Default GMT parameters can be set globally or locally using *pygmt.config*.

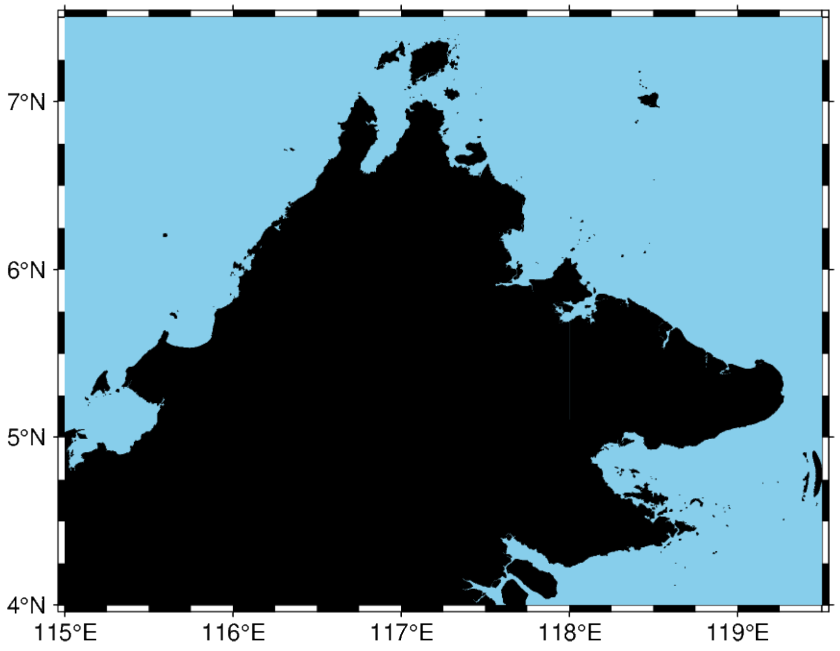
```
import pygmt
```

9.20.1 Configuring default GMT parameters

Users can override default parameters either temporarily (locally) or permanently (globally) using *pygmt.config*. The full list of default parameters that can be changed can be found at https://docs.generic-mapping-tools.org/latest/gmt.conf.html.

We demonstrate the usage of *pygmt.config* by configuring a map plot.

```
# Start with a basic figure with the default style
fig = pygmt.Figure()
fig.basemap(region=[115, 119.5, 4, 7.5], projection="M10c", frame=True)
fig.coast(land="black", water="skyblue")

fig.show()
```



Out:

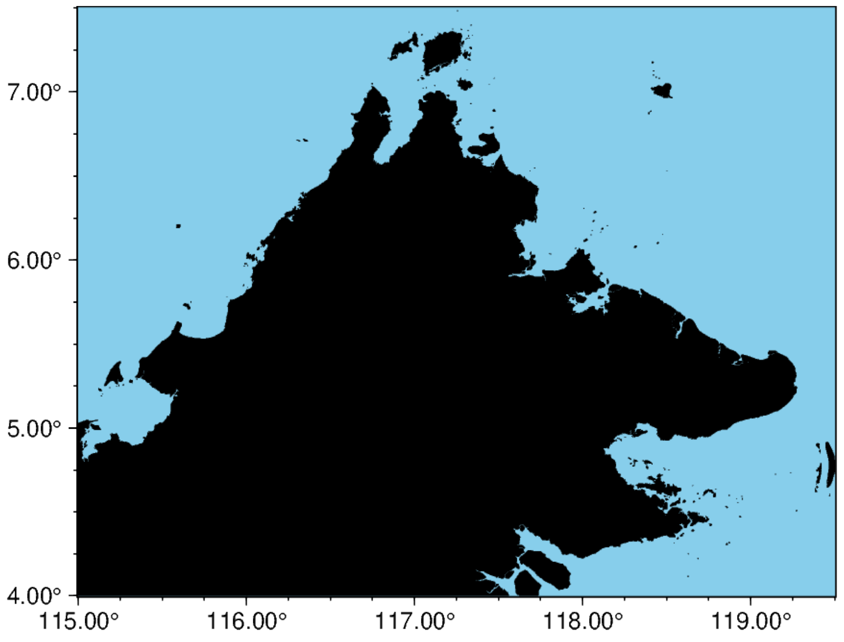
```
<IPython.core.display.Image object>
```

9.20.2 Globally overriding defaults

The `MAP_FRAME_TYPE` parameter specifies the style of map frame to use, of which there are 5 options: `fancy` (default, seen above), `fancy+`, `plain`, `graph` (which does not apply to geographical maps) and `inside`.

The `FORMAT_GEO_MAP` parameter controls the format of geographical tick annotations. The default uses degrees and minutes. Here we specify the ticks to be a decimal number of degrees.

```
fig = pygmt.Figure()

# Configuration for the 'current figure'.
pygmt.config(MAP_FRAME_TYPE="plain")
pygmt.config(FORMAT_GEO_MAP="ddd.xx")

fig.basemap(region=[115, 119.5, 4, 7.5], projection="M10c", frame=True)
fig.coast(land="black", water="skyblue")

fig.show()
```



Out:

```
<IPython.core.display.Image object>
```

9.20.3 Locally overriding defaults

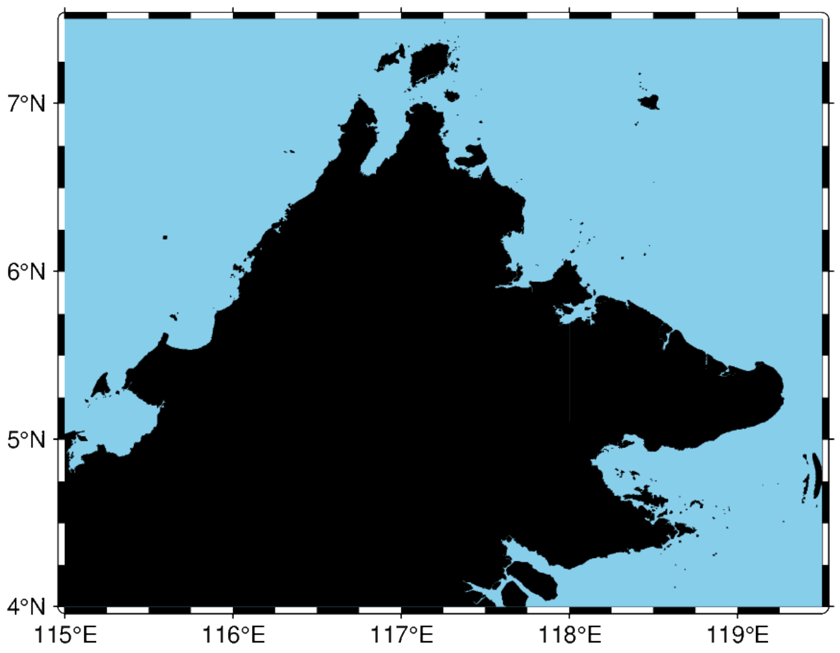
It is also possible to temporarily override the default parameters, which is very useful for limiting the scope of changes to a particular plot. *pygmt.config* is implemented as a context manager, which handles the setup and teardown of a GMT session. Python users are likely familiar with the `with open(...) as file:` snippet, which returns a `file` context manager. In this way, it can be used to override a parameter for a single command, or a sequence of commands. An application of *pygmt.config* as a context manager is shown below:

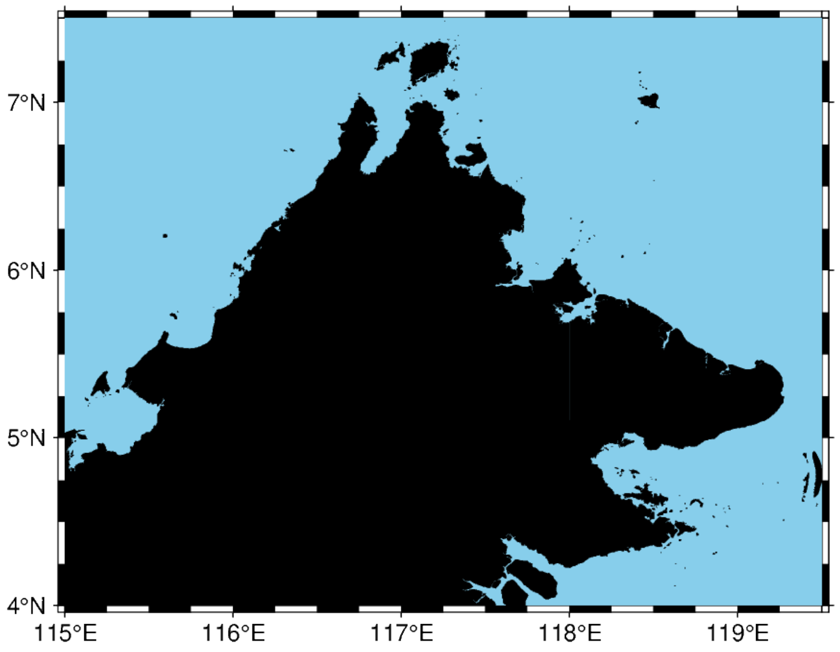
```
fig = pygmt.Figure()

```

(continues on next page)

(continued from previous page)

```
# This will have a fancy+ frame
with pygmt.config(MAP_FRAME_TYPE="fancy+"):
    fig.basemap(region=[115, 119.5, 4, 7.5], projection="M10c", frame=True)
fig.coast(land="black", water="skyblue")

# Shift plot origin down by 10cm to plot another map
fig.shift_origin(yshift="-10c")

# This figure retains the default "fancy" frame
fig.basemap(region=[115, 119.5, 4, 7.5], projection="M10c", frame=True)
fig.coast(land="black", water="skyblue")

fig.show()
```





Out:

```
<IPython.core.display.Image object>
```

Total running time of the script: (0 minutes 8.719 seconds)

9.21 API Reference

PyGMT is a library for processing geospatial and geophysical data and making publication quality maps and figures. It provides a Pythonic interface for the Generic Mapping Tools (GMT), a command-line program widely used in the Earth Sciences. Besides making GMT more accessible to new users, PyGMT aims to provide integration with the PyData ecosystem as well as support for rich display in Jupyter notebooks.

9.21.1 Main Features

Here are just a few of the things that PyGMT does well:

- Easy handling of individual types of data like Cartesian, geographic, or time-series data.
- Processing of (geo)spatial data including gridding, filtering, and masking
- Allows plotting of a large spectrum of objects on figures including lines, vectors, polygons, and symbols (pre-defined and customized)
- Generate publication-quality illustrations and make animations

9.21.2 Plotting

All plotting is handled through the `pygmt.Figure` class and its methods.

`Figure`()	A GMT figure to handle all plotting.

pygmt.Figure

class pygmt.**Figure**

A GMT figure to handle all plotting.

Use the plotting methods of this class to add elements to the figure. You can preview the figure using *`pygmt.Figure.show`* and save the figure to a file using *`pygmt.Figure.savefig`*.

Unlike traditional GMT figures, no figure file is generated until you call *`pygmt.Figure.savefig`* or *`pygmt.Figure.psconvert`*.

Examples

```
>>> fig = Figure()
>>> fig.basemap(region=[0, 360, -90, 90], projection="W7i", frame=True)
>>> fig.savefig("my-figure.png")
>>> # Make sure the figure file is generated and clean it up
>>> import os
>>> os.path.exists("my-figure.png")
True
>>> os.remove("my-figure.png")
```

The plot region can be specified through ISO country codes (for example, `'JP'` for Japan):

```
>>> fig = Figure()
>>> fig.basemap(region="JP", projection="M3i", frame=True)
```

(continues on next page)

(continued from previous page)

```
>>> # The fig.region attribute shows the WESN bounding box for the figure
>>> print(", ".join(f"{i:.2f}" for i in fig.region))
122.94, 145.82, 20.53, 45.52
```

Methods Summary

`Figure.basemap`(*[, region, projection, ...])	Plot base maps and frames for the figure.
`Figure.coast`(*[, region, projection, ...])	Plot continents, shorelines, rivers, and borders on maps
`Figure.colorbar`(*[, region, projection, ...])	Plot a gray or color scale-bar on maps.
`Figure.contour`([data, x, y, z, annotation, ...])	Contour table data by direct triangulation.
`Figure.grdcontour`(grid, *[, annotation, ...])	Convert grids or images to contours and plot them on maps.
`Figure.grdimage`(grid, *[, img_out, frame, ...])	Project and plot grids or images.
`Figure.grdview`(grid, *[, region, ...])	Create 3-D perspective image or surface mesh from a grid.
`Figure.histogram`(data, *[, horizontal, ...])	Plots a histogram, and can read data from a file or list, array, or dataframe.
`Figure.image`(imagefile, *[, region, ...])	Place images or EPS files on maps.
`Figure.inset`(*[, position, box, margin, ...])	Create an inset figure to be placed within a larger figure.
`Figure.legend`([spec, position, box, region, ...])	Plot legends on maps.
`Figure.logo`(*[, region, projection, ...])	Plot the GMT logo.
`Figure.meca`(spec, scale[, longitude, ...])	Plot focal mechanisms.
`Figure.plot`([data, x, y, size, direction, ...])	Plot lines, polygons, and symbols in 2-D.
`Figure.plot3d`([data, x, y, z, size, ...])	Plot lines, polygons, and symbols in 3-D.
`Figure.psconvert`(*[, crop, gs_option, dpi, ...])	Convert [E]PS file(s) to other formats.
`Figure.rose`([data, length, azimuth, sector, ...])	Plot windrose diagrams or polar histograms.
`Figure.savefig`(fname[, transparent, crop, ...])	Save the figure to a file.
`Figure.set_panel`([panel, fixedlabel, ...])	Set the current subplot panel to plot on.
`Figure.shift_origin`([xshift, yshift])	Shift plot origin in x and/or y directions.
`Figure.show`([dpi, width, method])	Display a preview of the figure.
`Figure.solar`([terminator, ...])	Plot day-light terminators or twilights.
`Figure.subplot`([nrows, ncols, figsize, ...])	Create multi-panel subplot figures.
`Figure.text`([textfiles, x, y, position, ...])	Plot or typeset text strings of variable size, font type, and orientation.
`Figure.velo`([data, vector, frame, cmap, ...])	Plot velocity vectors, crosses, anisotropy bars, and wedges.
`Figure.wiggle`([data, x, y, z, frame, ...])	Plot z=f(x,y) anomalies along tracks.

Examples using `pygmt.Figure`

- *Cartesian, circular, and geographic vectors*
- *Line colors with a custom CPT*
- *Line fronts*
- *Line styles*
- *Roads*
- *Vector heads and tails*

- *Wiggle along tracks*
- *Basic geometric symbols*
- *Color points by categories*
- *Custom symbols*
- *Datetime inputs*
- *Multi-parameter symbols*
- *Points*
- *Points with varying transparency*
- *Scatter plots with a legend*
- *Text symbols*
- *Calculating grid gradient and radiance*
- *Clipping grid values*
- *Contours*
- *Create 'wet-dry' mask grid*
- *Images on figures*
- *Sampling along tracks*
- *3D Scatter plots*
- *Plotting a surface*
- *Focal mechanisms*
- *Velocity arrows and confidence ellipses*
- *Double Y axes graph*
- *Histogram*
- *Rose diagram*
- *Colorbar*
- *Day-night terminator line and twilights*
- *Inset*
- *Inset map showing a rectangular region*
- *Legend*
- *Logo*
- *Multiple colormaps*
- *Adding an inset to the figure*
- *Coastlines and borders*
- *Configuring PyGMT defaults*
- *Creating a 3D perspective image*
- *Creating a map with contour lines*
- *Frames, ticks, titles, and labels*

- *Making subplots*
- *Making your first figure*
- *Plotting Earth relief*
- *Plotting data points*
- *Plotting datetime charts*
- *Plotting lines*
- *Plotting text*
- *Plotting vectors*
- *Setting the region*
- *Azimuthal Equidistant*
- *General Perspective*
- *General Stereographic*
- *Gnomonic*
- *Lambert Azimuthal Equal Area*
- *Orthographic*
- *Albers Conic Equal Area*
- *Equidistant conic*
- *Lambert Conic Conformal Projection*
- *Polyconic Projection*
- *Cassini Cylindrical*
- *Cylindrical Stereographic*
- *Cylindrical equal-area*
- *Cylindrical equidistant*
- *Mercator*
- *Miller cylindrical*
- *Oblique Mercator*
- *Oblique Mercator*
- *Oblique Mercator*
- *Transverse Mercator*
- *Universal Transverse Mercator*
- *Eckert IV*
- *Eckert VI*
- *Hammer*
- *Mollweide*
- *Robinson*
- *Sinusoidal*

- *Van der Grinten*
- *Winkel Tripel*
- *Cartesian linear*
- *Cartesian logarithmic*
- *Cartesian power*
- *Polar*

Plotting data and laying out the map:

`Figure.basemap`(*[, region, projection, ...])	Plot base maps and frames for the figure.
`Figure.coast`(*[, region, projection, ...])	Plot continents, shorelines, rivers, and borders on maps
`Figure.colorbar`(*[, region, projection, ...])	Plot a gray or color scale-bar on maps.
`Figure.contour`([data, x, y, z, annotation, ...])	Contour table data by direct triangulation.
`Figure.grdcontour`(grid, *[, annotation, ...])	Convert grids or images to contours and plot them on maps.
`Figure.grdimage`(grid, *[, img_out, frame, ...])	Project and plot grids or images.
`Figure.grdview`(grid, *[, region, ...])	Create 3-D perspective image or surface mesh from a grid.
`Figure.histogram`(data, *[, horizontal, ...])	Plots a histogram, and can read data from a file or list, array, or dataframe.
`Figure.image`(imagefile, *[, region, ...])	Place images or EPS files on maps.
`Figure.inset`(*[, position, box, margin, ...])	Create an inset figure to be placed within a larger figure.
`Figure.legend`([spec, position, box, region, ...])	Plot legends on maps.
`Figure.logo`(*[, region, projection, ...])	Plot the GMT logo.
`Figure.meca`(spec, scale[, longitude, ...])	Plot focal mechanisms.
`Figure.plot`([data, x, y, size, direction, ...])	Plot lines, polygons, and symbols in 2-D.
`Figure.plot3d`([data, x, y, z, size, ...])	Plot lines, polygons, and symbols in 3-D.
`Figure.rose`([data, length, azimuth, sector, ...])	Plot windrose diagrams or polar histograms.
`Figure.set_panel`([panel, fixedlabel, ...])	Set the current subplot panel to plot on.
`Figure.shift_origin`([xshift, yshift])	Shift plot origin in x and/or y directions.
`Figure.solar`([terminator, ...])	Plot day-light terminators or twilights.
`Figure.subplot`([nrows, ncols, figsize, ...])	Create multi-panel subplot figures.
`Figure.text`([textfiles, x, y, position, ...])	Plot or typeset text strings of variable size, font type, and orientation.
`Figure.velo`([data, vector, frame, cmap, ...])	Plot velocity vectors, crosses, anisotropy bars, and wedges.
`Figure.wiggle`([data, x, y, z, frame, ...])	Plot z=f(x,y) anomalies along tracks.

pygmt.Figure.basemap

`Figure.`**`basemap`**(*, *region=None*, *projection=None*, *zscale=None*, *zsize=None*, *frame=None*, *map_scale=None*, *rose=None*, *compass=None*, *timestamp=None*, *verbose=None*, *xshift=None*, *yshift=None*, *panel=None*, *coltypes=None*, *perspective=None*, *transparency=None*, ***kwargs*)

Plot base maps and frames for the figure.

Creates a basic or fancy basemap with axes, fill, and titles. Several map projections are available, and the user may specify separate tick-mark intervals for boundary annotation, ticking, and [optionally] gridlines. A simple map scale or directional rose may also be plotted.

At least one of the parameters `frame`, `map_scale`, `rose` or `compass` must be specified.

Full option list at https://docs.generic-mapping-tools.org/latest/basemap.html

Aliases:

- B = frame
- J = projection
- JZ = zsize
- Jz = zscale
- L = map_scale
- R = region
- Td = rose
- Tm = compass
- U = timestamp
- V = verbose
- X = xshift
- Y = yshift
- c = panel
- f = coltypes
- p = perspective
- t = transparency

Parameters

- **projection** (*str*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- **zscale/zsize** (*float or str*) – Set z-axis scaling or z-axis size.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest. *Required if this is the first plot command.*
- **frame** (*bool or str or list*) – Set map boundary *frame and axes attributes*.
- **map_scale** (*str*) – [**g**|**j**|**J**|**n**|**x**]*refpoint*+**w***length*. Draws a simple map scale centered on the reference point specified.
- **rose** (*str*) – Draws a map directional rose on the map at the location defined by the reference and anchor points.
- **compass** (*str*) – Draws a map magnetic rose on the map at the location defined by the reference and anchor points
- **timestamp** (*bool or str*) – Draw GMT time stamp logo on plot.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);

  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages

- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.
- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.
- **panel** (*bool or int or list*) – [*row,col*|*index*]. Select a specific subplot panel. Only allowed when in subplot mode. Use `panel=True` to advance to the next panel in the selected order. Instead of *row,col* you may also give a scalar value *index* which depends on the order you set via `autolabel` when the subplot was defined. **Note**: *row*, *col*, and *index* all start at 0.
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **perspective** (*list or str*) – [**x**|**y**|**z**]*azim*[/*elev*[/*zlevel*]][**+w***lon0*/*lat0*[/*z0*]][**+v***x0*/*y0*]. Select perspective view and set the azimuth and elevation angle of the viewpoint. Default is [180, 90]. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#perspective-full.
- **transparency** (*int or float*) – Set transparency level, in [0-100] percent range. Default is 0, i.e., opaque. Only visible when PDF or raster format output is selected. Only the PNG format selection adds a transparency layer in the image (for further processing).

Examples using `pygmt.Figure.basemap`

- *Line colors with a custom CPT*
- *Line fronts*
- *Line styles*
- *Roads*
- *Vector heads and tails*
- *Wiggle along tracks*
- *Basic geometric symbols*
- *Color points by categories*
- *Custom symbols*
- *Datetime inputs*
- *Multi-parameter symbols*
- *Points*
- *Points with varying transparency*
- *Scatter plots with a legend*
- *Text symbols*
- *Clipping grid values*
- *Create 'wet-dry' mask grid*

- *Images on figures*
- *Sampling along tracks*
- *Double Y axes graph*
- *Colorbar*
- *Inset map showing a rectangular region*
- *Legend*
- *Logo*
- *Coastlines and borders*
- *Configuring PyGMT defaults*
- *Frames, ticks, titles, and labels*
- *Making subplots*
- *Making your first figure*
- *Plotting data points*
- *Plotting text*
- *Polar*

pygmt.Figure.coast

`Figure.coast`(**, region=None, projection=None, area_thresh=None, lakes=None, frame=None, resolution=None, dcw=None, rivers=None, map_scale=None, borders=None, shorelines=None, land=None, water=None, timestamp=None, verbose=None, xshift=None, yshift=None, panel=None, perspective=None, transparency=None, **kwargs*)

Plot continents, shorelines, rivers, and borders on maps

Plots grayshaded, colored, or textured land-masses [or water-masses] on maps and [optionally] draws coastlines, rivers, and political boundaries. Alternatively, it can (1) issue clip paths that will contain all land or all water areas, or (2) dump the data to an ASCII table. The data files come in 5 different resolutions: (**f**)ull, (**h**)igh, (**i**)ntermediate, (**l**)ow, and (**c**)rude. The full resolution files amount to more than 55 Mb of data and provide great detail; for maps of larger geographical extent it is more economical to use one of the other resolutions. If the user selects to paint the land-areas and does not specify fill of water-areas then the latter will be transparent (i.e., earlier graphics drawn in those areas will not be overwritten). Likewise, if the water-areas are painted and no land fill is set then the land-areas will be transparent.

A map projection must be supplied.

Full option list at https://docs.generic-mapping-tools.org/latest/coast.html

Aliases:

- A = area_thresh
- B = frame
- C = lakes
- D = resolution
- E = dcw
- G = land
- I = rivers

- J = projection
- L = map_scale
- N = borders
- R = region
- S = water
- U = timestamp
- V = verbose
- W = shorelines
- X = xshift
- Y = yshift
- c = panel
- p = perspective
- t = transparency

Parameters

- **`projection`** (*`str`*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- **`region`** (*`str` or `list`*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest. *Required if this is the first plot command.*
- **`area_thresh`** (*`int` or `float` or `str`*) – *min_area*[/*min_level*/*max_level*][**+a**[**g**|**i**][**s**|**S**]][**+l**|**r**][**+p***percent*]. Features with an area smaller than *min_area* in km^2 or of hierarchical level that is lower than *min_level* or higher than *max_level* will not be plotted [Default is 0/0/4 (all features)].
- **`frame`** (*`bool` or `str` or `list`*) – Set map boundary *frame and axes attributes*.
- **`lakes`** (*`str` or `list`*) – *fill*[**+l**|**+r**]. Set the shade, color, or pattern for lakes and river-lakes. The default is the fill chosen for wet areas set by the `water` parameter. Optionally, specify separate fills by appending **+l** for lakes or **+r** for river-lakes, and passing multiple strings in a list.
- **`resolution`** (*`str`*) – **f**|**h**|**i**|**l**|**c**. Selects the resolution of the data set to: (**f**)ull, (**h**)igh, (**i**)ntermediate, (**l**)ow, and (**c**)rude.
- **`land`** (*`str`*) – Select filling or clipping of "dry" areas.
- **`rivers`** (*`int` or `str` or `list`*) – *river*[/*pen*]. Draw rivers. Specify the type of rivers and [optionally] append pen attributes [Default pen is width = default, color = black, style = solid].

  Choose from the list of river types below; pass a list to `rivers` to use multiple arguments.

  0 = Double-lined rivers (river-lakes)

  1 = Permanent major rivers

  2 = Additional major rivers

  3 = Additional rivers

  4 = Minor rivers

  5 = Intermittent rivers - major

  6 = Intermittent rivers - additional

7 = Intermittent rivers - minor

8 = Major canals

9 = Minor canals

10 = Irrigation canals

You can also choose from several preconfigured river groups:

a = All rivers and canals (0-10)

A = All rivers and canals except river-lakes (1-10)

r = All permanent rivers (0-4)

R = All permanent rivers except river-lakes (1-4)

i = All intermittent rivers (5-7)

c = All canals (8-10)

- **map_scale** (*str*) – [**g**|**j**|**J**|**n**|**x**]*refpoint*+**w***length*. Draws a simple map scale centered on the reference point specified.
- **borders** (*int or str or list*) – *border*[/*pen*]. Draw political boundaries. Specify the type of boundary and [optionally] append pen attributes [Default pen is width = default, color = black, style = solid].

  Choose from the list of boundaries below. Pass a list to `borders` to use multiple arguments.

  1 = National boundaries

  2 = State boundaries within the Americas

  3 = Marine boundaries

  a = All boundaries (1-3)
- **water** (*str*) – Select filling or clipping of "wet" areas.
- **timestamp** (*bool or str*) – Draw GMT time stamp logo on plot.
- **shorelines** (*int or str or list*) – [*level*/]*pen*. Draw shorelines [Default is no shorelines]. Append pen attributes [Default is width = default, color = black, style = solid] which apply to all four levels. To set the pen for a single level, pass a string with *level*/*pen*, where level is 1-4 and represent coastline, lakeshore, island-in-lake shore, and lake-in-island-in-lake shore. Pass a list of *level*/*pen* strings to `shorelines` to set multiple levels. When specific level pens are set, those not listed will not be drawn.
- **dcw** (*str or list*) – *code1,code2,…* [**+l**|**L**][**+g***fill*] [**+p***pen*][**+z**]. Select painting or dumping country polygons from the Digital Chart of the World. Append one or more comma-separated countries using the 2-character ISO 3166-1 alpha-2 convention. To select a state of a country (if available), append *.state*, (e.g, US.TX for Texas). To specify a whole continent, prepend **=** to any of the continent codes (e.g. =EU for Europe). Append **+p***pen* to draw polygon outlines (default is no outline) and **+g***fill* to fill them (default is no fill). Append **+l**|**+L** to =*continent* to only list countries in that continent; repeat if more than one continent is requested.
- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.
- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.

- **`panel`** (*`bool` or `int` or `list`*) – [*row,col*|*index*]. Select a specific subplot panel. Only allowed when in subplot mode. Use `panel=True` to advance to the next panel in the selected order. Instead of *row,col* you may also give a scalar value *index* which depends on the order you set via `autolabel` when the subplot was defined. **Note**: *row*, *col*, and *index* all start at 0.
- **`perspective`** (*`list` or `str`*) – [**x**|**y**|**z**]*azim*[/*elev*[/*zlevel*]][**+w***lon0*/*lat0*[/*z0*]][**+v***x0*/*y0*]. Select perspective view and set the azimuth and elevation angle of the viewpoint. Default is [180, 90]. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#perspective-full.
- **`transparency`** (*`int` or `float`*) – Set transparency level, in [0-100] percent range. Default is 0, i.e., opaque. Only visible when PDF or raster format output is selected. Only the PNG format selection adds a transparency layer in the image (for further processing).
- **`verbose`** (*`bool` or `str`*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages

Examples using `pygmt.Figure.coast`

- *Cartesian, circular, and geographic vectors*
- *Roads*
- *Sampling along tracks*
- *Focal mechanisms*
- *Day-night terminator line and twilights*
- *Inset*
- *Inset map showing a rectangular region*
- *Adding an inset to the figure*
- *Coastlines and borders*
- *Configuring PyGMT defaults*
- *Frames, ticks, titles, and labels*
- *Making subplots*
- *Making your first figure*
- *Plotting data points*
- *Plotting text*
- *Plotting vectors*

- *Setting the region*
- *Azimuthal Equidistant*
- *General Perspective*
- *General Stereographic*
- *Gnomonic*
- *Lambert Azimuthal Equal Area*
- *Orthographic*
- *Albers Conic Equal Area*
- *Equidistant conic*
- *Lambert Conic Conformal Projection*
- *Polyconic Projection*
- *Cassini Cylindrical*
- *Cylindrical Stereographic*
- *Cylindrical equal-area*
- *Cylindrical equidistant*
- *Mercator*
- *Miller cylindrical*
- *Oblique Mercator*
- *Oblique Mercator*
- *Oblique Mercator*
- *Transverse Mercator*
- *Universal Transverse Mercator*
- *Eckert IV*
- *Eckert VI*
- *Hammer*
- *Mollweide*
- *Robinson*
- *Sinusoidal*
- *Van der Grinten*
- *Winkel Tripel*

pygmt.Figure.colorbar

`Figure.colorbar`(*, *region=None*, *projection=None*, *frame=None*, *cmap=None*, *position=None*, *box=None*, *truncate=None*, *shading=None*, *scale=None*, *verbose=None*, *xshift=None*, *yshift=None*, *panel=None*, *perspective=None*, *transparency=None*, ***kwargs*)

Plot a gray or color scale-bar on maps.

Both horizontal and vertical scales are supported. For CPTs with gradational colors (i.e., the lower and upper boundary of an interval have different colors) we will interpolate to give a continuous scale. Variations in intensity due to shading/illumination may be displayed by setting the `shading` parameter. Colors may be spaced according to a linear scale, all be equal size, or by providing a file with individual tile widths.

Full option list at https://docs.generic-mapping-tools.org/latest/colorbar.html

Aliases:

- B = frame
- C = cmap
- D = position
- F = box
- G = truncate
- I = shading
- J = projection
- R = region
- V = verbose
- W = scale
- X = xshift
- Y = yshift
- c = panel
- p = perspective
- t = transparency

Parameters

- **frame** (*str or list*) – Set color bar boundary frame, labels, and axes attributes.
- **cmap** (*str*) – File name of a CPT file or a series of comma-separated colors (e.g., *color1*,*color2*,*color3*) to build a linear continuous CPT from those colors automatically.
- **position** (*str*) – [**g**|**j**|**J**|**n**|**x**]*refpoint*[**+w***length*[/*width*]][**+e**[**b**|**f**][*length*]][**+h**|**v**][**+j***justify*][**+m**[**a**|**c**|**l**|**u**]][**+n**[*txt*]][**+od**
Defines the reference point on the map for the color scale using one of four coordinate systems: (1) Use **g** for map (user) coordinates, (2) use **j** or **J** for setting *refpoint* via a 2-char justification code that refers to the (invisible) map domain rectangle, (3) use **n** for normalized (0-1) coordinates, or (4) use **x** for plot coordinates (inches, cm, etc.). All but **x** requires both `region` and `projection` to be specified. Append **+w** followed by the length and width of the color bar. If width is not specified then it is set to 4% of the given length. Give a negative length to reverse the scale bar. Append **+h** to get a horizontal scale [Default is vertical (**+v**)]. By default, the anchor point on the scale is assumed to be the bottom left corner (**BL**), but this can be changed by appending **+j** followed by a 2-char justification code *justify*.

- **box** (*bool or str*) – [**+c***clearances*][**+g***fill*][**+i**[[*gap*/]*pen*]][**+p**[*pen*]][**+r**[*radius*]][**+s**[[*dx*/*dy*/][*shade*]]]. If set to `True`, draws a rectangular border around the color scale. Alternatively, specify a different pen with **+p***pen*. Add **+g***fill* to fill the scale panel [default is no fill]. Append **+c***clearance* where *clearance* is either gap, xgap/ygap, or lgap/rgap/bgap/tgap where these items are uniform, separate in x- and y-direction, or individual side spacings between scale and border. Append **+i** to draw a secondary, inner border as well. We use a uniform gap between borders of 2p and the MAP_DEFAULTS_PEN unless other values are specified. Append **+r** to draw rounded rectangular borders instead, with a 6p corner radius. You can override this radius by appending another value. Finally, append **+s** to draw an offset background shaded region. Here, *dx/dy* indicates the shift relative to the foreground frame [4p/-4p] and shade sets the fill style to use for shading [default is gray50].
- **truncate** (*list or str*) – *zlo*/*zhi*. Truncate the incoming CPT so that the lowest and highest z-levels are to *zlo* and *zhi*. If one of these equal NaN then we leave that end of the CPT alone. The truncation takes place before the plotting.
- **scale** (*float*) – Multiply all z-values in the CPT by the provided scale. By default the CPT is used as is.
- **shading** (*str or list or bool*) – Add illumination effects. Passing a single numerical value sets the range of intensities from -value to +value. If not specified, 1 is used. Alternatively, set `shading=[low, high]` to specify an asymmetric intensity range from *low* to *high*. [Default is no illumination].
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.
- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.
- **panel** (*bool or int or list*) – [*row,col*|*index*]. Select a specific subplot panel. Only allowed when in subplot mode. Use `panel=True` to advance to the next panel in the selected order. Instead of *row,col* you may also give a scalar value *index* which depends on the order you set via `autolabel` when the subplot was defined. **Note**: *row*, *col*, and *index* all start at 0.
- **perspective** (*list or str*) – [**x**|**y**|**z**]*azim*[/*elev*[/*zlevel*]][**+w***lon0*/*lat0*[/*z0*]][**+v***x0*/*y0*]. Select perspective view and set the azimuth and elevation angle of the viewpoint. Default is [180, 90]. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#perspective-full.
- **transparency** (*int or float*) – Set transparency level, in [0-100] percent range. Default is 0, i.e., opaque. Only visible when PDF or raster format output is selected. Only the PNG format selection adds a transparency layer in the image (for further processing).

Examples using `pygmt.Figure.colorbar`

- *Line colors with a custom CPT*
- *Color points by categories*
- *Calculating grid gradient and radiance*
- *Clipping grid values*
- *Create 'wet-dry' mask grid*
- *3D Scatter plots*
- *Colorbar*
- *Multiple colormaps*
- *Creating a 3D perspective image*
- *Plotting Earth relief*
- *Plotting data points*

pygmt.Figure.contour

`Figure.contour`(*data=None*, *x=None*, *y=None*, *z=None*, ***, *annotation=None*, *frame=None*, *levels=None*, *label_placement=None*, *projection=None*, *triangular_mesh_pen=None*, *no_clip=None*, *region=None*, *skip=None*, *timestamp=None*, *verbose=None*, *pen=None*, *xshift=None*, *yshift=None*, *binary=None*, *panel=None*, *nodata=None*, *find=None*, *coltypes=None*, *header=None*, *incols=None*, *label=None*, *perspective=None*, *transparency=None*, ***kwargs*)

Contour table data by direct triangulation.

Takes a matrix, (x,y,z) pairs, or a file name as input and plots lines, polygons, or symbols at those locations on a map.

Must provide either `data` or `x`/`y`/`z`.

Full option list at https://docs.generic-mapping-tools.org/latest/contour.html

Aliases:

- A = annotation
- B = frame
- C = levels
- G = label_placement
- J = projection
- L = triangular_mesh_pen
- N = no_clip
- R = region
- S = skip
- U = timestamp
- V = verbose
- W = pen

- X = xshift
- Y = yshift
- b = binary
- c = panel
- d = nodata
- e = find
- f = coltypes
- h = header
- i = incols
- l = label
- p = perspective
- t = transparency

Parameters

- **`data`** (*`str` or `numpy.ndarray` or `pandas.DataFrame` or `xarray.Dataset` or `geopandas.GeoDataFrame`*) – Pass in (x, y, z) or (longitude, latitude, elevation) values by providing a file name to an ASCII data table, a 2D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data
- **`x/y/z`** (*`1d arrays`*) – Arrays of x and y coordinates and values z of the data points.
- **`projection`** (*`str`*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- **`region`** (*`str` or `list`*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **`annotation`** (*`str` or `int`*) – Specify or disable annotated contour levels, modifies annotated contours specified in `interval`.
  – Specify a fixed annotation interval *annot_int* or a single annotation level +*annot_int*.
- **`frame`** (*`bool` or `str` or `list`*) – Set map boundary *frame and axes attributes*.
- **`levels`** (*`str` or `int`*) – Specify the contour lines to generate.
  – The filename of a CPT file where the color boundaries will be used as contour levels.
  – The filename of a 2 (or 3) column file containing the contour levels (col 1), (**C**)ontour or (**A**)nnotate (col 2), and optional angle (col 3)
  – A fixed contour interval *cont_int* or a single contour with +*cont_int*
- **`D`** (*`str`*) – Dump contour coordinates.
- **`E`** (*`str`*) – Network information.
- **`label_placement`** (*`str`*) – [**d**|**f**|**n**|**l**|**L**|**x**|**X**]*args*. Control the placement of labels along the quoted lines. It supports five controlling algorithms. See https://docs.generic-mapping-tools.org/latest/contour.html#g for details.
- **`I`** (*`bool`*) – Color the triangles using CPT.
- **`triangular_mesh_pen`** (*`str`*) – Pen to draw the underlying triangulation [Default is none].

- **no_clip** (*bool*) – Do NOT clip contours or image at the boundaries [Default will clip to fit inside region].
- **Q** (*float or str*) – [*cut*][**+z**]. Do not draw contours with less than cut number of points.
- **skip** (*bool or str*) – [**p**|**t**]. Skip input points outside region.
- **pen** (*str*) – Set pen attributes for lines or the outline of symbols.
- **label** (*str*) – Add a legend entry for the contour being plotted. Normally, the annotated contour is selected for the legend. You can select the regular contour instead, or both of them, by considering the label to be of the format [*annotcontlabel*][/*contlabel*]. If either label contains a slash (/) character then use `|` as the separator for the two labels instead.
- **timestamp** (*bool or str*) – Draw GMT time stamp logo on plot.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.
- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.
- **binary** (*bool or str*) – **i**|**o**[*ncols*][*type*][**w**][**+l**|**b**]. Select native binary input (using `binary="i"`) or output (using `binary="o"`), where *ncols* is the number of data columns of *type*, which must be one of:
  - **c** - int8_t (1-byte signed char)
  - **u** - uint8_t (1-byte unsigned char)
  - **h** - int16_t (2-byte signed int)
  - **H** - uint16_t (2-byte unsigned int)
  - **i** - int32_t (4-byte signed int)
  - **I** - uint32_t (4-byte unsigned int)
  - **l** - int64_t (8-byte signed int)
  - **L** - uint64_t (8-byte unsigned int)
  - **f** - 4-byte single-precision float
  - **d** - 8-byte double-precision float
  - **x** - use to skip *ncols* anywhere in the record

  For records with mixed types, append additional comma-separated combinations of *ncols type* (no space). The following modifiers are supported:
  - **w** after any item to force byte-swapping.

  - **+l|b** to indicate that the entire data file should be read as little- or big-endian, respectively.

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#bi-full.

- **panel** (*bool or int or list*) – [*row,col*|*index*]. Select a specific subplot panel. Only allowed when in subplot mode. Use `panel=True` to advance to the next panel in the selected order. Instead of *row,col* you may also give a scalar value *index* which depends on the order you set via `autolabel` when the subplot was defined. **Note**: *row*, *col*, and *index* all start at 0.
- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `d="-9999"` will replace all values equal to -9999 with NaN during input and all NaN values with -9999 during output. Prepend **i** to the *nodata* value for input columns only. Prepend **o** to the *nodata* value for output columns only.
- **find** (*str*) – [**~**]*"pattern"* | [**~**]/*regexp*/[**i**]. Only pass records that match the given *pattern* or regular expressions [Default processes all records]. Prepend **~** to the *pattern* or *regexp* to instead only pass data expressions that do not match the pattern. Append **i** for case insensitive matching. This does not apply to headers or segment headers.
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **header** (*str*) – [**i**|**o**][*n*][**+c**][**+d**][**+m***segheader*][**+r***remark*][**+t***title*]. Specify that input and/or output file(s) have *n* header records [Default is 0]. Prepend **i** if only the primary input should have header records. Prepend **o** to control the writing of header records, with the following modifiers supported:

  - **+d** to remove existing header records.
  - **+c** to add a header comment with column names to the output [Default is no column names].
  - **+m** to add a segment header *segheader* to the output after the header block [Default is no segment header].
  - **+r** to add a *remark* comment to the output [Default is no comment]. The *remark* string may contain \n to indicate line-breaks.
  - **+t** to add a *title* comment to the output [Default is no title]. The *title* string may contain \n to indicate line-breaks.

  Blank lines and lines starting with # are always skipped.

- **incols** (*str or 1d array*) – Specify data columns for primary input in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default reads all columns in order, starting with the first (i.e., column 0)].

  - For *1d array*: specify individual columns in input order (e.g., `incols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `incols="0:2,4+l"` to input the first three columns followed by the log-transformed 5th column). To read from a given column until the end of the record, leave off *stop* when specifying the column range. To read trailing text, add the column **t**. Append the word number to **t** to ingest only a single word from the trailing text. Instead of specifying columns, use `incols="n"` to simply read numerical input and skip trailing text. Optionally, append one of the following modifiers to any column or column range to transform the input columns:

  - ∗ **+l** to take the *log10* of the input values.
  - ∗ **+d** to divide the input values by the factor *divisor* [Default is 1].
  - ∗ **+s** to multiple the input values by the factor *scale* [Default is 1].
  - ∗ **+o** to add the given *offset* to the input values [Default is 0].

- **perspective** (*list or str*) – [**x**|**y**|**z**]*azim*[/*elev*[/*zlevel*]][**+w***lon0*/*lat0*[/*z0*]][**+v***x0*/*y0*]. Select perspective view and set the azimuth and elevation angle of the viewpoint. Default is [180, 90]. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#perspective-full.
- **transparency** (*int or float*) – Set transparency level, in [0-100] percent range. Default is 0, i.e., opaque. Only visible when PDF or raster format output is selected. Only the PNG format selection adds a transparency layer in the image (for further processing).

Examples using `pygmt.Figure.contour`

- *Contours*

pygmt.Figure.grdcontour

`Figure.grdcontour`(*grid*, ***, *annotation=None*, *frame=None*, *interval=None*, *label_placement=None*, *projection=None*, *limit=None*, *cut=None*, *region=None*, *resample=None*, *timestamp=None*, *verbose=None*, *pen=None*, *label=None*, *xshift=None*, *yshift=None*, *panel=None*, *coltypes=None*, *perspective=None*, *transparency=None*, ***kwargs*)

Convert grids or images to contours and plot them on maps.

Takes a grid file name or an xarray.DataArray object as input.

Full option list at https://docs.generic-mapping-tools.org/latest/grdcontour.html

Aliases:

- A = annotation
- B = frame
- C = interval
- G = label_placement
- J = projection
- L = limit
- Q = cut
- R = region
- S = resample
- U = timestamp
- V = verbose
- W = pen
- X = xshift
- Y = yshift
- c = panel

- f = coltypes
- l = label
- p = perspective
- t = transparency

Parameters

- **grid** (*str or xarray.DataArray*) – The file name of the input grid or the grid loaded as a DataArray.
- **interval** (*str or int*) – Specify the contour lines to generate.
  - The filename of a CPT file where the color boundaries will be used as contour levels.
  - The filename of a 2 (or 3) column file containing the contour levels (col 1), (**C**)ontour or (**A**)nnotate (col 2), and optional angle (col 3)
  - A fixed contour interval *cont_int* or a single contour with +*cont_int*
- **annotation** (*str, int, or list*) – Specify or disable annotated contour levels, modifies annotated contours specified in `interval`.
  - Specify a fixed annotation interval *annot_int* or a single annotation level +*annot_int*
  - Disable all annotation with **-**
  - Optional label modifiers can be specified as a single string `'[annot_int]+e'` or with a list of arguments `([annot_int], 'e', 'f10p', 'gred')`.
- **limit** (*str or list of 2 ints*) – *low*/*high*. Do no draw contours below *low* or above *high*, specify as string
- **cut** (*str or int*) – Do not draw contours with less than *cut* number of points.
- **resample** (*str or int*) – Resample smoothing factor.
- **projection** (*str*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **frame** (*bool or str or list*) – Set map boundary *frame and axes attributes*.
- **label_placement** (*str*) – [**d**|**f**|**n**|**l**|**L**|**x**|**X**]*args*. Control the placement of labels along the quoted lines. It supports five controlling algorithms. See https://docs.generic-mapping-tools.org/latest/grdcontour.html#g for details.
- **timestamp** (*bool or str*) – Draw GMT time stamp logo on plot.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages

- **pen** (*str*) – Set pen attributes for lines or the outline of symbols.
- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.
- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.
- **panel** (*bool or int or list*) – [*row,col*|*index*]. Select a specific subplot panel. Only allowed when in subplot mode. Use `panel=True` to advance to the next panel in the selected order. Instead of *row,col* you may also give a scalar value *index* which depends on the order you set via `autolabel` when the subplot was defined. **Note**: *row*, *col*, and *index* all start at 0.
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **label** (*str*) – Add a legend entry for the contour being plotted. Normally, the annotated contour is selected for the legend. You can select the regular contour instead, or both of them, by considering the label to be of the format [*annotcontlabel*][/*contlabel*]. If either label contains a slash (/) character then use `|` as the separator for the two labels instead.
- **perspective** (*list or str*) – [**x**|**y**|**z**]*azim*[/*elev*[/*zlevel*]][+**w***lon0*/*lat0*[/*z0*]][+**v***x0*/*y0*]. Select perspective view and set the azimuth and elevation angle of the viewpoint. Default is [180, 90]. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#perspective-full.
- **transparency** (*int or float*) – Set transparency level, in [0-100] percent range. Default is 0, i.e., opaque. Only visible when PDF or raster format output is selected. Only the PNG format selection adds a transparency layer in the image (for further processing).

Examples using `pygmt.Figure.grdcontour`

- *Creating a map with contour lines*

pygmt.Figure.grdimage

`Figure.grdimage`(*grid*, ***, *img_out=None*, *frame=None*, *cmap=None*, *img_in=None*, *dpi=None*, *bit_color=None*, *shading=None*, *projection=None*, *monochrome=None*, *no_clip=None*, *nan_transparent=None*, *region=None*, *timestamp=None*, *verbose=None*, *xshift=None*, *yshift=None*, *interpolation=None*, *panel=None*, *coltypes=None*, *perspective=None*, *transparency=None*, *cores=None*, ***kwargs*)

Project and plot grids or images.

Reads a 2-D grid file and produces a gray-shaded (or colored) map by building a rectangular image and assigning pixels a gray-shade (or color) based on the z-value and the CPT file. Optionally, illumination may be added by providing a file with intensities in the (-1,+1) range or instructions to derive intensities from the input data grid. Values outside this range will be clipped. Such intensity files can be created from the grid using `pygmt.grdgradient` and, optionally, modified by `grdmath` or `grdhisteq`. If GMT is built with GDAL support, `grid` can be an image file (geo-referenced or not). In this case the image can optionally be illuminated with the file provided via the `shading` parameter. Here, if image has no coordinates then those of the intensity file will be used.

When using map projections, the grid is first resampled on a new rectangular grid with the same dimensions. Higher resolution images can be obtained by using the `dpi` parameter. To obtain the resampled value (and hence shade or color) of each map pixel, its location is inversely projected back onto the input grid after which a value is interpolated between the surrounding input grid values. By default bi-cubic interpolation is used. Aliasing is avoided by also forward projecting the input grid nodes. If two or more nodes are projected onto the same pixel, their average will dominate in the calculation of the pixel value. Interpolation and aliasing is controlled with the `interpolation` parameter.

The `region` parameter can be used to select a map region larger or smaller than that implied by the extent of the grid.

Full parameter list at https://docs.generic-mapping-tools.org/latest/grdimage.html

Aliases:

- A = img_out
- B = frame
- C = cmap
- D = img_in
- E = dpi
- G = bit_color
- I = shading
- J = projection
- M = monochrome
- N = no_clip
- Q = nan_transparent
- R = region
- U = timestamp
- V = verbose
- X = xshift
- Y = yshift
- c = panel
- f = coltypes
- n = interpolation
- p = perspective
- t = transparency
- x = cores

Parameters

- **grid** (*str or xarray.DataArray*) – The file name or a DataArray containing the input 2-D gridded data set or image to be plotted (See GRID FILE FORMATS at https://docs.generic-mapping-tools.org/latest/grdimage.html#grid-file-formats).
- **img_out** (*str*) – *out_img*[=*driver*]. Save an image in a raster format instead of PostScript. Use extension .ppm for a Portable Pixel Map format which is the only raster format GMT can natively write. For GMT installations configured with GDAL support there are more choices: Append *out_img* to select the image file name and extension. If the extension is one of .bmp, .gif, .jpg, .png, or .tif then no driver information is required. For other output formats you must append the required GDAL driver. The *driver* is the driver code name used by GDAL; see your GDAL installation's documentation for available drivers. Append a **+c***args*

string where *args* is a list of one or more concatenated number of GDAL **-co** arguments. For example, to write a GeoPDF with the TerraGo format use `=PDF+cGEO_ENCODING=OGC_BP`. Notes: (1) If a tiff file (.tif) is selected then we will write a GeoTiff image if the GMT projection syntax translates into a PROJ syntax, otherwise a plain tiff file is produced. (2) Any vector elements will be lost.

- **frame** (*bool or str or list*) – Set map boundary *frame and axes attributes*.
- **cmap** (*str*) – File name of a CPT file or a series of comma-separated colors (e.g., *color1*,*color2*,*color3*) to build a linear continuous CPT from those colors automatically.
- **img_in** (*str*) – [**r**]. GMT will automatically detect standard image files (Geotiff, TIFF, JPG, PNG, GIF, etc.) and will read those via GDAL. For very obscure image formats you may need to explicitly set `img_in`, which specifies that the grid is in fact an image file to be read via GDAL. Append **r** to assign the region specified by `region` to the image. For example, if you have used `region='d'` then the image will be assigned a global domain. This mode allows you to project a raw image (an image without referencing coordinates).
- **dpi** (*int*) – [**i**|*dpi*]. Sets the resolution of the projected grid that will be created if a map projection other than Linear or Mercator was selected [100]. By default, the projected grid will be of the same size (rows and columns) as the input file. Specify **i** to use the PostScript image operator to interpolate the image at the device resolution.
- **bit_color** (*str*) – *color*[**+b**|**f**]. This parameter only applies when a resulting 1-bit image otherwise would consist of only two colors: black (0) and white (255). If so, this parameter will instead use the image as a transparent mask and paint the mask with the given color. Append **+b** to paint the background pixels (1) or **+f** for the foreground pixels [Default is **+f**].
- **shading** (*str or xarray.DataArray*) – [*intensfile*|*intensity*|*modifiers*]. Give the name of a grid file or a DataArray with intensities in the (-1,+1) range, or a constant intensity to apply everywhere (affects the ambient light). Alternatively, derive an intensity grid from the input data grid via a call to *pygmt.grdgradient*; append **+a***azimuth*, **+n***args*, and **+m***ambient* to specify azimuth, intensity, and ambient arguments for that module, or just give **+d** to select the default arguments (`+a-45+nt1+m0`). If you want a more specific intensity scenario then run *pygmt.grdgradient* separately first. If we should derive intensities from another file than grid, specify the file with suitable modifiers [Default is no illumination]. Note: If the input data is an *image* then an *intensfile* or constant *intensity* must be provided.
- **projection** (*str*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- **monochrome** (*bool*) – Force conversion to monochrome image using the (television) YIQ transformation. Cannot be used with `nan_transparent`.
- **no_clip** (*bool*) – Do not clip the image at the map boundary (only relevant for non-rectangular maps).
- **nan_transparent** (*bool*) – Make grid nodes with z = NaN transparent, using the color-masking feature in PostScript Level 3 (the PS device must support PS Level 3).
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);

  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.
- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.
- **panel** (*bool or int or list*) – [*row,col*|*index*]. Select a specific subplot panel. Only allowed when in subplot mode. Use `panel=True` to advance to the next panel in the selected order. Instead of *row,col* you may also give a scalar value *index* which depends on the order you set via `autolabel` when the subplot was defined. **Note**: *row*, *col*, and *index* all start at 0.
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **interpolation** (*str*) – [**b**|**c**|**l**|**n**][**+a**][**+b***BC*][**+c**][**+t***threshold*]. Select interpolation mode for grids. You can select the type of spline used:
  - **b** for B-spline
  - **c** for bicubic [Default]
  - **l** for bilinear
  - **n** for nearest-neighbor
- **perspective** (*list or str*) – [**x**|**y**|**z**]*azim*[/*elev*[/*zlevel*]][**+w***lon0*/*lat0*[/*z0*]][**+v***x0*/*y0*]. Select perspective view and set the azimuth and elevation angle of the viewpoint. Default is [180, 90]. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#perspective-full.
- **transparency** (*int or float*) – Set transparency level, in [0-100] percent range. Default is 0, i.e., opaque. Only visible when PDF or raster format output is selected. Only the PNG format selection adds a transparency layer in the image (for further processing).
- **cores** (*bool or int*) – [[**-**]*n*]. Limit the number of cores to be used in any OpenMP-enabled multi-threaded algorithms. By default we try to use all available cores. Set a number *n* to only use n cores (if too large it will be truncated to the maximum cores available). Finally, give a negative number -*n* to select (all - *n*) cores (or at least 1 if *n* equals or exceeds all).

Examples using `pygmt.Figure.grdimage`

- *Calculating grid gradient and radiance*
- *Clipping grid values*
- *Create 'wet-dry' mask grid*
- *Sampling along tracks*
- *Multiple colormaps*
- *Creating a map with contour lines*
- *Plotting Earth relief*

pygmt.Figure.grdview

`Figure.grdview`(*grid*, ***, *region=None*, *projection=None*, *zscale=None*, *zsize=None*, *frame=None*, *cmap=None*, *drapegrid=None*, *plane=None*, *surftype=None*, *contourpen=None*, *meshpen=None*, *facadepen=None*, *shading=None*, *verbose=None*, *xshift=None*, *yshift=None*, *panel=None*, *coltypes=None*, *interpolation=None*, *perspective=None*, *transparency=None*, ***kwargs*)

Create 3-D perspective image or surface mesh from a grid.

Reads a 2-D grid file and produces a 3-D perspective plot by drawing a mesh, painting a colored/gray-shaded surface made up of polygons, or by scanline conversion of these polygons to a raster image. Options include draping a data set on top of a surface, plotting of contours on top of the surface, and apply artificial illumination based on intensities provided in a separate grid file.

Full option list at https://docs.generic-mapping-tools.org/latest/grdview.html

Aliases:

- B = frame
- C = cmap
- G = drapegrid
- I = shading
- J = projection
- JZ = zsize
- Jz = zscale
- N = plane
- Q = surftype
- R = region
- V = verbose
- Wc = contourpen
- Wf = facadepen
- Wm = meshpen
- X = xshift
- Y = yshift
- c = panel
- f = coltypes
- n = interpolation
- p = perspective
- t = transparency

Parameters

- **grid** (*str or xarray.DataArray*) – The file name of the input relief grid or the grid loaded as a DataArray.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest. When used with `perspective`, optionally append */zmin/zmax* to indicate the range to use for the 3-D axes [Default is the region in the input grid].

- **projection** (*str*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- **zscale/zsize** (*float or str*) – Set z-axis scaling or z-axis size.
- **frame** (*bool or str or list*) – Set map boundary *frame and axes attributes*.
- **cmap** (*str*) – The name of the color palette table to use.
- **drapegrid** (*str or xarray.DataArray*) – The file name or a DataArray of the image grid to be draped on top of the relief provided by grid. [Default determines colors from grid]. Note that `zscale` and `plane` always refers to the grid. The drapegrid only provides the information pertaining to colors, which (if drapegrid is a grid) will be looked-up via the CPT (see `cmap`).
- **plane** (*float or str*) – *level*[**+g***fill*]. Draws a plane at this z-level. If the optional color is provided via the **+g** modifier, and the projection is not oblique, the frontal facade between the plane and the data perimeter is colored.
- **surftype** (*str*) – Specifies cover type of the grid. Select one of following settings:
  - **m** - mesh plot [Default].
  - **mx** or **my** - waterfall plots (row or column profiles).
  - **s** - surface plot, and optionally append **m** to have mesh lines drawn on top of the surface.
  - **i** - image plot.
  - **c** - Same as **i** but will make nodes with z = NaN transparent.

  For any of these choices, you may force a monochrome image by appending the modifier **+m**.
- **contourpen** (*str*) – Draw contour lines on top of surface or mesh (not image). Append pen attributes used for the contours.
- **meshpen** (*str*) – Sets the pen attributes used for the mesh. You must also select `surftype` of **m** or **sm** for meshlines to be drawn.
- **facadepen** (*str*) – Sets the pen attributes used for the facade. You must also select `plane` for the facade outline to be drawn.
- **shading** (*str*) – Provide the name of a grid file with intensities in the (-1,+1) range, or a constant intensity to apply everywhere (affects the ambient light). Alternatively, derive an intensity grid from the input data grid reliefgrid via a call to `grdgradient`; append **+a***azimuth*, **+n***args*, and **+m***ambient* to specify azimuth, intensity, and ambient arguments for that module, or just give **+d** to select the default arguments [Default is **+a**-45**+nt**1**+m**0].
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.

- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.
- **panel** (*bool or int or list*) – [*row,col*|*index*]. Select a specific subplot panel. Only allowed when in subplot mode. Use `panel=True` to advance to the next panel in the selected order. Instead of *row,col* you may also give a scalar value *index* which depends on the order you set via `autolabel` when the subplot was defined. **Note**: *row*, *col*, and *index* all start at 0.
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **interpolation** (*str*) – [**b**|**c**|**l**|**n**][**+a**][**+b***BC*][**+c**][**+t***threshold*]. Select interpolation mode for grids. You can select the type of spline used:
  - **b** for B-spline
  - **c** for bicubic [Default]
  - **l** for bilinear
  - **n** for nearest-neighbor
- **perspective** (*list or str*) – [**x**|**y**|**z**]*azim*[/*elev*[/*zlevel*]][**+w***lon0*/*lat0*[/*z0*]][**+v***x0*/*y0*]. Select perspective view and set the azimuth and elevation angle of the viewpoint. Default is [180, 90]. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#perspective-full.
- **transparency** (*int or float*) – Set transparency level, in [0-100] percent range. Default is 0, i.e., opaque. Only visible when PDF or raster format output is selected. Only the PNG format selection adds a transparency layer in the image (for further processing).

Examples using `pygmt.Figure.grdview`

- *Plotting a surface*
- *Creating a 3D perspective image*

pygmt.Figure.histogram

`Figure.`**`histogram`**(*data*, ***, *horizontal=None*, *frame=None*, *cmap=None*, *annotate=None*, *barwidth=None*, *center=None*, *fill=None*, *projection=None*, *extreme=None*, *distribution=None*, *cumulative=None*, *region=None*, *stairs=None*, *series=None*, *timestamp=None*, *verbose=None*, *pen=None*, *xshift=None*, *yshift=None*, *histtype=None*, *binary=None*, *panel=None*, *nodata=None*, *find=None*, *header=None*, *incols=None*, *label=None*, *perspective=None*, *transparency=None*, *wrap=None*, ***kwargs*)

Plots a histogram, and can read data from a file or list, array, or dataframe.

Full option list at https://docs.generic-mapping-tools.org/latest/histogram.html

Aliases:

- A = horizontal
- B = frame
- C = cmap
- D = annotate

- E = barwidth
- F = center
- G = fill
- J = projection
- L = extreme
- N = distribution
- Q = cumulative
- R = region
- S = stairs
- T = series
- U = timestamp
- V = verbose
- W = pen
- X = xshift
- Y = yshift
- Z = histtype
- b = binary
- c = panel
- d = nodata
- e = find
- h = header
- i = incols
- l = label
- p = perspective
- t = transparency
- w = wrap

Parameters

- **data** (*str or list or numpy.ndarray or pandas.DataFrame or xarray.Dataset or geopandas.GeoDataFrame*) – Pass in either a file name to an ASCII data table, a Python list, a 2D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **projection** (*str*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **frame** (*bool or str or list*) – Set map boundary *frame and axes attributes*.
- **cmap** (*str*) – File name of a CPT file or a series of comma-separated colors (e.g., *color1*,*color2*,*color3*) to build a linear continuous CPT from those colors automatically.

- **color** (*str or 1d array*) – Select color or pattern for filling of symbols or polygons. Default is no fill.
- **pen** (*str*) – Set pen attributes for lines or the outline of symbols.
- **panel** (*bool or int or list*) – [*row,col*|*index*]. Select a specific subplot panel. Only allowed when in subplot mode. Use `panel=True` to advance to the next panel in the selected order. Instead of *row,col* you may also give a scalar value *index* which depends on the order you set via `autolabel` when the subplot was defined. **Note**: *row*, *col*, and *index* all start at 0.
- **annotate** (*bool or str*) – [**+b**][**+f***font*][**+o***off*][**+r**]. Annotate each bar with the count it represents. Append any of the following modifiers: Use **+b** to place the labels beneath the bars instead of above; use **+f** to change to another font than the default annotation font; use **+o** to change the offset between bar and label [6p]; use **+r** to rotate the labels from horizontal to vertical.
- **barwidth** (*int or float or str*) – *width*[**+o***offset*]. Use an alternative histogram bar width than the default set via `series`, and optionally shift all bars by an *offset*. Here *width* is either an alternative width in data units, or the user may append a valid plot dimension unit (**c**|**i**|**p**) for a fixed dimension instead. Optionally, all bins may be shifted along the axis by *offset*. As for *width*, it may be given in data units of plot dimension units by appending the relevant unit.
- **center** (*bool*) – Center bin on each value. [Default is left edge].
- **distribution** (*bool or int or float or str*) – [*mode*][**+p***pen*]. Draw the equivalent normal distribution; append desired *pen* [Default is 0.25p,black]. The *mode* selects which central location and scale to use:
  - 0 = mean and standard deviation [Default];
  - 1 = median and L1 scale (1.4826 * median absolute deviation; MAD);
  - 2 = LMS (least median of squares) mode and scale.
- **cumulative** (*bool or str*) – [**r**]. Draw a cumulative histogram by passing `True`. Use **r** to display a reverse cumulative histogram.
- **extreme** (*str*) – **l**|**h**|**b**. The modifiers specify the handling of extreme values that fall outside the range set by `series`. By default these values are ignored. Append **b** to let these values be included in the first or last bins. To only include extreme values below first bin into the first bin, use **l**, and to only include extreme values above the last bin into that last bin, use **h**.
- **stairs** (*bool*) – Draws a stairs-step diagram which does not include the internal bars of the default histogram.
- **horizontal** (*bool*) – Plot the histogram using horizontal bars instead of the default vertical bars.
- **series** (*int or str or list*) – [*min*/*max*/]*inc*[**+n**]. Set the interval for the width of each bar in the histogram.
- **histtype** (*int or str*) – [*type*][**+w**]. Choose between 6 types of histograms:
  - 0 = counts [Default]
  - 1 = frequency_percent
  - 2 = log (1.0 + count)
  - 3 = log (1.0 + frequency_percent)
  - 4 = log10 (1.0 + count)

  - 5 = log10 (1.0 + frequency_percent).

  To use weights provided as a second data column instead of pure counts, append **+w**.

- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.
- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.
- **timestamp** (*bool or str*) – Draw GMT time stamp logo on plot.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **binary** (*bool or str*) – **i**|**o**[*ncols*][*type*][**w**][**+l**|**b**]. Select native binary input (using `binary="i"`) or output (using `binary="o"`), where *ncols* is the number of data columns of *type*, which must be one of:
  - **c** - int8_t (1-byte signed char)
  - **u** - uint8_t (1-byte unsigned char)
  - **h** - int16_t (2-byte signed int)
  - **H** - uint16_t (2-byte unsigned int)
  - **i** - int32_t (4-byte signed int)
  - **I** - uint32_t (4-byte unsigned int)
  - **l** - int64_t (8-byte signed int)
  - **L** - uint64_t (8-byte unsigned int)
  - **f** - 4-byte single-precision float
  - **d** - 8-byte double-precision float
  - **x** - use to skip *ncols* anywhere in the record

  For records with mixed types, append additional comma-separated combinations of *ncols type* (no space). The following modifiers are supported:

  - **w** after any item to force byte-swapping.
  - **+l**|**b** to indicate that the entire data file should be read as little- or big-endian, respectively.

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#bi-full.
- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `d="-9999"` will replace all values equal to -9999 with NaN during input and all NaN values with -9999 during output. Prepend **i** to the *nodata* value for input columns only. Prepend **o** to the *nodata* value for output columns only.

- **find** (*str*) – [~]*"pattern"* | [~]/*regexp*/[**i**]. Only pass records that match the given *pattern* or regular expressions [Default processes all records]. Prepend **~** to the *pattern* or *regexp* to instead only pass data expressions that do not match the pattern. Append **i** for case insensitive matching. This does not apply to headers or segment headers.
- **header** (*str*) – [**i**|**o**][*n*][**+c**][**+d**][**+m***segheader*][**+r***remark*][**+t***title*]. Specify that input and/or output file(s) have *n* header records [Default is 0]. Prepend **i** if only the primary input should have header records. Prepend **o** to control the writing of header records, with the following modifiers supported:
  - **+d** to remove existing header records.
  - **+c** to add a header comment with column names to the output [Default is no column names].
  - **+m** to add a segment header *segheader* to the output after the header block [Default is no segment header].
  - **+r** to add a *remark* comment to the output [Default is no comment]. The *remark* string may contain \n to indicate line-breaks.
  - **+t** to add a *title* comment to the output [Default is no title]. The *title* string may contain \n to indicate line-breaks.

  Blank lines and lines starting with # are always skipped.
- **incols** (*str or 1d array*) – Specify data columns for primary input in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default reads all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in input order (e.g., `incols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `incols="0:2,4+l"` to input the first three columns followed by the log-transformed 5th column). To read from a given column until the end of the record, leave off *stop* when specifying the column range. To read trailing text, add the column **t**. Append the word number to **t** to ingest only a single word from the trailing text. Instead of specifying columns, use `incols="n"` to simply read numerical input and skip trailing text. Optionally, append one of the following modifiers to any column or column range to transform the input columns:
    - **+l** to take the *log10* of the input values.
    - **+d** to divide the input values by the factor *divisor* [Default is 1].
    - **+s** to multiple the input values by the factor *scale* [Default is 1].
    - **+o** to add the given *offset* to the input values [Default is 0].
- **label** (*str*) – Add a legend entry for the symbol or line being plotted. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#l-full.
- **perspective** (*list or str*) – [**x**|**y**|**z**]*azim*[/*elev*[/*zlevel*]][**+w***lon0*/*lat0*[/*z0*]][**+v***x0*/*y0*]. Select perspective view and set the azimuth and elevation angle of the viewpoint. Default is [180, 90]. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#perspective-full.
- **transparency** (*int or float*) – Set transparency level, in [0-100] percent range. Default is 0, i.e., opaque. Only visible when PDF or raster format output is selected. Only the PNG format selection adds a transparency layer in the image (for further processing).

- **wrap** (*str*) – **y**|**a**|**w**|**d**|**h**|**m**|**s**|**c***period*[/*phase*][**+c***col*]. Convert the input *x*-coordinate to a cyclical coordinate, or a different column if selected via **+c***col*. The following cyclical coordinate transformations are supported:
  - **y** - yearly cycle (normalized)
  - **a** - annual cycle (monthly)
  - **w** - weekly cycle (day)
  - **d** - daily cycle (hour)
  - **h** - hourly cycle (minute)
  - **m** - minute cycle (second)
  - **s** - second cycle (second)
  - **c** - custom cycle (normalized)

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#w-full.

Examples using `pygmt.Figure.histogram`

- *Histogram*

pygmt.Figure.image

`Figure.`**`image`**(*imagefile*, ***, *region=None*, *projection=None*, *position=None*, *box=None*, *monochrome=None*, *timestamp=None*, *verbose=None*, *xshift=None*, *yshift=None*, *panel=None*, *perspective=None*, *transparency=None*, ***kwargs*)

Place images or EPS files on maps.

Reads an Encapsulated PostScript file or a raster image file and plots it on a map.

Full option list at https://docs.generic-mapping-tools.org/latest/image.html

Aliases:

- D = position
- F = box
- J = projection
- M = monochrome
- R = region
- U = timestamp
- V = verbose
- X = xshift
- Y = yshift
- c = panel
- p = perspective
- t = transparency

Parameters

- **imagefile** (*str*) – This must be an Encapsulated PostScript (EPS) file or a raster image. An EPS file must contain an appropriate BoundingBox. A raster file can have a depth of 1, 8, 24, or 32 bits and is read via GDAL. Note: If GDAL was not configured during GMT installation then only EPS files are supported.
- **projection** (*str*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **position** (*str*) – [**g**|**j**|**J**|**n**|**x**]*refpoint***+r***dpi***+w**[**-**]*width*[/*height*][**+j***justify*][**+n***nx*[/*ny*]][**+o***dx*[/*dy*]]. Sets reference point on the map for the image.
- **box** (*bool or str*) – [**+c***clearances*][**+g***fill*][**+i**[[*gap*/]*pen*]][**+p**[*pen*]][**+r**[*radius*]][**+s**[[*dx*/*dy*/][*shade*]]]. Without further arguments, draws a rectangular border around the image using MAP_FRAME_PEN.
- **monochrome** (*bool*) – Convert color image to monochrome grayshades using the (television) YIQ-transformation.
- **timestamp** (*bool or str*) – Draw GMT time stamp logo on plot.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.
- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.
- **panel** (*bool or int or list*) – [*row,col*|*index*]. Select a specific subplot panel. Only allowed when in subplot mode. Use `panel=True` to advance to the next panel in the selected order. Instead of *row,col* you may also give a scalar value *index* which depends on the order you set via `autolabel` when the subplot was defined. **Note**: *row*, *col*, and *index* all start at 0.
- **perspective** (*list or str*) – [**x**|**y**|**z**]*azim*[/*elev*[/*zlevel*]][**+w***lon0*/*lat0*[/*z0*]][**+v***x0*/*y0*]. Select perspective view and set the azimuth and elevation angle of the viewpoint. Default is [180, 90]. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#perspective-full.
- **transparency** (*int or float*) – Set transparency level, in [0-100] percent range. Default is 0, i.e., opaque. Only visible when PDF or raster format output is selected. Only the PNG format selection adds a transparency layer in the image (for further processing).

Examples using `pygmt.Figure.image`

- *Images on figures*

pygmt.Figure.inset

`Figure.inset`(***, *position=None*, *box=None*, *margin=None*, *no_clip=None*, *verbose=None*, ***kwargs*)

Create an inset figure to be placed within a larger figure.

This function sets the position, frame, and margins for a smaller figure inside of the larger figure. Plotting functions that are called within the context manager are added to the inset figure.

Full option list at https://docs.generic-mapping-tools.org/latest/inset.html

Aliases:

- D = position
- F = box
- M = margin
- N = no_clip
- V = verbose

Parameters

- **position** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]] | [**g**|**j**|**J**|**n**|**x**]*refpoint***+w***width*[/*height*][**+j***justify*][**+o***dx*[/*dy*]].

  This is the only required parameter. Define the map inset rectangle on the map. Specify the rectangle in one of three ways:

  Append **g***lon*/*lat* for map (user) coordinates, **j***code* or **J***code* for setting the *refpoint* via a 2-char justification code that refers to the (invisible) projected map bounding box, **n***xn*/*yn* for normalized (0-1) bounding box coordinates, or **x***x*/*y* for plot coordinates (inches, cm, points, append unit). All but **x** requires both `region` and `projection` to be specified. You can offset the reference point via **+o***dx*/*dy* in the direction implied by *code* or **+j***justify*.

  Alternatively, give *west/east/south/north* of geographic rectangle bounded by parallels and meridians; append **+r** if the coordinates instead are the lower left and upper right corners of the desired rectangle. (Or, give *xmin/xmax/ymin/ymax* of bounding rectangle in projected coordinates and optionally append **+u***unit* [Default coordinate unit is meter (e)].

  Append **+w***width*[/*height*] of bounding rectangle or box in plot coordinates (inches, cm, etc.). By default, the anchor point on the scale is assumed to be the bottom left corner (**BL**), but this can be changed by appending **+j** followed by a 2-char justification code *justify*. **Note**: If **j** is used then *justify* defaults to the same as *refpoint*, if **J** is used then *justify* defaults to the mirror opposite of *refpoint*. Specify inset box attributes via the `box` parameter [Default is outline only].

- **box** (*str or bool*) – [**+c***clearances*][**+g***fill*][**+i**[[*gap*/]*pen*]][**+p**[*pen*]][**+r**[*radius*]][**+s**[[*dx*/*dy*/][*shade*]]].

  If passed `True`, this draws a rectangular box around the map inset using the default pen; specify a different pen with **+p***pen*. Add **+g***fill* to fill the logo box [Default is no fill]. Append **+c***clearance* where *clearance* is either *gap*, *xgap*/*ygap*, or *lgap*/*rgap*/*bgap*/*tgap* where these items are uniform, separate in x- and y-direction, or individual side spacings between logo and border. Append **+i** to draw a secondary, inner border as well. We use a uniform *gap* between borders of 2**p** and the default pen unless other values are specified. Append **+r** to draw rounded rectangular borders instead, with a 6p corner radius. You can override this radius by appending another value. Append **+s** to draw an offset background shaded region. Here, *dx*/*dy* indicates the shift relative to the foreground frame [4p/-4p] and `shade` sets the fill style to use for shading [Default is gray50].

- **margin** (*int or str or list*) – This is clearance that is added around the inside of the inset. Plotting will take place within the inner region only. The margins can be a single value, a pair of values separated (for setting separate horizontal and vertical margins), or the full set of four margins (for setting separate left, right, bottom, and top margins). When passing multiple values, it can be either a list or a string with the values separated by forward slashes [Default is no margins].
- **no_clip** (*bool*) – Do NOT clip features extruding outside map inset boundaries [Default is clip].
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages

Examples

```
>>> import pygmt
>>>
>>> # Create the larger figure
>>> fig = pygmt.Figure()
>>> fig.coast(region="MG+r2", water="lightblue", shorelines="thin")
>>> # Use a "with" statement to initialize the inset context manager
>>> # Setting the position to top left and a width of 3.5 centimeters
>>> with fig.inset(position="jTL+w3.5c+o0.2c", margin=0, box="+pgreen"):
...     # Map elements under the "with" statement are plotted in the inset
...     fig.coast(
...         region="g",
...         projection="G47/-20/3.5c",
...         land="gray",
...         water="white",
...         dcw="MG+gred",
...     )
...
>>> # Map elements outside the "with" block are plotted in the main figure
>>> fig.logo(position="jBR+o0.2c+w3c")
>>> fig.show()
<IPython.core.display.Image object>
```

Examples using `pygmt.Figure.inset`

- *Inset*
- *Inset map showing a rectangular region*
- *Adding an inset to the figure*

pygmt.Figure.legend

`Figure.legend`(*spec=None*, *position='JTR+jTR+o0.2c'*, *box='+gwhite+p1p'*, ***, *region=None*, *projection=None*, *timestamp=None*, *verbose=None*, *xshift=None*, *yshift=None*, *panel=None*, *perspective=None*, *transparency=None*, ***kwargs*)

Plot legends on maps.

Makes legends that can be overlaid on maps. Reads specific legend-related information from an input file, or automatically creates legend entries from plotted symbols that have labels. Unless otherwise noted, annotations will be made using the primary annotation font and size in effect (i.e., FONT_ANNOT_PRIMARY).

Full option list at https://docs.generic-mapping-tools.org/latest/legend.html

Aliases:

- D = position
- F = box
- J = projection
- R = region
- U = timestamp
- V = verbose
- X = xshift
- Y = yshift
- c = panel
- p = perspective
- t = transparency

Parameters

- **spec** (*None or str*) – Either `None` [default] for using the automatically generated legend specification file, or a *filename* pointing to the legend specification file.
- **projection** (*str*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **position** (*str*) – [**g**|**j**|**J**|**n**|**x**]*refpoint*+**w***width*[/*height*][**+j***justify*][**+l***spacing*][**+o***dx*[/*dy*]]. Defines the reference point on the map for the legend. By default, uses **JTR**+**jTR**+**o***0.2c* which places the legend at the top-right corner inside the map frame, with a 0.2 cm offset.
- **box** (*bool or str*) – [**+c***clearances*][**+g***fill*][**+i**[[*gap*/]*pen*]][**+p**[*pen*]][**+r**[*radius*]][**+s**[[*dx*/*dy*/][*shade*]]]. Without further arguments, draws a rectangular border around the legend using MAP_FRAME_PEN. By default, uses **+g**white**+p**1p which draws a box around the legend using a 1p black pen and adds a white background.

- **timestamp** (*bool or str*) – Draw GMT time stamp logo on plot.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.
- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.
- **panel** (*bool or int or list*) – [*row,col*|*index*]. Select a specific subplot panel. Only allowed when in subplot mode. Use `panel=True` to advance to the next panel in the selected order. Instead of *row,col* you may also give a scalar value *index* which depends on the order you set via `autolabel` when the subplot was defined. **Note**: *row*, *col*, and *index* all start at 0.
- **perspective** (*list or str*) – [**x**|**y**|**z**]*azim*[/*elev*[/*zlevel*]][+**w***lon0*/*lat0*[/*z0*]][+**v***x0*/*y0*]. Select perspective view and set the azimuth and elevation angle of the viewpoint. Default is [180, 90]. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#perspective-full.
- **transparency** (*int or float*) – Set transparency level, in [0-100] percent range. Default is 0, i.e., opaque. Only visible when PDF or raster format output is selected. Only the PNG format selection adds a transparency layer in the image (for further processing).

Examples using `pygmt.Figure.legend`

- *Roads*
- *Scatter plots with a legend*
- *Double Y axes graph*
- *Legend*

pygmt.Figure.logo

`Figure.logo`(**, region=None, projection=None, position=None, box=None, style=None, timestamp=None, verbose=None, xshift=None, yshift=None, panel=None, transparency=None, **kwargs*)

Plot the GMT logo.

By default, the GMT logo is 2 inches wide and 1 inch high and will be positioned relative to the current plot origin. Use various options to change this and to place a transparent or opaque rectangular map panel behind the GMT logo.

Full option list at https://docs.generic-mapping-tools.org/latest/gmtlogo.html.

Aliases:

- D = position
- F = box
- J = projection
- R = region
- S = style
- U = timestamp
- V = verbose
- X = xshift
- Y = yshift
- c = panel
- t = transparency

Parameters

- **projection** (*str*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **position** (*str*) – [**g**|**j**|**J**|**n**|**x**]*refpoint***+w***width*[**+j***justify*][**+o***dx*[/*dy*]]. Sets reference point on the map for the image.
- **box** (*bool or str*) – Without further arguments, draws a rectangular border around the GMT logo.
- **style** (*str*) – [**l**|**n**|**u**]. Control what is written beneath the map portion of the logo.
  - **l** to plot the text label "The Generic Mapping Tools" [Default]
  - **n** to skip the label placement
  - **u** to place the URL to the GMT site
- **timestamp** (*bool or str*) – Draw GMT time stamp logo on plot.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.
- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.

- **panel** (*bool or int or list*) – [*row,col*|*index*]. Select a specific subplot panel. Only allowed when in subplot mode. Use `panel=True` to advance to the next panel in the selected order. Instead of *row,col* you may also give a scalar value *index* which depends on the order you set via `autolabel` when the subplot was defined. **Note**: *row*, *col*, and *index* all start at 0.
- **transparency** (*int or float*) – Set transparency level, in [0-100] percent range. Default is 0, i.e., opaque. Only visible when PDF or raster format output is selected. Only the PNG format selection adds a transparency layer in the image (for further processing).

Examples using `pygmt.Figure.logo`

- *Logo*

pygmt.Figure.meca

`Figure.meca`(*spec*, *scale*, *longitude=None*, *latitude=None*, *depth=None*, *convention=None*, *component='full'*, *plot_longitude=None*, *plot_latitude=None*, ***, *region=None*, *projection=None*, *offset=None*, *frame=None*, *no_clip=None*, *verbose=None*, *xshift=None*, *yshift=None*, *panel=None*, *perspective=None*, *transparency=None*, ***kwargs*)

Plot focal mechanisms.

Full option list at https://docs.generic-mapping-tools.org/latest/supplements/seis/meca.html

Note: Currently, labeling of beachballs with text strings is only supported via providing a file to *spec* as input.

Aliases:

- A = offset
- B = frame
- J = projection
- N = no_clip
- R = region
- V = verbose
- X = xshift
- Y = yshift
- c = panel
- p = perspective
- t = transparency

Parameters

- **spec** (*dict, 1D array, 2D array, pd.DataFrame, or str*) – Either a filename containing focal mechanism parameters as columns, a 1- or 2-D array with the same, or a dictionary. If a filename or array, *convention* is required so we know how to interpret the columns/entries. If a dictionary, the following combinations of keys are supported; these determine the convention. Dictionary may contain values for a single focal mechanism or lists of values for many focal mechanisms. A Pandas DataFrame may optionally contain columns latitude, longitude, depth, plot_longitude, and/or plot_latitude instead of passing them to the meca method.

  - "aki" — *strike, dip, rake, magnitude*
  - "gcmt" — *strike1, dip1, rake1, strike2, dip2, rake2, mantissa, exponent*
  - "mt" — *mrr, mtt, mff, mrt, mrf, mtf, exponent*
  - "partial" — *strike1, dip1, strike2, fault_type, magnitude*
  - "principal_axis" — *t_exponent, t_azimuth, t_plunge, n_exponent, n_azimuth, n_plunge, p_exponent, p_azimuth, p_plunge, exponent*

- **scale** (*str*) – Adjusts the scaling of the radius of the beachball, which is proportional to the magnitude. Scale defines the size for magnitude = 5 (i.e. scalar seismic moment M0 = 4.0E23 dynes-cm)
- **longitude** (*int, float, list, or 1d numpy array*) – Longitude(s) of event location. Ignored if *spec* is not a dictionary. List must be the length of the number of events. Ignored if *spec* is a DataFrame and contains a 'longitude' column.
- **latitude** (*int, float, list, or 1d numpy array*) – Latitude(s) of event location. Ignored if *spec* is not a dictionary. List must be the length of the number of events. Ignored if *spec* is a DataFrame and contains a 'latitude' column.
- **depth** (*int, float, list, or 1d numpy array*) – Depth(s) of event location in kilometers. Ignored if *spec* is not a dictionary. List must be the length of the number of events. Ignored if *spec* is a DataFrame and contains a 'depth' column.
- **convention** (*str*) – "aki" (Aki & Richards), "gcmt" (global CMT), "mt" (seismic moment tensor), "partial" (partial focal mechanism), or "principal_axis" (principal axis). Ignored if *spec* is a dictionary or dataframe.
- **component** (*str*) – The component of the seismic moment tensor to plot. "full" (the full seismic moment tensor), "dc" (the closest double couple with zero trace and zero determinant), "deviatoric" (zero trace)
- **plot_longitude** (*int, float, list, or 1d numpy array*) – Longitude(s) at which to place beachball, only used if *spec* is a dictionary. List must be the length of the number of events. Ignored if *spec* is a DataFrame and contains a 'plot_longitude' column.
- **plot_latitude** (*int, float, list, or 1d numpy array*) – Latitude(s) at which to place beachball, only used if *spec* is a dictionary. List must be the length of the number of events. Ignored if *spec* is a DataFrame and contains a 'plot_latitude' column.
- **offset** (*bool or str*) – Offsets beachballs to the longitude, latitude specified in the last two columns of the input file or array, or by *plot_longitude* and *plot_latitude* if provided. A small circle is plotted at the initial location and a line connects the beachball to the circle. Specify pen and optionally append +s*size* to change the line style and/or size of the circle.
- **no_clip** (*bool*) – Does NOT skip symbols that fall outside frame boundary specified by *region* [Default is False, i.e. plot symbols inside map frame only].
- **projection** (*str*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **frame** (*bool or str or list*) – Set map boundary *frame and axes attributes*.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:

    - **q** - Quiet, not even fatal error messages are produced
    - **e** - Error messages only
    - **w** - Warnings [Default]
    - **t** - Timings (report runtimes for time-intensive algorithms);
    - **i** - Informational messages (same as `verbose=True`)
    - **c** - Compatibility warnings
    - **d** - Debugging messages

- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.
- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.
- **panel** (*bool or int or list*) – [*row,col*|*index*]. Select a specific subplot panel. Only allowed when in subplot mode. Use `panel=True` to advance to the next panel in the selected order. Instead of *row,col* you may also give a scalar value *index* which depends on the order you set via `autolabel` when the subplot was defined. **Note**: *row*, *col*, and *index* all start at 0.
- **perspective** (*list or str*) – [**x**|**y**|**z**]*azim*[/*elev*[/*zlevel*]][**+w***lon0*/*lat0*[/*z0*]][**+v***x0*/*y0*]. Select perspective view and set the azimuth and elevation angle of the viewpoint. Default is [180, 90]. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#perspective-full.
- **transparency** (*int or float*) – Set transparency level, in [0-100] percent range. Default is 0, i.e., opaque. Only visible when PDF or raster format output is selected. Only the PNG format selection adds a transparency layer in the image (for further processing).

Examples using `pygmt.Figure.meca`

- *Focal mechanisms*

pygmt.Figure.plot

`Figure.plot`(*data=None*, *x=None*, *y=None*, *size=None*, *direction=None*, ***, *straight_line=None*, *frame=None*, *cmap=None*, *offset=None*, *error_bar=None*, *connection=None*, *color=None*, *intensity=None*, *projection=None*, *close=None*, *no_clip=None*, *region=None*, *style=None*, *timestamp=None*, *verbose=None*, *pen=None*, *xshift=None*, *yshift=None*, *zvalue=None*, *aspatial=None*, *binary=None*, *panel=None*, *nodata=None*, *find=None*, *coltypes=None*, *gap=None*, *header=None*, *incols=None*, *label=None*, *perspective=None*, *transparency=None*, *wrap=None*, ***kwargs*)

Plot lines, polygons, and symbols in 2-D.

Takes a matrix, (x,y) pairs, or a file name as input and plots lines, polygons, or symbols at those locations on a map.

Must provide either `data` or `x`/`y`.

If providing data through `x`/`y`, `color` can be a 1d array that will be mapped to a colormap.

If a symbol is selected and no symbol size given, then plot will interpret the third column of the input data as symbol size. Symbols whose size is <= 0 are skipped. If no symbols are specified then the symbol code (see `style` below) must be present as last column in the input. If `style` is not used, a line connecting the data points will be drawn instead. To explicitly close polygons, use `close`. Select a fill with `color`. If `color` is set, `pen` will control whether the polygon outline is drawn or not. If a symbol is selected, `color` and `pen` determines the fill and outline/no outline, respectively.

Full parameter list at https://docs.generic-mapping-tools.org/latest/plot.html

Aliases:

- A = straight_line
- B = frame
- C = cmap
- D = offset
- E = error_bar
- F = connection
- G = color
- I = intensity
- J = projection
- L = close
- N = no_clip
- R = region
- S = style
- U = timestamp
- V = verbose
- W = pen
- X = xshift
- Y = yshift
- Z = zvalue
- a = aspatial
- b = binary
- c = panel
- d = nodata
- e = find
- f = coltypes
- g = gap
- h = header
- i = incols
- l = label
- p = perspective
- t = transparency
- w = wrap

Parameters

- **data** (*str or numpy.ndarray or pandas.DataFrame or xarray.Dataset or geopandas.GeoDataFrame*) – Pass in either a file name to an ASCII data table, a 2D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data. Use parameter `incols` to choose which columns are x, y, color, and size, respectively.
- **x/y** (*float or 1d arrays*) – The x and y coordinates, or arrays of x and y coordinates of the data points
- **size** (*1d array*) – The size of the data points in units specified using `style`. Only valid if using `x`/`y`.
- **direction** (*list of two 1d arrays*) – If plotting vectors (using `style='V'` or `style='v'`), then should be a list of two 1d arrays with the vector directions. These can be angle and length, azimuth and length, or x and y components, depending on the style options chosen.
- **projection** (*str*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **straight_line** (*bool or str*) – [**m**|**p**|**x**|**y**]. By default, geographic line segments are drawn as great circle arcs. To draw them as straight lines, use `straight_line`. Alternatively, add **m** to draw the line by first following a meridian, then a parallel. Or append **p** to start following a parallel, then a meridian. (This can be practical to draw a line along parallels, for example). For Cartesian data, points are simply connected, unless you append **x** or **y** to draw stair-case curves that whose first move is along *x* or *y*, respectively.
- **frame** (*bool or str or list*) – Set map boundary *frame and axes attributes*.
- **cmap** (*str*) – File name of a CPT file or a series of comma-separated colors (e.g., *color1*,*color2*,*color3*) to build a linear continuous CPT from those colors automatically.
- **offset** (*str*) – *dx/dy*. Offset the plot symbol or line locations by the given amounts *dx/dy* [Default is no offset]. If *dy* is not given it is set equal to *dx*.
- **error_bar** (*bool or str*) – [**+b**|**d**|**D**][**+xl**|**r**|*x0*][**+yl**|**r**|*y0*][**+p***pen*]. Draw symmetrical error bars. Full documentation is at https://docs.generic-mapping-tools.org/latest/plot.html#e.
- **connection** (*str*) – [**c**|**n**|**r**][**a**|**f**|**s**|**r**|*refpoint*]. Alter the way points are connected (by specifying a *scheme*) and data are grouped (by specifying a *method*). Append one of three line connection schemes:
  - **c** : Draw continuous line segments for each group [Default].
  - **r** : Draw line segments from a reference point reset for each group.
  - **n** : Draw networks of line segments between all points in each group.

  Optionally, append the one of four segmentation methods to define the group:
  - **a** : Ignore all segment headers, i.e., let all points belong to a single group, and set group reference point to the very first point of the first file.
  - **f** : Consider all data in each file to be a single separate group and reset the group reference point to the first point of each group.
  - **s** : Segment headers are honored so each segment is a group; the group reference point is reset to the first point of each incoming segment [Default].

  - **r** : Same as **s**, but the group reference point is reset after each record to the previous point (this method is only available with the `connection='r'` scheme).

  Instead of the codes **a**|**f**|**s**|**r** you may append the coordinates of a *refpoint* which will serve as a fixed external reference point for all groups.

- **color** (*str or 1d array*) – Select color or pattern for filling of symbols or polygons. Default is no fill. *color* can be a 1d array, but it is only valid if using `x`/`y` and `cmap=True` is also required.
- **intensity** (*float or bool or 1d array*) – Provide an *intensity* value (nominally in the -1 to +1 range) to modulate the fill color by simulating illumination. If using `intensity=True`, we will instead read *intensity* from the first data column after the symbol parameters (if given). *intensity* can also be a 1d array to set varying intensity for symbols, but it is only valid for `x`/`y` pairs.
- **close** (*str*) – [**+b**|**d**|**D**][**+xl**|**r**|*x0*][**+yl**|**r**|*y0*][**+p***pen*]. Force closed polygons. Full documentation is at https://docs.generic-mapping-tools.org/latest/plot.html#l.
- **no_clip** (*bool or str*) – [**c**|**r**]. Do NOT clip symbols that fall outside map border [Default plots points whose coordinates are strictly inside the map border only]. The parameter does not apply to lines and polygons which are always clipped to the map region. For periodic (360-longitude) maps we must plot all symbols twice in case they are clipped by the repeating boundary. `no_clip=True` will turn off clipping and not plot repeating symbols. Use `no_clip="r"` to turn off clipping but retain the plotting of such repeating symbols, or use `no_clip="c"` to retain clipping but turn off plotting of repeating symbols.
- **style** (*str*) – Plot symbols (including vectors, pie slices, fronts, decorated or quoted lines).
- **pen** (*str*) – Set pen attributes for lines or the outline of symbols.
- **timestamp** (*bool or str*) – Draw GMT time stamp logo on plot.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.
- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.
- **zvalue** (*str*) – *value*|*file*. Instead of specifying a symbol or polygon fill and outline color via `color` and `pen`, give both a *value* via `zvalue` and a color lookup table via `cmap`. Alternatively, give the name of a *file* with one z-value (read from the last column) for each polygon in the input data. To apply it to the fill color, use `color='+z'`. To apply it to the pen color, append **+z** to `pen`.
- **aspatial** (*bool or str*) – [*col*=]*name*[,...]. Control how aspatial data are handled during input and output. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#aspatial-full.

- **binary** (*bool or str*) – **i**|**o**[*ncols*][*type*][**w**][**+l**|**b**]. Select native binary input (using `binary="i"`) or output (using `binary="o"`), where *ncols* is the number of data columns of *type*, which must be one of:
  - **c** - int8_t (1-byte signed char)
  - **u** - uint8_t (1-byte unsigned char)
  - **h** - int16_t (2-byte signed int)
  - **H** - uint16_t (2-byte unsigned int)
  - **i** - int32_t (4-byte signed int)
  - **I** - uint32_t (4-byte unsigned int)
  - **l** - int64_t (8-byte signed int)
  - **L** - uint64_t (8-byte unsigned int)
  - **f** - 4-byte single-precision float
  - **d** - 8-byte double-precision float
  - **x** - use to skip *ncols* anywhere in the record

  For records with mixed types, append additional comma-separated combinations of *ncols type* (no space). The following modifiers are supported:
  - **w** after any item to force byte-swapping.
  - **+l**|**b** to indicate that the entire data file should be read as little- or big-endian, respectively.

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#bi-full.
- **panel** (*bool or int or list*) – [*row,col*|*index*]. Select a specific subplot panel. Only allowed when in subplot mode. Use `panel=True` to advance to the next panel in the selected order. Instead of *row,col* you may also give a scalar value *index* which depends on the order you set via `autolabel` when the subplot was defined. **Note**: *row*, *col*, and *index* all start at 0.
- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `d="-9999"` will replace all values equal to -9999 with NaN during input and all NaN values with -9999 during output. Prepend **i** to the *nodata* value for input columns only. Prepend **o** to the *nodata* value for output columns only.
- **find** (*str*) – [**~**]*"pattern"* | [**~**]/*regexp*/[**i**]. Only pass records that match the given *pattern* or regular expressions [Default processes all records]. Prepend **~** to the *pattern* or *regexp* to instead only pass data expressions that do not match the pattern. Append **i** for case insensitive matching. This does not apply to headers or segment headers.
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **gap** (*str or list*) – [**a**]**x**|**y**|**d**|**X**|**Y**|**D**|[*col*]**z***gap*[**+n**|**p**]. Examine the spacing between consecutive data points in order to impose breaks in the line. To specify multiple criteria, provide a list with each item containing a string describing one set of criteria. Prepend **a** to specify that all the criteria must be met [Default is to impose breaks if any criteria are met]. The following modifiers are supported:
  - **x**|**X** - define a gap when there is a large enough change in the x coordinates (upper case to use projected coordinates).

  - **y**|**Y** - define a gap when there is a large enough change in the y coordinates (upper case to use projected coordinates).
  - **d**|**D** - define a gap when there is a large enough distance between coordinates (upper case to use projected coordinates).
  - [*col*]**z** - define a gap when there is a large enough change in the data in column *col* [default *col* is 2 (i.e., 3rd column)].

  A unit **u** may be appended to the specified *gap*:

  - For geographic data (**x**|**y**|**d**), the unit may be arc **d**(egree), **m**(inute), and **s**(econd), or (m)**e**(ter), **f**(eet), **k**(ilometer), **M**(iles), or **n**(autical miles) [Default is (m)**e**(ter)].
  - For projected data (**X**|**Y**|**D**), the unit may be **i**(nch), **c**(entimeter), or **p**(oint).

  One of the following modifiers can be appended to *gap* [Default imposes breaks based on the absolute value of the difference between the current and previous value]:

  - **+n** - specify that the previous value minus the current column value must exceed *gap* for a break to be imposed.
  - **+p** - specify that the current value minus the previous value must exceed *gap* for a break to be imposed.

- **header** (*str*) – [**i**|**o**][*n*][**+c**][**+d**][**+m***segheader*][**+r***remark*][**+t***title*]. Specify that input and/or output file(s) have *n* header records [Default is 0]. Prepend **i** if only the primary input should have header records. Prepend **o** to control the writing of header records, with the following modifiers supported:

  - **+d** to remove existing header records.
  - **+c** to add a header comment with column names to the output [Default is no column names].
  - **+m** to add a segment header *segheader* to the output after the header block [Default is no segment header].
  - **+r** to add a *remark* comment to the output [Default is no comment]. The *remark* string may contain \n to indicate line-breaks.
  - **+t** to add a *title* comment to the output [Default is no title]. The *title* string may contain \n to indicate line-breaks.

  Blank lines and lines starting with # are always skipped.

- **incols** (*str or 1d array*) – Specify data columns for primary input in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default reads all columns in order, starting with the first (i.e., column 0)].

  - For *1d array*: specify individual columns in input order (e.g., `incols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `incols="0:2,4+l"` to input the first three columns followed by the log-transformed 5th column). To read from a given column until the end of the record, leave off *stop* when specifying the column range. To read trailing text, add the column **t**. Append the word number to **t** to ingest only a single word from the trailing text. Instead of specifying columns, use `incols="n"` to simply read numerical input and skip trailing text. Optionally, append one of the following modifiers to any column or column range to transform the input columns:

    * **+l** to take the *log10* of the input values.
    * **+d** to divide the input values by the factor *divisor* [Default is 1].
    * **+s** to multiple the input values by the factor *scale* [Default is 1].
    * **+o** to add the given *offset* to the input values [Default is 0].

- **label** (*str*) – Add a legend entry for the symbol or line being plotted. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#l-full.
- **perspective** (*list or str*) – [**x**|**y**|**z**]*azim*[/*elev*[/*zlevel*]][**+w***lon0*/*lat0*[/*z0*]][**+v***x0*/*y0*]. Select perspective view and set the azimuth and elevation angle of the viewpoint. Default is [180, 90]. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#perspective-full.
- **transparency** (*int or float*) – Set transparency level, in [0-100] percent range. Default is 0, i.e., opaque. Only visible when PDF or raster format output is selected. Only the PNG format selection adds a transparency layer in the image (for further processing). *transparency* can also be a 1d array to set varying transparency for symbols, but this option is only valid if using x/y.
- **wrap** (*str*) – **y**|**a**|**w**|**d**|**h**|**m**|**s**|**c***period*[/*phase*][**+c***col*]. Convert the input *x*-coordinate to a cyclical coordinate, or a different column if selected via **+c***col*. The following cyclical coordinate transformations are supported:
  - **y** - yearly cycle (normalized)
  - **a** - annual cycle (monthly)
  - **w** - weekly cycle (day)
  - **d** - daily cycle (hour)
  - **h** - hourly cycle (minute)
  - **m** - minute cycle (second)
  - **s** - second cycle (second)
  - **c** - custom cycle (normalized)

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#w-full.

Examples using `pygmt.Figure.plot`

- *Cartesian, circular, and geographic vectors*
- *Line colors with a custom CPT*
- *Line fronts*
- *Line styles*
- *Roads*
- *Vector heads and tails*
- *Basic geometric symbols*
- *Color points by categories*
- *Custom symbols*
- *Datetime inputs*

- *Multi-parameter symbols*
- *Points*
- *Points with varying transparency*
- *Scatter plots with a legend*
- *Text symbols*
- *Sampling along tracks*
- *Double Y axes graph*
- *Inset map showing a rectangular region*
- *Legend*
- *Plotting data points*
- *Plotting datetime charts*
- *Plotting lines*
- *Plotting vectors*
- *Cartesian linear*
- *Cartesian logarithmic*
- *Cartesian power*

pygmt.Figure.plot3d

`Figure.plot3d`(*data=None*, *x=None*, *y=None*, *z=None*, *size=None*, *direction=None*, ***, *straight_line=None*, *frame=None*, *cmap=None*, *offset=None*, *color=None*, *intensity=None*, *projection=None*, *zscale=None*, *zsize=None*, *close=None*, *no_clip=None*, *no_sort=None*, *region=None*, *style=None*, *verbose=None*, *pen=None*, *xshift=None*, *yshift=None*, *zvalue=None*, *aspatial=None*, *binary=None*, *panel=None*, *nodata=None*, *find=None*, *coltypes=None*, *gap=None*, *header=None*, *incols=None*, *label=None*, *perspective=None*, *transparency=None*, *wrap=None*, ***kwargs*)

Plot lines, polygons, and symbols in 3-D.

Takes a matrix, (x,y,z) triplets, or a file name as input and plots lines, polygons, or symbols at those locations in 3-D.

Must provide either `data` or `x/y/z`.

If providing data through `x/y/z`, `color` can be a 1d array that will be mapped to a colormap.

If a symbol is selected and no symbol size given, then plot3d will interpret the fourth column of the input data as symbol size. Symbols whose size is <= 0 are skipped. If no symbols are specified then the symbol code (see `style` below) must be present as last column in the input. If `style` is not used, a line connecting the data points will be drawn instead. To explicitly close polygons, use `close`. Select a fill with `color`. If `color` is set, `pen` will control whether the polygon outline is drawn or not. If a symbol is selected, `color` and `pen` determines the fill and outline/no outline, respectively.

Full parameter list at https://docs.generic-mapping-tools.org/latest/plot3d.html

Aliases:

- A = straight_line
- B = frame
- C = cmap

- D = offset
- G = color
- I = intensity
- J = projection
- JZ = zsize
- Jz = zscale
- L = close
- N = no_clip
- Q = no_sort
- R = region
- S = style
- V = verbose
- W = pen
- X = xshift
- Y = yshift
- Z = zvalue
- a = aspatial
- b = binary
- c = panel
- d = nodata
- e = find
- f = coltypes
- g = gap
- h = header
- i = incols
- l = label
- p = perspective
- t = transparency
- w = wrap

Parameters

- **data** (*str or numpy.ndarray or pandas.DataFrame or xarray.Dataset or geopandas.GeoDataFrame*) – Either a data file name, a 2d `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data. Optionally, use parameter `incols` to specify which columns are x, y, z, color, and size, respectively.
- **x/y/z** (*float or 1d arrays*) – The x, y, and z coordinates, or arrays of x, y and z coordinates of the data points

- **size** (*1d array*) – The size of the data points in units specified in `style`. Only valid if using `x`/`y`/`z`.
- **direction** (*list of two 1d arrays*) – If plotting vectors (using `style='V'` or `style='v'`), then should be a list of two 1d arrays with the vector directions. These can be angle and length, azimuth and length, or x and y components, depending on the style options chosen.
- **projection** (*str*) – *projcode*[*projparams/*]*width*. Select map *projection*.
- **zscale/zsize** (*float or str*) – Set z-axis scaling or z-axis size.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **straight_line** (*bool or str*) – [**m**|**p**|**x**|**y**]. By default, geographic line segments are drawn as great circle arcs. To draw them as straight lines, use `straight_line`. Alternatively, add **m** to draw the line by first following a meridian, then a parallel. Or append **p** to start following a parallel, then a meridian. (This can be practical to draw a line along parallels, for example). For Cartesian data, points are simply connected, unless you append **x** or **y** to draw stair-case curves that whose first move is along *x* or *y*, respectively. **Note**: The `straight_line` parameter requires constant *z*-coordinates.
- **frame** (*bool or str or list*) – Set map boundary *frame and axes attributes*.
- **cmap** (*str*) – File name of a CPT file or a series of comma-separated colors (e.g., *color1*,*color2*,*color3*) to build a linear continuous CPT from those colors automatically.
- **offset** (*str*) – *dx*/*dy*[/*dz*]. Offset the plot symbol or line locations by the given amounts *dx*/*dy*[/*dz*] [Default is no offset].
- **color** (*str or 1d array*) – Select color or pattern for filling of symbols or polygons. Default is no fill. *color* can be a 1d array, but it is only valid if using `x`/`y` and `cmap=True` is also required.
- **intensity** (*float or bool or 1d array*) – Provide an *intensity* value (nominally in the -1 to +1 range) to modulate the fill color by simulating illumination. If using `intensity=True`, we will instead read *intensity* from the first data column after the symbol parameters (if given). *intensity* can also be a 1d array to set varying intensity for symbols, but it is only valid for `x`/`y`/`z`.
- **close** (*str*) – [**+b**|**d**|**D**][**+xl**|**r**|*x0*][**+yl**|**r**|*y0*][**+p***pen*]. Force closed polygons. Full documentation is at https://docs.generic-mapping-tools.org/latest/plot3d.html#l.
- **no_clip** (*bool or str*) – [**c**|**r**]. Do NOT clip symbols that fall outside map border [Default plots points whose coordinates are strictly inside the map border only]. This parameter does not apply to lines and polygons which are always clipped to the map region. For periodic (360-longitude) maps we must plot all symbols twice in case they are clipped by the repeating boundary. `no_clip=True` will turn off clipping and not plot repeating symbols. Use `no_clip="r"` to turn off clipping but retain the plotting of such repeating symbols, or use `no_clip="c"` to retain clipping but turn off plotting of repeating symbols.
- **no_sort** (*bool*) – Turn off the automatic sorting of items based on their distance from the viewer. The default is to sort the items so that items in the foreground are plotted after items in the background.
- **style** (*str*) – Plot symbols. Full documentation is at https://docs.generic-mapping-tools.org/latest/plot3d.html#s.
- **timestamp** (*bool or str*) – Draw GMT time stamp logo on plot.

- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **pen** (*str*) – Set pen attributes for lines or the outline of symbols.
- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.
- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.
- **zvalue** (*str*) – *value*|*file*. Instead of specifying a symbol or polygon fill and outline color via `color` and `pen`, give both a *value* via **zvalue** and a color lookup table via `cmap`. Alternatively, give the name of a *file* with one z-value (read from the last column) for each polygon in the input data. To apply it to the fill color, use `color='+z'`. To apply it to the pen color, append **+z** to `pen`.
- **aspatial** (*bool or str*) – [*col*=]*name*[,. . .]. Control how aspatial data are handled during input and output. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#aspatial-full.
- **binary** (*bool or str*) – **i**|**o**[*ncols*][*type*][**w**][**+l**|**b**]. Select native binary input (using `binary="i"`) or output (using `binary="o"`), where *ncols* is the number of data columns of *type*, which must be one of:
  - **c** - int8_t (1-byte signed char)
  - **u** - uint8_t (1-byte unsigned char)
  - **h** - int16_t (2-byte signed int)
  - **H** - uint16_t (2-byte unsigned int)
  - **i** - int32_t (4-byte signed int)
  - **I** - uint32_t (4-byte unsigned int)
  - **l** - int64_t (8-byte signed int)
  - **L** - uint64_t (8-byte unsigned int)
  - **f** - 4-byte single-precision float
  - **d** - 8-byte double-precision float
  - **x** - use to skip *ncols* anywhere in the record

  For records with mixed types, append additional comma-separated combinations of *ncols* *type* (no space). The following modifiers are supported:
  - **w** after any item to force byte-swapping.
  - **+l**|**b** to indicate that the entire data file should be read as little- or big-endian, respectively.

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#bi-full.

- **panel** (*bool or int or list*) – [*row,col*|*index*]. Select a specific subplot panel. Only allowed when in subplot mode. Use `panel=True` to advance to the next panel in the selected order. Instead of *row,col* you may also give a scalar value *index* which depends on the order you set via `autolabel` when the subplot was defined. **Note**: *row*, *col*, and *index* all start at 0.
- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `d="-9999"` will replace all values equal to -9999 with NaN during input and all NaN values with -9999 during output. Prepend **i** to the *nodata* value for input columns only. Prepend **o** to the *nodata* value for output columns only.
- **find** (*str*) – [**~**]*"pattern"* | [**~**]/*regexp*/[**i**]. Only pass records that match the given *pattern* or regular expressions [Default processes all records]. Prepend **~** to the *pattern* or *regexp* to instead only pass data expressions that do not match the pattern. Append **i** for case insensitive matching. This does not apply to headers or segment headers.
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **gap** (*str or list*) – [**a**]**x**|**y**|**d**|**X**|**Y**|**D**|[*col*]**z***gap*[**+n**|**p**]. Examine the spacing between consecutive data points in order to impose breaks in the line. To specify multiple criteria, provide a list with each item containing a string describing one set of criteria. Prepend **a** to specify that all the criteria must be met [Default is to impose breaks if any criteria are met]. The following modifiers are supported:
  - **x**|**X** - define a gap when there is a large enough change in the x coordinates (upper case to use projected coordinates).
  - **y**|**Y** - define a gap when there is a large enough change in the y coordinates (upper case to use projected coordinates).
  - **d**|**D** - define a gap when there is a large enough distance between coordinates (upper case to use projected coordinates).
  - [*col*]**z** - define a gap when there is a large enough change in the data in column *col* [default *col* is 2 (i.e., 3rd column)].

  A unit **u** may be appended to the specified *gap*:

  - For geographic data (**x**|**y**|**d**), the unit may be arc **d**(egree), **m**(inute), and **s**(econd), or (m)**e**(ter), **f**(eet), **k**(ilometer), **M**(iles), or **n**(autical miles) [Default is (m)**e**(ter)].
  - For projected data (**X**|**Y**|**D**), the unit may be **i**(nch), **c**(entimeter), or **p**(oint).

  One of the following modifiers can be appended to *gap* [Default imposes breaks based on the absolute value of the difference between the current and previous value]:

  - **+n** - specify that the previous value minus the current column value must exceed *gap* for a break to be imposed.
  - **+p** - specify that the current value minus the previous value must exceed *gap* for a break to be imposed.
- **header** (*str*) – [**i**|**o**][*n*][**+c**][**+d**][**+m***segheader*][**+r***remark*][**+t***title*]. Specify that input and/or output file(s) have *n* header records [Default is 0]. Prepend **i** if only the primary input should have header records. Prepend **o** to control the writing of header records, with the following modifiers supported:
  - **+d** to remove existing header records.

  - **+c** to add a header comment with column names to the output [Default is no column names].
  - **+m** to add a segment header *segheader* to the output after the header block [Default is no segment header].
  - **+r** to add a *remark* comment to the output [Default is no comment]. The *remark* string may contain \n to indicate line-breaks.
  - **+t** to add a *title* comment to the output [Default is no title]. The *title* string may contain \n to indicate line-breaks.

  Blank lines and lines starting with # are always skipped.

- **incols** (*str or 1d array*) – Specify data columns for primary input in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default reads all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in input order (e.g., `incols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `incols="0:2,4+l"` to input the first three columns followed by the log-transformed 5th column). To read from a given column until the end of the record, leave off *stop* when specifying the column range. To read trailing text, add the column **t**. Append the word number to **t** to ingest only a single word from the trailing text. Instead of specifying columns, use `incols="n"` to simply read numerical input and skip trailing text. Optionally, append one of the following modifiers to any column or column range to transform the input columns:
    - **+l** to take the *log10* of the input values.
    - **+d** to divide the input values by the factor *divisor* [Default is 1].
    - **+s** to multiple the input values by the factor *scale* [Default is 1].
    - **+o** to add the given *offset* to the input values [Default is 0].
- **label** (*str*) – Add a legend entry for the symbol or line being plotted. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#l-full.
- **perspective** (*list or str*) – [**x**|**y**|**z**]*azim*[/*elev*[/*zlevel*]][**+w***lon0*/*lat0*[/*z0*]][**+v***x0*/*y0*]. Select perspective view and set the azimuth and elevation angle of the viewpoint. Default is [180, 90]. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#perspective-full.
- **transparency** (*int or float*) – Set transparency level, in [0-100] percent range. Default is 0, i.e., opaque. Only visible when PDF or raster format output is selected. Only the PNG format selection adds a transparency layer in the image (for further processing). *transparency* can also be a 1d array to set varying transparency for symbols, but this option is only valid if using x/y/z.
- **wrap** (*str*) – **y**|**a**|**w**|**d**|**h**|**m**|**s**|**c***period*[/*phase*][**+c***col*]. Convert the input *x*-coordinate to a cyclical coordinate, or a different column if selected via **+c***col*. The following cyclical coordinate transformations are supported:
  - **y** - yearly cycle (normalized)
  - **a** - annual cycle (monthly)
  - **w** - weekly cycle (day)

- **d** - daily cycle (hour)
- **h** - hourly cycle (minute)
- **m** - minute cycle (second)
- **s** - second cycle (second)
- **c** - custom cycle (normalized)

Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#w-full.

Examples using `pygmt.Figure.plot3d`

- *3D Scatter plots*

pygmt.Figure.rose

`Figure.`**`rose`**(*data=None*, *length=None*, *azimuth=None*, ***, *sector=None*, *frame=None*, *cmap=None*, *shift=None*, *vectors=None*, *no_scale=None*, *color=None*, *inquire=None*, *diameter=None*, *labels=None*, *vector_params=None*, *alpha=None*, *region=None*, *norm=None*, *orientation=None*, *timestamp=None*, *verbose=None*, *pen=None*, *xshift=None*, *yshift=None*, *scale=None*, *binary=None*, *nodata=None*, *find=None*, *header=None*, *incols=None*, *panel=None*, *perspective=None*, *transparency=None*, *wrap=None*, ***kwargs*)

Plot windrose diagrams or polar histograms.

Takes a matrix, (length,azimuth) pairs, or a file name as input and plots windrose diagrams or polar histograms (sector diagram or rose diagram).

Must provide either `data` or `length` and `azimuth`.

Options include full circle and half circle plots. The outline of the windrose is drawn with the same color as MAP_DEFAULT_PEN.

Full option list at https://docs.generic-mapping-tools.org/latest/rose.html

Aliases:

- A = sector
- B = frame
- C = cmap
- D = shift
- Em = vectors
- F = no_scale
- G = color
- I = inquire
- JX = diameter
- L = labels
- M = vector_params
- Q = alpha
- R = region

- S = norm
- T = orientation
- U = timestamp
- V = verbose
- W = pen
- X = xshift
- Y = yshift
- Z = scale
- b = binary
- c = panel
- d = nodata
- e = find
- h = header
- i = incols
- p = perspective
- t = transparency
- w = wrap

Parameters

- **data** (*str or numpy.ndarray or pandas.DataFrame or xarray.Dataset or geopandas.GeoDataFrame*) – Pass in either a file name to an ASCII data table, a 2D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data. Use option `columns` to choose which columns are length and azimuth, respectively. If a file with only azimuths is given, use `columns` to indicate the single column with azimuths; then all lengths are set to unity (see `scale = 'u'` to set actual lengths to unity as well).
- **length/azimuth** (*float or 1d arrays*) – Length and azimuth values, or arrays of length and azimuth values
- **orientation** (*bool*) – Specifies that the input data are orientation data (i.e., have a 180 degree ambiguity) instead of true 0-360 degree directions [Default is 0-360 degrees]. We compensate by counting each record twice: First as azimuth and second as azimuth +180. Ignored if range is given as -90/90 or 0/180.
- **region** (*str or list*) – *Required if this is the first plot command. r0/r1/az0/az1*. Specifies the 'region' of interest in (*r*, *azimuth*) space. Here, *r0* is 0, *r1* is max length in units. For *az0* and *az1*, specify either -90/90 or 0/180 for half circle plot or 0/360 for full circle.
- **diameter** (*str*) – Sets the diameter of the rose diagram. If not given, then we default to a diameter of 7.5 cm.
- **sector** (*str*) – Gives the sector width in degrees for sector and rose diagram. Default `'0'` means windrose diagram. Append **+r** to draw rose diagram instead of sector diagram (e.g. `'10+r'`).
- **norm** (*bool*) – Normalize input radii (or bin counts if `sector_width` is used) by the largest value so all radii (or bin counts) range from 0 to 1.

- **frame** (*str*) – Set map boundary frame and axes attributes. Remember that *x* here is radial distance and *y* is azimuth. The ylabel may be used to plot a figure caption. The scale bar length is determined by the radial gridline spacing.
- **scale** (*float or str*) – Multiply the data radii by scale. E.g., use `scale = 0.001` to convert your data from m to km. To exclude the radii from consideration, set them all to unity with `scale = 'u'` [Default is no scaling].
- **color** (*str*) – Selects shade, color or pattern for filling the sectors [Default is no fill].
- **cmap** (*str*) – Give a CPT. The *r*-value for each sector is used to look-up the sector color. Cannot be used with a rose diagram.
- **pen** (*str*) – Set pen attributes for sector outline or rose plot, e.g. `pen = '0.5p'`. [Default is no outline]. To change pen used to draw vector (requires `vectors`) [Default is same as sector outline] use e.g. `pen = 'v0.5p'`.
- **labels** (*str*) – `'wlabel,elabel,slabel,nlabel'`. Specify labels for the 0, 90, 180, and 270 degree marks. For full-circle plot the default is WEST,EAST,SOUTH,NORTH and for half-circle the default is 90W,90E,-,0. A **-** in any entry disables that label (e.g. `labels = 'W,E,-,N'`). Use `labels = ''` to disable all four labels. Note that the GMT_LANGUAGE setting will affect the words used.
- **no_scale** (*bool*) – Do NOT draw the scale length bar (`no_scale = True`). Default plots scale in lower right corner provided `frame` is used.
- **shift** (*bool*) – Shift sectors so that they are centered on the bin interval (e.g., first sector is centered on 0 degrees).
- **vectors** (*str*) – `vectors = 'mode_file'`. Plot vectors showing the principal directions given in the *mode_file* file. Alternatively, specify `vectors` to compute and plot mean direction. See `vector_params` to control the vector attributes. Finally, to instead save the computed mean direction and other statistics, use `vectors = '+wmode_file'`. The eight items saved to a single record are: *mean_az*, *mean_r*, *mean_resultant*, *max_r*, *scaled_mean_r*, *length_sum*, *n*, *sign@alpha*, where the last term is 0 or 1 depending on whether the mean resultant is significant at the level of confidence set via `alpha`.
- **vector_params** (*str*) – Used with `vectors` to modify vector parameters. For vector heads, append vector head size [Default is 0, i.e., a line]. See https://docs.generic-mapping-tools.org/latest/rose.html#vector-attributes for specifying additional attributes. If `vectors` is not given and the current plot mode is to draw a windrose diagram then using `vector_params` will add vector heads to all individual directions using the supplied attributes.
- **alpha** (*float or str*) – Sets the confidence level used to determine if the mean resultant is significant (i.e., Lord Rayleigh test for uniformity) [`alpha = 0.05`]. Note: The critical values are approximated [Berens, 2009] and requires at least 10 points; the critical resultants are accurate to at least 3 significant digits. For smaller data sets you should consult exact statistical tables.

  Berens, P., 2009, CircStat: A MATLAB Toolbox for Circular Statistics, *J. Stat. Software*, 31(10), 1-21, https://doi.org/10.18637/jss.v031.i10.
- **timestamp** (*bool or str*) – Draw GMT time stamp logo on plot.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]

  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.
- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.
- **binary** (*bool or str*) – **i**|**o**[*ncols*][*type*][**w**][**+l**|**b**]. Select native binary input (using `binary="i"`) or output (using `binary="o"`), where *ncols* is the number of data columns of *type*, which must be one of:
  - **c** - int8_t (1-byte signed char)
  - **u** - uint8_t (1-byte unsigned char)
  - **h** - int16_t (2-byte signed int)
  - **H** - uint16_t (2-byte unsigned int)
  - **i** - int32_t (4-byte signed int)
  - **I** - uint32_t (4-byte unsigned int)
  - **l** - int64_t (8-byte signed int)
  - **L** - uint64_t (8-byte unsigned int)
  - **f** - 4-byte single-precision float
  - **d** - 8-byte double-precision float
  - **x** - use to skip *ncols* anywhere in the record

  For records with mixed types, append additional comma-separated combinations of *ncols type* (no space). The following modifiers are supported:
  - **w** after any item to force byte-swapping.
  - **+l**|**b** to indicate that the entire data file should be read as little- or big-endian, respectively.

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#bi-full.
- **panel** (*bool or int or list*) – [*row,col*|*index*]. Select a specific subplot panel. Only allowed when in subplot mode. Use `panel=True` to advance to the next panel in the selected order. Instead of *row,col* you may also give a scalar value *index* which depends on the order you set via `autolabel` when the subplot was defined. **Note**: *row*, *col*, and *index* all start at 0.
- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `d="-9999"` will replace all values equal to -9999 with NaN during input and all NaN values with -9999 during output. Prepend **i** to the *nodata* value for input columns only. Prepend **o** to the *nodata* value for output columns only.
- **find** (*str*) – [**~**]*"pattern"* | [**~**]/*regexp*/[**i**]. Only pass records that match the given *pattern* or regular expressions [Default processes all records]. Prepend **~** to the *pattern* or *regexp* to instead only pass data expressions that do not match the pattern. Append **i** for case insensitive matching. This does not apply to headers or segment headers.
- **header** (*str*) – [**i**|**o**][*n*][**+c**][**+d**][**+m***segheader*][**+r***remark*][**+t***title*]. Specify that input and/or output file(s) have *n* header records [Default is 0]. Prepend **i** if only the primary

input should have header records. Prepend **o** to control the writing of header records, with the following modifiers supported:

- **+d** to remove existing header records.
- **+c** to add a header comment with column names to the output [Default is no column names].
- **+m** to add a segment header *segheader* to the output after the header block [Default is no segment header].
- **+r** to add a *remark* comment to the output [Default is no comment]. The *remark* string may contain \n to indicate line-breaks.
- **+t** to add a *title* comment to the output [Default is no title]. The *title* string may contain \n to indicate line-breaks.

Blank lines and lines starting with # are always skipped.

- **incols** (*str or 1d array*) – Specify data columns for primary input in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default reads all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in input order (e.g., `incols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `incols="0:2,4+l"` to input the first three columns followed by the log-transformed 5th column). To read from a given column until the end of the record, leave off *stop* when specifying the column range. To read trailing text, add the column **t**. Append the word number to **t** to ingest only a single word from the trailing text. Instead of specifying columns, use `incols="n"` to simply read numerical input and skip trailing text. Optionally, append one of the following modifiers to any column or column range to transform the input columns:
    - **+l** to take the *log10* of the input values.
    - **+d** to divide the input values by the factor *divisor* [Default is 1].
    - **+s** to multiple the input values by the factor *scale* [Default is 1].
    - **+o** to add the given *offset* to the input values [Default is 0].
- **perspective** (*list or str*) – [**x**|**y**|**z**]*azim*[/*elev*[/*zlevel*]][**+w***lon0*/*lat0*[/*z0*]][**+v***x0*/*y0*]. Select perspective view and set the azimuth and elevation angle of the viewpoint. Default is [180, 90]. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#perspective-full.
- **transparency** (*int or float*) – Set transparency level, in [0-100] percent range. Default is 0, i.e., opaque. Only visible when PDF or raster format output is selected. Only the PNG format selection adds a transparency layer in the image (for further processing).
- **wrap** (*str*) – **y**|**a**|**w**|**d**|**h**|**m**|**s**|**c***period*[/*phase*][**+c***col*]. Convert the input *x*-coordinate to a cyclical coordinate, or a different column if selected via **+c***col*. The following cyclical coordinate transformations are supported:
  - **y** - yearly cycle (normalized)
  - **a** - annual cycle (monthly)
  - **w** - weekly cycle (day)
  - **d** - daily cycle (hour)

- **h** - hourly cycle (minute)
- **m** - minute cycle (second)
- **s** - second cycle (second)
- **c** - custom cycle (normalized)

Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#w-full.

Examples using `pygmt.Figure.rose`

- *Rose diagram*

pygmt.Figure.set_panel

`Figure.set_panel`(*panel=None*, *, *fixedlabel=None*, *clearance=None*, *verbose=None*, ***kwargs*)

Set the current subplot panel to plot on.

Before you start plotting you must first select the active subplot. Note: If any *projection* option is passed with the question mark **?** as scale or width when plotting subplots, then the dimensions of the map are automatically determined by the subplot size and your region. For Cartesian plots: If you want the scale to apply equally to both dimensions then you must specify `projection="x"` [The default `projection="X"` will fill the subplot by using unequal scales].

Aliases:

- A = fixedlabel
- C = clearance
- V = verbose

Parameters

- **panel** (*str or list*) – *row,col*|*index*. Sets the current subplot until further notice. **Note**: First *row* or *col* is 0, not 1. If not given we go to the next subplot by order specified via `autolabel` in *pygmt.Figure.subplot*. As an alternative, you may bypass using *pygmt.Figure.set_panel* and instead supply the common option **panel**=[*row,col*] to the first plot command you issue in that subplot. GMT maintains information about the current figure and subplot. Also, you may give the one-dimensional *index* instead which starts at 0 and follows the row or column order set via `autolabel` in *pygmt.Figure.subplot*.
- **fixedlabel** (*str*) – Overrides the automatic labeling with the given string. No modifiers are allowed. Placement, justification, etc. are all inherited from how `autolabel` was specified by the initial *pygmt.Figure.subplot* command.
- **clearance** (*str or list*) – [*side*]*clearance*. Reserve a space of dimension *clearance* between the margin and the subplot on the specified side, using *side* values from **w**, **e**, **s**, or **n**. The option is repeatable to set aside space on more than one side (e.g. `clearance=['w1c', 's2c']` would set a clearance of 1 cm on west side and 2 cm on south side). Such space will be left untouched by the main map plotting but can be accessed by modules that plot scales, bars, text, etc. This setting overrides the common clearances set by `clearance` in the initial *pygmt.Figure.subplot* call.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced

- **e** - Error messages only
- **w** - Warnings [Default]
- **t** - Timings (report runtimes for time-intensive algorithms);
- **i** - Informational messages (same as `verbose=True`)
- **c** - Compatibility warnings
- **d** - Debugging messages

Examples using `pygmt.Figure.set_panel`

- *Multiple colormaps*
- *Making subplots*

pygmt.Figure.shift_origin

`Figure.`**`shift_origin`**(*xshift=None*, *yshift=None*)

Shift plot origin in x and/or y directions.

This method shifts plot origin relative to the current origin by (*xshift*, *yshift*) and optionally append the length unit (**c**, **i**, or **p**).

Prepend **a** to shift the origin back to the original position after plotting, prepend **c** to center the plot on the center of the paper (optionally add shift), prepend **f** to shift the origin relative to the fixed lower left corner of the page, or prepend **r** [Default] to move the origin relative to its current location.

Detailed usage at https://docs.generic-mapping-tools.org/latest/cookbook/options.html#plot-positioning-and-layout-the-x-y-options

Parameters

- **`xshift`** (*`str`*) – Shift plot origin in x direction.
- **`yshift`** (*`str`*) – Shift plot origin in y direction.

Examples using `pygmt.Figure.shift_origin`

- *Calculating grid gradient and radiance*
- *Clipping grid values*
- *Coastlines and borders*
- *Configuring PyGMT defaults*
- *Making subplots*
- *Polar*

pygmt.Figure.solar

`Figure.solar`(*terminator='d'*, *terminator_datetime=None*, ***, *frame=None*, *fill=None*, *projection=None*, *region=None*, *timestamp=None*, *verbose=None*, *pen=None*, *xshift=None*, *yshift=None*, *panel=None*, *perspective=None*, *transparency=None*, ***kwargs*)

Plot day-light terminators or twilights.

This function plots the day-night terminator. Alternatively, it can plot the terminators for civil twilight, nautical twilight, or astronomical twilight.

Full parameter list at https://docs.generic-mapping-tools.org/latest/solar.html

Aliases:

- B = frame
- G = fill
- J = projection
- R = region
- U = timestamp
- V = verbose
- W = pen
- X = xshift
- Y = yshift
- c = panel
- p = perspective
- t = transparency

Parameters

- **terminator** (*str*) – Set the type of terminator displayed. Valid arguments are **day_night**, **civil**, **nautical**, and **astronomical**, which can be set with either the full name or the first letter of the name. [Default is **day_night**]

  Refer to https://en.wikipedia.org/wiki/Twilight for the definitions of different types of twilight.

- **terminator_datetime** (*str or datetime object*) – Set the UTC date and time of the displayed terminator. It can be passed as a string or Python datetime object. [Default is the current UTC date and time]
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **projection** (*str*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- **frame** (*bool or str or list*) – Set map boundary *frame and axes attributes*.
- **fill** (*str*) – Color or pattern for filling of terminators.
- **pen** (*str*) – Set pen attributes for lines. The default pen is `default,black,solid`.
- **timestamp** (*bool or str*) – Draw GMT time stamp logo on plot.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:

  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.
- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.
- **panel** (*bool or int or list*) – [*row,col*|*index*]. Select a specific subplot panel. Only allowed when in subplot mode. Use `panel=True` to advance to the next panel in the selected order. Instead of *row,col* you may also give a scalar value *index* which depends on the order you set via `autolabel` when the subplot was defined. **Note**: *row*, *col*, and *index* all start at 0.
- **perspective** (*list or str*) – [**x**|**y**|**z**]*azim*[/*elev*[/*zlevel*]][**+w***lon0*/*lat0*[/*z0*]][**+v***x0*/*y0*]. Select perspective view and set the azimuth and elevation angle of the viewpoint. Default is [180, 90]. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#perspective-full.
- **transparency** (*int or float*) – Set transparency level, in [0-100] percent range. Default is 0, i.e., opaque. Only visible when PDF or raster format output is selected. Only the PNG format selection adds a transparency layer in the image (for further processing).

Examples using `pygmt.Figure.solar`

- *Day-night terminator line and twilights*

pygmt.Figure.subplot

`Figure.subplot`(*nrows=1*, *ncols=1*, ***, *figsize=None*, *subsize=None*, *autolabel=None*, *frame=None*, *clearance=None*, *projection=None*, *margins=None*, *region=None*, *sharex=None*, *sharey=None*, *title=None*, *verbose=None*, *xshift=None*, *yshift=None*, ***kwargs*)

Create multi-panel subplot figures.

This function is used to split the current figure into a rectangular layout of subplots that each may contain a single self-contained figure. Begin by defining the layout of the entire multi-panel illustration. Several parameters are available to specify the systematic layout, labeling, dimensions, and more for the subplots.

Full option list at https://docs.generic-mapping-tools.org/latest/subplot.html#synopsis-begin-mode

Aliases:

- A = autolabel
- B = frame
- C = clearance
- Ff = figsize
- Fs = subsize

- J = projection
- M = margins
- R = region
- SC = sharex
- SR = sharey
- T = title
- V = verbose
- X = xshift
- Y = yshift

Parameters

- **nrows** (*int*) – Number of vertical rows of the subplot grid.
- **ncols** (*int*) – Number of horizontal columns of the subplot grid.
- **figsize** (*tuple*) – Specify the final figure dimensions as (*width*, *height*).
- **subsize** (*tuple*) – Specify the dimensions of each subplot directly as (*width*, *height*). Note that only one of `figsize` or `subsize` can be provided at once.
- **autolabel** (*bool or str*) – [*autolabel*][**+c***dx*[/*dy*]][**+g***fill*][**+j**|**J***refpoint*][**+o***dx*[/*dy*]][**+p***pen*][**+r**|**R**] [**+v**]. Specify automatic tagging of each subplot. Append either a number or letter [a]. This sets the tag of the first, top-left subplot and others follow sequentially. Surround the number or letter by parentheses on any side if these should be typeset as part of the tag. Use **+j**|**J***refpoint* to specify where the tag should be placed in the subplot [TL]. Note: **+j** sets the justification of the tag to *refpoint* (suitable for interior tags) while **+J** instead selects the mirror opposite (suitable for exterior tags). Append **+c***dx*[/*dy*] to set the clearance between the tag and a surrounding text box requested via **+g** or **+p** [3p/3p, i.e., 15% of the FONT_TAG size dimension]. Append **+g***fill* to paint the tag's text box with *fill* [no painting]. Append **+o***dx*[/*dy*] to offset the tag's reference point in the direction implied by the justification [4p/4p, i.e., 20% of the FONT_TAG size]. Append **+p***pen* to draw the outline of the tag's text box using selected *pen* [no outline]. Append **+r** to typeset your tag numbers using lowercase Roman numerals; use **+R** for uppercase Roman numerals [Arabic numerals]. Append **+v** to increase tag numbers vertically down columns [horizontally across rows].
- **frame** (*bool or str or list*) – Set map boundary *frame and axes attributes*.
- **clearance** (*str or list*) – [*side*]*clearance*. Reserve a space of dimension *clearance* between the margin and the subplot on the specified side, using *side* values from **w**, **e**, **s**, or **n**; or **x** for both **w** and **e**; or **y** for both **s** and **n**. No *side* means all sides (i.e. `clearance='1c'` would set a clearance of 1 cm on all sides). The option is repeatable to set aside space on more than one side (e.g. `clearance=['w1c', 's2c']` would set a clearance of 1 cm on west side and 2 cm on south side). Such space will be left untouched by the main map plotting but can be accessed by modules that plot scales, bars, text, etc.
- **projection** (*str*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- **margins** (*str or list*) – This is margin space that is added between neighboring subplots (i.e., the interior margins) in addition to the automatic space added for tick marks, annotations, and labels. The margins can be specified as either:
  - a single value (for same margin on all sides). E.g. '5c'.

  - a pair of values (for setting separate horizontal and vertical margins). E.g. [‘5c’, ‘3c’].
  - a set of four values (for setting separate left, right, bottom, and top margins). E.g. [‘1c’, ‘2c’, ‘3c’, ‘4c’].

  The actual gap created is always a sum of the margins for the two opposing sides (e.g., east plus west or south plus north margins) [Default is half the primary annotation font size, giving the full annotation font size as the default gap].

- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **sharex** (*bool or str*) – Set subplot layout for shared x-axes. Use when all subplots in a column share a common *x*-range. If `sharex=True`, the first (i.e., **t**op) and the last (i.e., **b**ottom) rows will have *x*-annotations; use `sharex='t'` or `sharex='b'` to select only one of those two rows [both]. Append **+l** if annotated *x*-axes should have a label [none]; optionally append the label if it is the same for the entire subplot. Append **+t** to make space for subplot titles for each row; use **+tc** for top row titles only [no subplot titles].
- **sharey** (*bool or str*) – Set subplot layout for shared y-axes. Use when all subplots in a row share a common *y*-range. If `sharey=True`, the first (i.e., **l**eft) and the last (i.e., **r**ight) columns will have *y*-annotations; use `sharey='l'` or `sharey='r'` to select only one of those two columns [both]. Append **+l** if annotated *y*-axes will have a label [none]; optionally, append the label if it is the same for the entire subplot. Append **+p** to make all annotations axis-parallel [horizontal]; if not used you may have to set `clearance` to secure extra space for long horizontal annotations.

  Notes for `sharex`/`sharey`:

  - Labels and titles that depends on which row or column are specified as usual via a subplot’s own `frame` setting.
  - Append **+w** to the `figsize` or `subsize` parameter to draw horizontal and vertical lines between interior panels using selected pen [no lines].
- **title** (*str*) – While individual subplots can have titles (see `sharex`/`sharey` or `frame`), the entire figure may also have an overarching *heading* [no heading]. Font is determined by setting FONT_HEADING.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.
- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.

Examples using `pygmt.Figure.subplot`

- *Multiple colormaps*
- *Making subplots*

pygmt.Figure.text

`Figure.text`(*textfiles=None*, *x=None*, *y=None*, *position=None*, *text=None*, *angle=None*, *font=None*, *justify=None*, ***, *region=None*, *projection=None*, *frame=None*, *clearance=None*, *offset=None*, *fill=None*, *no_clip=None*, *timestamp=None*, *verbose=None*, *pen=None*, *xshift=None*, *yshift=None*, *aspatial=None*, *panel=None*, *find=None*, *coltypes=None*, *header=None*, *incols=None*, *perspective=None*, *transparency=None*, *wrap=None*, ***kwargs*)

Plot or typeset text strings of variable size, font type, and orientation.

Must provide at least one of the following combinations as input:

- `textfiles`
- `x`/`y`, and `text`
- `position` and `text`

Full parameter list at https://docs.generic-mapping-tools.org/latest/text.html

Aliases:

- B = frame
- C = clearance
- D = offset
- G = fill
- J = projection
- N = no_clip
- R = region
- U = timestamp
- V = verbose
- W = pen
- X = xshift
- Y = yshift
- a = aspatial
- c = panel
- e = find
- f = coltypes
- h = header
- i = incols
- p = perspective
- t = transparency

- w = wrap

Parameters

- **textfiles** (*str or list*) – A text data file name, or a list of filenames containing 1 or more records with (x, y[, angle, font, justify], text).
- **x/y** (*float or 1d arrays*) – The x and y coordinates, or an array of x and y coordinates to plot the text
- **position** (*str*) – Sets reference point on the map for the text by using x,y coordinates extracted from `region` instead of providing them through `x/y`. Specify with a two letter (order independent) code, chosen from:
  - Horizontal: **L**(eft), **C**(entre), **R**(ight)
  - Vertical: **T**(op), **M**(iddle), **B**(ottom)

  For example, `position="TL"` plots the text at the Upper Left corner of the map.
- **text** (*str or 1d array*) – The text string, or an array of strings to plot on the figure
- **angle** (*int, float, str or bool*) – Set the angle measured in degrees counter-clockwise from horizontal (e.g. 30 sets the text at 30 degrees). If no angle is explicitly given (i.e. `angle=True`) then the input to `textfiles` must have this as a column.
- **font** (*str or bool*) – Set the font specification with format *size*,*font*,*color* where *size* is text size in points, *font* is the font to use, and *color* sets the font color. For example, `font="12p,Helvetica-Bold,red"` selects a 12p, red, Helvetica-Bold font. If no font info is explicitly given (i.e. `font=True`), then the input to `textfiles` must have this information in one of its columns.
- **justify** (*str or bool*) – Set the alignment which refers to the part of the text string that will be mapped onto the (x,y) point. Choose a 2 character combination of **L**, **C**, **R** (for left, center, or right) and **T**, **M**, **B** for top, middle, or bottom. E.g., **BL** for lower left. If no justification is explicitly given (i.e. `justify=True`), then the input to `textfiles` must have this as a column.
- **projection** (*str*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest. *Required if this is the first plot command.*
- **clearance** (*str*) – [*dx/dy*][**+to**|**O**|**c**|**C**]. Adjust the clearance between the text and the surrounding box [Default is 15% of the font size]. Only used if `pen` or `fill` are specified. Append the unit you want (*c* for cm, *i* for inch, or *p* for point; if not given we consult **PROJ_LENGTH_UNIT**) or *%* for a percentage of the font size. Optionally, use modifier **+t** to set the shape of the textbox when using `fill` and/or `pen`. Append lower case **o** to get a straight rectangle [Default is **o**]. Append upper case **O** to get a rounded rectangle. In paragraph mode (*paragraph*) you can also append lower case **c** to get a concave rectangle or append upper case **C** to get a convex rectangle.
- **fill** (*str*) – Sets the shade or color used for filling the text box [Default is no fill].
- **offset** (*str*) – [**j**|**J**]*dx*[/*dy*][**+v**[*pen*]]. Offsets the text from the projected (x,y) point by *dx*,*dy* [0/0]. If *dy* is not specified then it is set equal to *dx*. Use **j** to offset the text away from the point instead (i.e., the text justification will determine the direction of the shift). Using **J** will shorten diagonal offsets at corners by sqrt(2). Optionally, append **+v** which will draw a line from the original point to the shifted point; append a pen to change the attributes for this line.

- **pen** (*str*) – Sets the pen used to draw a rectangle around the text string (see `clearance`) [Default is width = default, color = black, style = solid].
- **no_clip** (*bool*) – Do NOT clip text at map boundaries [Default is will clip].
- **timestamp** (*bool or str*) – Draw GMT time stamp logo on plot.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.
- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.
- **aspatial** (*bool or str*) – [*col*=]*name*[,. . .]. Control how aspatial data are handled during input and output. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#aspatial-full.
- **panel** (*bool or int or list*) – [*row,col*|*index*]. Select a specific subplot panel. Only allowed when in subplot mode. Use `panel=True` to advance to the next panel in the selected order. Instead of *row,col* you may also give a scalar value *index* which depends on the order you set via `autolabel` when the subplot was defined. **Note**: *row*, *col*, and *index* all start at 0.
- **find** (*str*) – [**~**]*"pattern"* | [**~**]/*regexp*/[**i**]. Only pass records that match the given *pattern* or regular expressions [Default processes all records]. Prepend **~** to the *pattern* or *regexp* to instead only pass data expressions that do not match the pattern. Append **i** for case insensitive matching. This does not apply to headers or segment headers.
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **header** (*str*) – [**i**|**o**][*n*][**+c**][**+d**][**+m***segheader*][**+r***remark*][**+t***title*]. Specify that input and/or output file(s) have *n* header records [Default is 0]. Prepend **i** if only the primary input should have header records. Prepend **o** to control the writing of header records, with the following modifiers supported:
  - **+d** to remove existing header records.
  - **+c** to add a header comment with column names to the output [Default is no column names].
  - **+m** to add a segment header *segheader* to the output after the header block [Default is no segment header].
  - **+r** to add a *remark* comment to the output [Default is no comment]. The *remark* string may contain \n to indicate line-breaks.

  - **+t** to add a *title* comment to the output [Default is no title]. The *title* string may contain \n to indicate line-breaks.

  Blank lines and lines starting with # are always skipped.

- **incols** (*str or 1d array*) – Specify data columns for primary input in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default reads all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in input order (e.g., `incols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `incols="0:2,4+l"` to input the first three columns followed by the log-transformed 5th column). To read from a given column until the end of the record, leave off *stop* when specifying the column range. To read trailing text, add the column **t**. Append the word number to **t** to ingest only a single word from the trailing text. Instead of specifying columns, use `incols="n"` to simply read numerical input and skip trailing text. Optionally, append one of the following modifiers to any column or column range to transform the input columns:
    - **+l** to take the *log10* of the input values.
    - **+d** to divide the input values by the factor *divisor* [Default is 1].
    - **+s** to multiple the input values by the factor *scale* [Default is 1].
    - **+o** to add the given *offset* to the input values [Default is 0].
- **perspective** (*list or str*) – [**x**|**y**|**z**]*azim*[/*elev*[/*zlevel*]][**+w***lon0*/*lat0*[/*z0*]][**+v***x0*/*y0*]. Select perspective view and set the azimuth and elevation angle of the viewpoint. Default is [180, 90]. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#perspective-full.
- **transparency** (*int or float*) – Set transparency level, in [0-100] percent range. Default is 0, i.e., opaque. Only visible when PDF or raster format output is selected. Only the PNG format selection adds a transparency layer in the image (for further processing). *transparency* can also be a 1d array to set varying transparency for texts, but this option is only valid if using x/y/text.
- **wrap** (*str*) – **y**|**a**|**w**|**d**|**h**|**m**|**s**|**c***period*[/*phase*][**+c***col*]. Convert the input *x*-coordinate to a cyclical coordinate, or a different column if selected via **+c***col*. The following cyclical coordinate transformations are supported:
  - **y** - yearly cycle (normalized)
  - **a** - annual cycle (monthly)
  - **w** - weekly cycle (day)
  - **d** - daily cycle (hour)
  - **h** - hourly cycle (minute)
  - **m** - minute cycle (second)
  - **s** - second cycle (second)
  - **c** - custom cycle (normalized)

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#w-full.

Examples using `pygmt.Figure.text`

- *Cartesian, circular, and geographic vectors*
- *Line fronts*
- *Line styles*
- *Vector heads and tails*
- *Basic geometric symbols*
- *Custom symbols*
- *Text symbols*
- *Making subplots*
- *Plotting text*
- *Polar*

pygmt.Figure.velo

`Figure.velo`(*data=None*, ***, *vector=None*, *frame=None*, *cmap=None*, *rescale=None*, *uncertaintycolor=None*, *color=None*, *scale=None*, *shading=None*, *projection=None*, *line=None*, *no_clip=None*, *region=None*, *spec=None*, *timestamp=None*, *verbose=None*, *pen=None*, *xshift=None*, *yshift=None*, *zvalue=None*, *panel=None*, *nodata=None*, *find=None*, *header=None*, *incols=None*, *perspective=None*, *transparency=None*, ***kwargs*)

Plot velocity vectors, crosses, anisotropy bars, and wedges.

Reads data values from files, `numpy.ndarray` or `pandas.DataFrame` and plots the selected geodesy symbol on a map. You may choose from velocity vectors and their uncertainties, rotational wedges and their uncertainties, anisotropy bars, or strain crosses. Symbol fills or their outlines may be colored based on constant parameters or via color lookup tables.

Must provide `data` and `spec`.

Full option list at https://docs.generic-mapping-tools.org/latest/supplements/geodesy/velo.html

Aliases:

- A = vector
- B = frame
- C = cmap
- D = rescale
- E = uncertaintycolor
- G = color
- H = scale
- I = shading
- J = projection
- L = line
- N = no_clip
- R = region

- S = spec
- U = timestamp
- V = verbose
- W = pen
- X = xshift
- Y = yshift
- Z = zvalue
- c = panel
- d = nodata
- e = find
- h = header
- i = incols
- p = perspective
- t = transparency

Parameters

- **data** (*str or numpy.ndarray or pandas.DataFrame or xarray.Dataset or geopandas.GeoDataFrame*) – Pass in either a file name to an ASCII data table, a 2D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data. Note that text columns are only supported with file or `pandas.DataFrame` inputs.
- **spec** (*str*) – Selects the meaning of the columns in the data file and the figure to be plotted. In all cases, the scales are in data units per length unit and sizes are in length units (default length unit is controlled by PROJ_LENGTH_UNIT unless **c**, **i**, or **p** is appended).
  - **e**[*velscale*/]*confidence*[**+f***font*]

    Velocity ellipses in (N,E) convention. The *velscale* sets the scaling of the velocity arrows. If *velscale* is not given then we read it from the data file as an extra column. The *confidence* sets the 2-dimensional confidence limit for the ellipse, e.g. 0.95 for 95% confidence ellipse. Use **+f** to set the font and size of the text [Default is 9p,Helvetica,black]; give **+f**0 to deactivate labeling. The arrow will be drawn with the pen attributes specified by the `pen` option and the arrow-head can be colored via `color`. The ellipse will be filled with the color or shade specified by the `uncertaintycolor` option [Default is transparent], and its outline will be drawn if `line` is selected using the pen selected (by `pen` if not given by `line`). Parameters are expected to be in the following columns:

    * **1**,**2**: longitude, latitude of station
    * **3**,**4**: eastward, northward velocity
    * **5**,**6**: uncertainty of eastward, northward velocities (1-sigma)
    * **7**: correlation between eastward and northward components
    * **Trailing text**: name of station (optional)
  - **n**[*barscale*]

Anisotropy bars. *barscale* sets the scaling of the bars. If *barscale* is not given then we read it from the data file as an extra column. Parameters are expected to be in the following columns:

* **1**,**2**: longitude, latitude of station
* **3**,**4**: eastward, northward components of anisotropy vector

– **r**[*velscale*/]*confidence*[**+f***font*]

Velocity ellipses in rotated convention. The *velscale* sets the scaling of the velocity arrows. If *velscale* is not given then we read it from the data file as an extra column. The *confidence* sets the 2-dimensional confidence limit for the ellipse, e.g. 0.95 for 95% confidence ellipse. Use **+f** to set the font and size of the text [Default is 9p,Helvetica,black]; give **+f**0 to deactivate labeling. The arrow will be drawn with the pen attributes specified by the `pen` option and the arrow-head can be colored via `color`. The ellipse will be filled with the color or shade specified by the `uncertaintycolor` option [Default is transparent], and its outline will be drawn if `line` is selected using the pen selected (by `pen` if not given by `line`). Parameters are expected to be in the following columns:

* **1**,**2**: longitude, latitude of station
* **3**,**4**: eastward, northward velocity
* **5**,**6**: semi-major, semi-minor axes
* **7**: counter-clockwise angle, in degrees, from horizontal axis to major axis of ellipse.
* **Trailing text**: name of station (optional)

– **w**[*wedgescale*/]*wedgemag*

Rotational wedges. The *wedgescale* sets the size of the wedges. If *wedgescale* is not given then we read it from the data file as an extra column. Rotation values are multiplied by *wedgemag* before plotting. For example, setting *wedgemag* to 1.e7 works well for rotations of the order of 100 nanoradians/yr. Use `color` to set the fill color or shade for the wedge, and `uncertaintycolor` to set the color or shade for the uncertainty. Parameters are expected to be in the following columns:

* **1**,**2**: longitude, latitude of station
* **3**: rotation in radians
* **4**: rotation uncertainty in radians

– **x**[*cross_scale*]

Strain crosses. The *cross_scale* sets the size of the cross. If *cross_scale* is not given then we read it from the data file as an extra column. Parameters are expected to be in the following columns:

* **1**,**2**: longitude, latitude of station
* **3**: eps1, the most extensional eigenvalue of strain tensor, with extension taken positive.
* **4**: eps2, the most compressional eigenvalue of strain tensor, with extension taken positive.
* **5**: azimuth of eps2 in degrees CW from North.

• **projection** (*str*) – *projcode*[*projparams*/]*width*. Select map *projection*.

• **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.

- **vector** (*bool or str*) – Modify vector parameters. For vector heads, append vector head *size* [Default is 9p]. See https://docs.generic-mapping-tools.org/latest/supplements/geodesy/velo.html#vector-attributes for specifying additional attributes.
- **frame** (*bool or str or list*) – Set map boundary *frame and axes attributes*.
- **cmap** (*str*) – File name of a CPT file or a series of comma-separated colors (e.g., *color1*,*color2*,*color3*) to build a linear continuous CPT from those colors automatically.
- **rescale** (*str*) – can be used to rescale the uncertainties of velocities (`spec='e'` and `spec='r'`) and rotations (`spec='w'`). Can be combined with the `confidence` variable.
- **uncertaintycolor** (*str*) – Sets the color or shade used for filling uncertainty wedges (`spec='w'`) or velocity error ellipses (`spec='e'` or `spec='r'`). If `uncertaintycolor` is not specified, the uncertainty regions will be transparent. **Note**: Using `cmap` and `zvalue='+e'` will update the uncertainty fill color based on the selected measure in `zvalue` [magnitude error]. More details at https://docs.generic-mapping-tools.org/latest/cookbook/features.html#gfill-attrib.
- **color** (*str*) – Select color or pattern for filling of symbols [Default is no fill]. **Note**: Using `cmap` (and optionally `zvalue`) will update the symbol fill color based on the selected measure in `zvalue` [magnitude]. More details at https://docs.generic-mapping-tools.org/latest/cookbook/features.html#gfill-attrib.
- **scale** (*float or bool*) – [*scale*]. Scale symbol sizes and pen widths on a per-record basis using the *scale* read from the data set, given as the first column after the (optional) *z* and *size* columns [Default is no scaling]. The symbol size is either provided by `spec` or via the input *size* column. Alternatively, append a constant *scale* that should be used instead of reading a scale column.
- **shading** (*float or bool*) – *intens*. Use the supplied *intens* value (nominally in the -1 to +1 range) to modulate the symbol fill color by simulating illumination [Default is none]. If *intens* is not provided we will instead read the intensity from an extra data column after the required input columns determined by `spec`.
- **line** (*str*) – [*pen*[**+c**[**f**|**l**]]]. Draw lines. Ellipses and rotational wedges will have their outlines drawn using the current pen (see `pen`). Alternatively, append a separate pen to use for the error outlines. If the modifier **+cl** is appended then the color of the pen is updated from the CPT (see `cmap`). If instead modifier **+cf** is appended then the color from the cpt file is applied to error fill only [Default]. Use just **+c** to set both pen and fill color.
- **no_clip** (*bool or str*) – Do NOT skip symbols that fall outside the frame boundary specified by `region`. [Default plots symbols inside frame only].
- **timestamp** (*bool or str*) – Draw GMT time stamp logo on plot.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages

- **pen** (*str*) – [*pen*][**+c**[**f**|**l**]]. Set pen attributes for velocity arrows, ellipse circumference and fault plane edges. [Defaults: width = default, color = black, style = solid]. If the modifier **+cl** is appended then the color of the pen is updated from the CPT (see `cmap`). If instead modifier **+cf** is appended then the color from the cpt file is applied to symbol fill only [Default]. Use just **+c** to set both pen and fill color.
- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.
- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.
- **zvalue** (*str*) – [**m**|**e**|**n**|**u**][**+e**]. Select the quantity that will be used with the CPT given via `cmap` to set the fill color. Choose from **m**agnitude (vector magnitude or rotation magnitude), **e**ast-west velocity, **n**orth-south velocity, or **u**ser-supplied data column (supplied after the required columns). To instead use the corresponding error estimates (i.e., vector or rotation uncertainty) to lookup the color and paint the error ellipse or wedge instead, append **+e**.
- **panel** (*bool or int or list*) – [*row,col*|*index*]. Select a specific subplot panel. Only allowed when in subplot mode. Use `panel=True` to advance to the next panel in the selected order. Instead of *row,col* you may also give a scalar value *index* which depends on the order you set via `autolabel` when the subplot was defined. **Note**: *row*, *col*, and *index* all start at 0.
- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `d="-9999"` will replace all values equal to -9999 with NaN during input and all NaN values with -9999 during output. Prepend **i** to the *nodata* value for input columns only. Prepend **o** to the *nodata* value for output columns only.
- **find** (*str*) – [**~**]*"pattern"* | [**~**]/*regexp*/[**i**]. Only pass records that match the given *pattern* or regular expressions [Default processes all records]. Prepend **~** to the *pattern* or *regexp* to instead only pass data expressions that do not match the pattern. Append **i** for case insensitive matching. This does not apply to headers or segment headers.
- **header** (*str*) – [**i**|**o**][*n*][**+c**][**+d**][**+m***segheader*][**+r***remark*][**+t***title*]. Specify that input and/or output file(s) have *n* header records [Default is 0]. Prepend **i** if only the primary input should have header records. Prepend **o** to control the writing of header records, with the following modifiers supported:
  - **+d** to remove existing header records.
  - **+c** to add a header comment with column names to the output [Default is no column names].
  - **+m** to add a segment header *segheader* to the output after the header block [Default is no segment header].
  - **+r** to add a *remark* comment to the output [Default is no comment]. The *remark* string may contain \n to indicate line-breaks.
  - **+t** to add a *title* comment to the output [Default is no title]. The *title* string may contain \n to indicate line-breaks.

  Blank lines and lines starting with # are always skipped.
- **incols** (*str or 1d array*) – Specify data columns for primary input in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default reads all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in input order (e.g., `incols=[1,0]` for the 2nd column followed by the 1st column).

  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `incols="0:2,4+l"` to input the first three columns followed by the log-transformed 5th column). To read from a given column until the end of the record, leave off *stop* when specifying the column range. To read trailing text, add the column **t**. Append the word number to **t** to ingest only a single word from the trailing text. Instead of specifying columns, use `incols="n"` to simply read numerical input and skip trailing text. Optionally, append one of the following modifiers to any column or column range to transform the input columns:
    - **+l** to take the *log10* of the input values.
    - **+d** to divide the input values by the factor *divisor* [Default is 1].
    - **+s** to multiple the input values by the factor *scale* [Default is 1].
    - **+o** to add the given *offset* to the input values [Default is 0].
- **perspective** (*list or str*) – [**x**|**y**|**z**]*azim*[/*elev*[/*zlevel*]][**+w***lon0*/*lat0*[/*z0*]][**+v***x0*/*y0*]. Select perspective view and set the azimuth and elevation angle of the viewpoint. Default is [180, 90]. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#perspective-full.
- **transparency** (*int or float*) – Set transparency level, in [0-100] percent range. Default is 0, i.e., opaque. Only visible when PDF or raster format output is selected. Only the PNG format selection adds a transparency layer in the image (for further processing).

Examples using `pygmt.Figure.velo`

- *Velocity arrows and confidence ellipses*

pygmt.Figure.wiggle

`Figure.wiggle`(*data=None*, *x=None*, *y=None*, *z=None*, ***, *frame=None*, *position=None*, *color=None*, *projection=None*, *region=None*, *track=None*, *timestamp=None*, *verbose=None*, *pen=None*, *xshift=None*, *yshift=None*, *scale=None*, *binary=None*, *panel=None*, *nodata=None*, *find=None*, *coltypes=None*, *gap=None*, *header=None*, *incols=None*, *perspective=None*, *transparency=None*, *wrap=None*, ***kwargs*)

Plot z=f(x,y) anomalies along tracks.

Takes a matrix, (x,y,z) triplets, or a file name as input and plots z as a function of distance along track.

Must provide either `data` or `x`/`y`/`z`.

Full parameter list at https://docs.generic-mapping-tools.org/latest/wiggle.html

Aliases:

- B = frame
- D = position
- G = color
- J = projection
- R = region
- T = track
- U = timestamp

- V = verbose
- W = pen
- X = xshift
- Y = yshift
- Z = scale
- b = binary
- c = panel
- d = nodata
- e = find
- f = coltypes
- g = gap
- h = header
- i = incols
- p = perspective
- t = transparency
- w = wrap

Parameters

- **x/y/z** (*1d arrays*) – The arrays of x and y coordinates and z data points.
- **data** (*str or numpy.ndarray or pandas.DataFrame or xarray.Dataset or geopandas.GeoDataFrame*) – Pass in either a file name to an ASCII data table, a 2D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data. Use parameter `incols` to choose which columns are x, y, z, respectively.
- **projection** (*str*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **scale** (*str or float*) – Gives anomaly scale in data-units/distance-unit. Append **c**, **i**, or **p** to indicate the distance unit (cm, inch, or point); if no unit is given we use the default unit that is controlled by PROJ_LENGTH_UNIT.
- **frame** (*bool or str or list*) – Set map boundary *frame and axes attributes*.
- **position** (*str*) – [**g**|**j**|**J**|**n**|**x**]*refpoint***+w***length*[**+j***justify*][**+al**|**r**][**+o***dx*[/*dy*]][**+l**[*label*]]. Defines the reference point on the map for the vertical scale bar.
- **color** (*str*) – Set fill shade, color or pattern for positive and/or negative wiggles [Default is no fill]. Optionally, append **+p** to fill positive areas (this is the default behavior). Append **+n** to fill negative areas. Append **+n+p** to fill both positive and negative areas with the same fill. Note: You will need to repeat the color parameter to select different fills for the positive and negative wiggles.
- **track** (*str*) – Draw track [Default is no track]. Append pen attributes to use [Default is **0.25p,black,solid**].
- **timestamp** (*bool or str*) – Draw GMT time stamp logo on plot.

- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **pen** (*str*) – Specify outline pen attributes [Default is no outline].
- **xshift** (*str*) – [**a**|**c**|**f**|**r**][*xshift*]. Shift plot origin in x-direction.
- **yshift** (*str*) – [**a**|**c**|**f**|**r**][*yshift*]. Shift plot origin in y-direction. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#xy-full.
- **binary** (*bool or str*) – **i**|**o**[*ncols*][*type*][**w**][**+l**|**b**]. Select native binary input (using `binary="i"`) or output (using `binary="o"`), where *ncols* is the number of data columns of *type*, which must be one of:
  - **c** - int8_t (1-byte signed char)
  - **u** - uint8_t (1-byte unsigned char)
  - **h** - int16_t (2-byte signed int)
  - **H** - uint16_t (2-byte unsigned int)
  - **i** - int32_t (4-byte signed int)
  - **I** - uint32_t (4-byte unsigned int)
  - **l** - int64_t (8-byte signed int)
  - **L** - uint64_t (8-byte unsigned int)
  - **f** - 4-byte single-precision float
  - **d** - 8-byte double-precision float
  - **x** - use to skip *ncols* anywhere in the record

  For records with mixed types, append additional comma-separated combinations of *ncols type* (no space). The following modifiers are supported:
  - **w** after any item to force byte-swapping.
  - **+l**|**b** to indicate that the entire data file should be read as little- or big-endian, respectively.

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#bi-full.
- **panel** (*bool or int or list*) – [*row,col*|*index*]. Select a specific subplot panel. Only allowed when in subplot mode. Use `panel=True` to advance to the next panel in the selected order. Instead of *row,col* you may also give a scalar value *index* which depends on the order you set via `autolabel` when the subplot was defined. **Note**: *row*, *col*, and *index* all start at 0.
- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `d="-9999"` will replace all values equal to -9999 with NaN during input and all

NaN values with -9999 during output. Prepend **i** to the *nodata* value for input columns only. Prepend **o** to the *nodata* value for output columns only.

- **find** (*str*) – [~]*"pattern"* | [~]/*regexp*/[**i**]. Only pass records that match the given *pattern* or regular expressions [Default processes all records]. Prepend **~** to the *pattern* or *regexp* to instead only pass data expressions that do not match the pattern. Append **i** for case insensitive matching. This does not apply to headers or segment headers.
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **gap** (*str or list*) – [**a**]**x**|**y**|**d**|**X**|**Y**|**D**|[*col*]**z***gap*[**+n**|**p**]. Examine the spacing between consecutive data points in order to impose breaks in the line. To specify multiple criteria, provide a list with each item containing a string describing one set of criteria. Prepend **a** to specify that all the criteria must be met [Default is to impose breaks if any criteria are met]. The following modifiers are supported:
  - **x**|**X** - define a gap when there is a large enough change in the x coordinates (upper case to use projected coordinates).
  - **y**|**Y** - define a gap when there is a large enough change in the y coordinates (upper case to use projected coordinates).
  - **d**|**D** - define a gap when there is a large enough distance between coordinates (upper case to use projected coordinates).
  - [*col*]**z** - define a gap when there is a large enough change in the data in column *col* [default *col* is 2 (i.e., 3rd column)].

  A unit **u** may be appended to the specified *gap*:

  - For geographic data (**x**|**y**|**d**), the unit may be arc **d**(egree), **m**(inute), and **s**(econd), or (m)**e**(ter), **f**(eet), **k**(ilometer), **M**(iles), or **n**(autical miles) [Default is (m)**e**(ter)].
  - For projected data (**X**|**Y**|**D**), the unit may be **i**(nch), **c**(entimeter), or **p**(oint).

  One of the following modifiers can be appended to *gap* [Default imposes breaks based on the absolute value of the difference between the current and previous value]:

  - **+n** - specify that the previous value minus the current column value must exceed *gap* for a break to be imposed.
  - **+p** - specify that the current value minus the previous value must exceed *gap* for a break to be imposed.
- **header** (*str*) – [**i**|**o**][*n*][**+c**][**+d**][**+m***segheader*][**+r***remark*][**+t***title*]. Specify that input and/or output file(s) have *n* header records [Default is 0]. Prepend **i** if only the primary input should have header records. Prepend **o** to control the writing of header records, with the following modifiers supported:
  - **+d** to remove existing header records.
  - **+c** to add a header comment with column names to the output [Default is no column names].
  - **+m** to add a segment header *segheader* to the output after the header block [Default is no segment header].
  - **+r** to add a *remark* comment to the output [Default is no comment]. The *remark* string may contain \n to indicate line-breaks.

  - **+t** to add a *title* comment to the output [Default is no title]. The *title* string may contain \n to indicate line-breaks.

  Blank lines and lines starting with # are always skipped.

- **incols** (*str or 1d array*) – Specify data columns for primary input in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default reads all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in input order (e.g., `incols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `incols="0:2,4+l"` to input the first three columns followed by the log-transformed 5th column). To read from a given column until the end of the record, leave off *stop* when specifying the column range. To read trailing text, add the column **t**. Append the word number to **t** to ingest only a single word from the trailing text. Instead of specifying columns, use `incols="n"` to simply read numerical input and skip trailing text. Optionally, append one of the following modifiers to any column or column range to transform the input columns:
    - **+l** to take the *log10* of the input values.
    - **+d** to divide the input values by the factor *divisor* [Default is 1].
    - **+s** to multiple the input values by the factor *scale* [Default is 1].
    - **+o** to add the given *offset* to the input values [Default is 0].
- **perspective** (*list or str*) – [**x**|**y**|**z**]*azim*[/*elev*[/*zlevel*]][**+w***lon0*/*lat0*[/*z0*]][**+v***x0*/*y0*]. Select perspective view and set the azimuth and elevation angle of the viewpoint. Default is [180, 90]. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#perspective-full.
- **transparency** (*int or float*) – Set transparency level, in [0-100] percent range. Default is 0, i.e., opaque. Only visible when PDF or raster format output is selected. Only the PNG format selection adds a transparency layer in the image (for further processing).
- **wrap** (*str*) – **y**|**a**|**w**|**d**|**h**|**m**|**s**|**c***period*[/*phase*][**+c***col*]. Convert the input *x*-coordinate to a cyclical coordinate, or a different column if selected via **+c***col*. The following cyclical coordinate transformations are supported:
  - **y** - yearly cycle (normalized)
  - **a** - annual cycle (monthly)
  - **w** - weekly cycle (day)
  - **d** - daily cycle (hour)
  - **h** - hourly cycle (minute)
  - **m** - minute cycle (second)
  - **s** - second cycle (second)
  - **c** - custom cycle (normalized)

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#w-full.

Examples using `pygmt.Figure.wiggle`

- *Wiggle along tracks*

Color palette table generation:

grd2cpt(grid, *[, transparency, cmap, ...])	Make GMT color palette tables from a grid file.
makecpt(*[, transparency, cmap, background, ...])	Make GMT color palette tables.

pygmt.grd2cpt

`pygmt.grd2cpt`(*grid*, *, *transparency=None*, *cmap=None*, *background=None*, *color_model=None*, *nlevels=None*, *truncate=None*, *output=None*, *reverse=None*, *limit=None*, *overrule_bg=None*, *no_bg=None*, *log=None*, *region=None*, *series=None*, *verbose=None*, *categorical=None*, *cyclic=None*, *continuous=None*, ***kwargs*)

Make GMT color palette tables from a grid file.

This is a module that will help you make static color palette tables (CPTs). By default, the CPT will simply be saved to the current session, but you can use `output` to save it to a file. The CPT is based on an existing dynamic master CPT of your choice, and the mapping from data value to colors is through the data's cumulative distribution function (CDF), so that the colors are histogram equalized. Thus if the grid(s) and the resulting CPT are used in *pygmt.Figure.grdimage* with a linear projection, the colors will be uniformly distributed in area on the plot. Let z be the data values in the grid. Define CDF(Z) = (# of z < Z) / (# of z in grid). (NaNs are ignored). These z-values are then normalized to the master CPT and colors are sampled at the desired intervals.

The CPT includes three additional colors beyond the range of z-values. These are the background color (B) assigned to values lower than the lowest *z*-value, the foreground color (F) assigned to values higher than the highest *z*-value, and the NaN color (N) painted wherever values are undefined. For color tables beyond the standard GMT offerings, visit cpt-city and Scientific Colour-Maps.

If the master CPT includes B, F, and N entries, these will be copied into the new master file. If not, the parameters COLOR_BACKGROUND, COLOR_FOREGROUND, and COLOR_NAN from the gmt.conf file or the command line will be used. This default behavior can be overruled using the options `background`, `overrule_bg` or `no_bg`.

The color model (RGB, HSV or CMYK) of the palette created by *pygmt.grd2cpt* will be the same as specified in the header of the master CPT. When there is no COLOR_MODEL entry in the master CPT, the COLOR_MODEL specified in the gmt.conf file or the `color_model` option will be used.

Full option list at https://docs.generic-mapping-tools.org/latest/grd2cpt.html

Aliases:

- A = transparency
- C = cmap
- D = background
- E = nlevels
- F = color_model
- G = truncate
- H = output
- I = reverse
- L = limit

- M = overrule_bg
- N = no_bg
- Q = log
- R = region
- T = series
- V = verbose
- W = categorical
- Ww = cyclic
- Z = continuous

Parameters

- **grid** (*str or xarray.DataArray*) – The file name of the input grid or the grid loaded as a DataArray.
- **transparency** (*int or float or str*) – Sets a constant level of transparency (0-100) for all color slices. Append **+a** to also affect the fore-, back-, and nan-colors [Default is no transparency, i.e., 0 (opaque)].
- **cmap** (*str*) – Selects the master color palette table (CPT) to use in the interpolation. Full list of built-in color palette tables can be found at https://docs.generic-mapping-tools.org/latest/cookbook/cpts.html#built-in-color-palette-tables-cpt.
- **background** (*bool or str*) – Select the back- and foreground colors to match the colors for lowest and highest *z*-values in the output CPT [Default (`background=True` or `background='o'`) uses the colors specified in the master file, or those defined by the parameters COLOR_BACKGROUND, COLOR_FOREGROUND, and COLOR_NAN]. Use `background='i'` to match the colors for the lowest and highest values in the input (instead of the output) CPT.
- **color_model** – [**R**|**r**|**h**|**c**][**+c**[*label*]]. Force output CPT to be written with r/g/b codes, gray-scale values or color name (**R**, default) or r/g/b codes only (**r**), or h-s-v codes (**h**), or c/m/y/k codes (**c**). Optionally or alternatively, append **+c** to write discrete palettes in categorical format. If *label* is appended then we create labels for each category to be used when the CPT is plotted. The *label* may be a comma-separated list of category names (you can skip a category by not giving a name), or give *start*[-], where we automatically build monotonically increasing labels from *start* (a single letter or an integer). Append `-` to build ranges *start-start+1* instead.
- **nlevels** (*bool or int or str*) – Set to `True` to create a linear color table by using the grid z-range as the new limits in the CPT. Alternatively, set *nlevels* to resample the color table into *nlevels* equidistant slices.
- **series** (*list or str*) – [*min/max/inc*[**+b**|**l**|**n**]|*file*|*list*]. Defines the range of the new CPT by giving the lowest and highest z-value (and optionally an interval). If this is not given, the existing range in the master CPT will be used intact. The values produced defines the color slice boundaries. If **+n** is used it refers to the number of such boundaries and not the number of slices. For details on array creation, see https://docs.generic-mapping-tools.org/latest/makecpt.html#generate-1d-array.
- **truncate** (*list or str*) – *zlo/zhi*. Truncate the incoming CPT so that the lowest and highest z-levels are to *zlo* and *zhi*. If one of these equal NaN then we leave that end of the CPT alone. The truncation takes place before any resampling. See also https://docs.generic-mapping-tools.org/latest/cookbook/features.html#manipulating-cpts.

- **output** (*str*) – Optional parameter to set the file name with extension .cpt to store the generated CPT file. If not given or False (default), saves the CPT as the session current CPT.
- **reverse** (*str*) – Set this to True or c [Default] to reverse the sense of color progression in the master CPT. Set this to z to reverse the sign of z-values in the color table. Note that this change of z-direction happens before *truncate* and *series* values are used so the latter must be compatible with the changed *z*-range. See also https://docs.generic-mapping-tools.org/latest/cookbook/features.html#manipulating-cpts.
- **overrule_bg** (*str*) – Overrule background, foreground, and NaN colors specified in the master CPT with the values of the parameters COLOR_BACKGROUND, COLOR_FOREGROUND, and COLOR_NAN specified in the gmt.conf file or on the command line. When combined with `background`, only COLOR_NAN is considered.
- **no_bg** (*bool*) – Do not write out the background, foreground, and NaN-color fields [Default will write them, i.e. `no_bg=False`].
- **log** (*bool*) – For logarithmic interpolation scheme with input given as logarithms. Expects input z-values provided via `series` to be log10(*z*), assigns colors, and writes out *z*.
- **continuous** (*bool*) – Force a continuous CPT when building from a list of colors and a list of z-values [Default is None, i.e. discrete values].
- **categorical** (*bool*) – Do not interpolate the input color table but pick the output colors starting at the beginning of the color table, until colors for all intervals are assigned. This is particularly useful in combination with a categorical color table, like `cmap='categorical'`.
- **cyclic** (*bool*) – Produce a wrapped (cyclic) color table that endlessly repeats its range. Note that `cyclic=True` cannot be set together with `categorical=True`.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages

pygmt.makecpt

`pygmt.makecpt`(*, *transparency=None*, *cmap=None*, *background=None*, *color_model=None*, *truncate=None*, *output=None*, *reverse=None*, *overrule_bg=None*, *no_bg=None*, *log=None*, *series=None*, *verbose=None*, *categorical=None*, *cyclic=None*, *continuous=None*, ***kwargs*)

Make GMT color palette tables.

This is a module that will help you make static color palette tables (CPTs). By default, the CPT will simply be saved to the current session, but you can use `output` to save it to a file. You define an equidistant set of contour intervals or pass your own z-table or list, and create a new CPT based on an existing master (dynamic) CPT. The resulting CPT can be reversed relative to the master cpt, and can be made continuous or discrete. For color tables beyond the standard GMT offerings, visit cpt-city and Scientific Colour-Maps.

The CPT includes three additional colors beyond the range of z-values. These are the background color (B) assigned to values lower than the lowest *z*-value, the foreground color (F) assigned to values higher than the highest *z*-value, and the NaN color (N) painted wherever values are undefined.

If the master CPT includes B, F, and N entries, these will be copied into the new master file. If not, the parameters COLOR_BACKGROUND, COLOR_FOREGROUND, and COLOR_NAN from the gmt.conf file or the command line will be used. This default behavior can be overruled using the parameters `background`, `overrule_bg` or `no_bg`.

The color model (RGB, HSV or CMYK) of the palette created by **makecpt** will be the same as specified in the header of the master CPT. When there is no COLOR_MODEL entry in the master CPT, the COLOR_MODEL specified in the gmt.conf file or on the command line will be used.

Full option list at https://docs.generic-mapping-tools.org/latest/makecpt.html

Aliases:

- A = transparency
- C = cmap
- D = background
- F = color_model
- G = truncate
- H = output
- I = reverse
- M = overrule_bg
- N = no_bg
- Q = log
- T = series
- V = verbose
- W = categorical
- Ww = cyclic
- Z = continuous

Parameters

- **`transparency`** (*`str`*) – Sets a constant level of transparency (0-100) for all color slices. Append **+a** to also affect the fore-, back-, and nan-colors [Default is no transparency, i.e., 0 (opaque)].
- **`cmap`** (*`str`*) – Selects the master color palette table (CPT) to use in the interpolation. Full list of built-in color palette tables can be found at https://docs.generic-mapping-tools.org/latest/cookbook/cpts.html#built-in-color-palette-tables-cpt.
- **`background`** (*`bool or str`*) – Select the back- and foreground colors to match the colors for lowest and highest *z*-values in the output CPT [Default (`background=True` or `background='o'`) uses the colors specified in the master file, or those defined by the parameters COLOR_BACKGROUND, COLOR_FOREGROUND, and COLOR_NAN]. Use `background='i'` to match the colors for the lowest and highest values in the input (instead of the output) CPT.

- **color_model** – [**R**|**r**|**h**|**c**][**+c**[*label*]]. Force output CPT to be written with r/g/b codes, gray-scale values or color name (**R**, default) or r/g/b codes only (**r**), or h-s-v codes (**h**), or c/m/y/k codes (**c**). Optionally or alternatively, append **+c** to write discrete palettes in categorical format. If *label* is appended then we create labels for each category to be used when the CPT is plotted. The *label* may be a comma-separated list of category names (you can skip a category by not giving a name), or give *start*[-], where we automatically build monotonically increasing labels from *start* (a single letter or an integer). Append - to build ranges *start-start+1* instead.
- **series** (*list or str*) – [*min/max/inc*[**+b**|**l**|**n**]|*file*|*list*]. Defines the range of the new CPT by giving the lowest and highest z-value (and optionally an interval). If this is not given, the existing range in the master CPT will be used intact. The values produced defines the color slice boundaries. If **+n** is used it refers to the number of such boundaries and not the number of slices. For details on array creation, see https://docs.generic-mapping-tools.org/latest/makecpt.html#generate-1d-array.
- **truncate** (*list or str*) – *zlow/zhigh*. Truncate the incoming CPT so that the lowest and highest z-levels are to *zlow* and *zhigh*. If one of these equal NaN then we leave that end of the CPT alone. The truncation takes place before any resampling. See also https://docs.generic-mapping-tools.org/latest/cookbook/features.html#manipulating-cpts.
- **output** (*str*) – Optional. The file name with extension .cpt to store the generated CPT file. If not given or False (default), saves the CPT as the session current CPT.
- **reverse** (*str*) – Set this to True or **c**[Default] to reverse the sense of color progression in the master CPT. Set this to z to reverse the sign of z-values in the color table. Note that this change of z-direction happens before `truncate` and `series` values are used so the latter must be compatible with the changed *z*-range. See also https://docs.generic-mapping-tools.org/latest/cookbook/features.html#manipulating-cpts.
- **overrule_bg** (*str*) – Overrule background, foreground, and NaN colors specified in the master CPT with the values of the parameters COLOR_BACKGROUND, COLOR_FOREGROUND, and COLOR_NAN specified in the gmt.conf file or on the command line. When combined with **background**, only COLOR_NAN is considered.
- **no_bg** (*bool*) – Do not write out the background, foreground, and NaN-color fields [Default will write them, i.e. `no_bg=False`].
- **log** (*bool*) – For logarithmic interpolation scheme with input given as logarithms. Expects input z-values provided via **series** to be log10(*z*), assigns colors, and writes out *z*.
- **continuous** (*bool*) – Force a continuous CPT when building from a list of colors and a list of z-values [Default is None, i.e. discrete values].
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages

- **categorical** (*bool*) – Do not interpolate the input color table but pick the output colors starting at the beginning of the color table, until colors for all intervals are assigned. This is particularly useful in combination with a categorical color table, like `cmap='categorical'`.
- **cyclic** (*bool*) – Produce a wrapped (cyclic) color table that endlessly repeats its range. Note that `cyclic=True` cannot be set together with `categorical=True`.

Examples using `pygmt.makecpt`

- *Line colors with a custom CPT*
- *Color points by categories*
- *Calculating grid gradient and radiance*
- *Create 'wet-dry' mask grid*
- *3D Scatter plots*
- *Multiple colormaps*
- *Plotting data points*

Saving and displaying the figure:

Figure.savefig(fname[, transparent, crop, ...])	Save the figure to a file.
Figure.show([dpi, width, method])	Display a preview of the figure.
Figure.psconvert(*[, crop, gs_option, dpi, ...])	Convert [E]PS file(s) to other formats.

pygmt.Figure.savefig

Figure.**savefig**(*fname*, *transparent=False*, *crop=True*, *anti_alias=True*, *show=False*, ***kwargs*)

Save the figure to a file.

This method implements a matplotlib-like interface for *pygmt.Figure.psconvert*.

Supported formats: PNG (`.png`), JPEG (`.jpg`), PDF (`.pdf`), BMP (`.bmp`), TIFF (`.tif`), EPS (`.eps`), and KML (`.kml`). The KML output generates a companion PNG file.

You can pass in any keyword arguments that *pygmt.Figure.psconvert* accepts.

Parameters

- **fname** (*str*) – The desired figure file name, including the extension. See the list of supported formats and their extensions above.
- **transparent** (*bool*) – If True, will use a transparent background for the figure. Only valid for PNG format.
- **crop** (*bool*) – If True, will crop the figure canvas (page) to the plot area.
- **anti_alias** (*bool*) – If True, will use anti aliasing when creating raster images (PNG, JPG, TIFF). More specifically, it passes arguments `t2` and `g2` to the `anti_aliasing` parameter of *pygmt.Figure.psconvert*. Ignored if creating vector graphics.
- **show** (*bool*) – If True, will open the figure in an external viewer.
- **dpi** (*int*) – Set raster resolution in dpi. Default is 720 for PDF, 300 for others.

Examples using `pygmt.Figure.savefig`

- *Making your first figure*

pygmt.Figure.show

`Figure.show`(*dpi=300*, *width=500*, *method=None*)

Display a preview of the figure.

Inserts the preview in the Jupyter notebook output if available, otherwise opens it in the default viewer for your operating system (falls back to the default web browser).

`pygmt.set_display` can select the default display method (**notebook**, **external**, or **none**).

The `method` parameter can also override the default display method for the current figure. Parameters `dpi` and `width` can be used to control the resolution and dimension of the figure in the notebook.

Note: The external viewer can be disabled by setting the PYGMT_USE_EXTERNAL_DISPLAY environment variable to **false**. This is useful when running unit tests and building the documentation in consoles without a Graphical User Interface.

Note that the external viewer does not block the current process.

Parameters

- **dpi** (*int*) – The image resolution (dots per inch) in Jupyter notebooks.
- **width** (*int*) – The image width (in pixels) in Jupyter notebooks.
- **method** (*str*) – How the current figure will be displayed. Options are
  - **external**: PDF preview in an external program [default]
  - **notebook**: PNG preview [default in Jupyter notebooks]
  - **none**: Disable image preview

Examples using `pygmt.Figure.show`

- *Cartesian, circular, and geographic vectors*
- *Line colors with a custom CPT*
- *Line fronts*
- *Line styles*
- *Roads*
- *Vector heads and tails*
- *Wiggle along tracks*
- *Basic geometric symbols*
- *Color points by categories*
- *Custom symbols*
- *Datetime inputs*
- *Multi-parameter symbols*
- *Points*

- *Points with varying transparency*
- *Scatter plots with a legend*
- *Text symbols*
- *Calculating grid gradient and radiance*
- *Clipping grid values*
- *Contours*
- *Create 'wet-dry' mask grid*
- *Images on figures*
- *Sampling along tracks*
- *3D Scatter plots*
- *Plotting a surface*
- *Focal mechanisms*
- *Velocity arrows and confidence ellipses*
- *Double Y axes graph*
- *Histogram*
- *Rose diagram*
- *Colorbar*
- *Day-night terminator line and twilights*
- *Inset*
- *Inset map showing a rectangular region*
- *Legend*
- *Logo*
- *Multiple colormaps*
- *Adding an inset to the figure*
- *Coastlines and borders*
- *Configuring PyGMT defaults*
- *Creating a 3D perspective image*
- *Creating a map with contour lines*
- *Frames, ticks, titles, and labels*
- *Making subplots*
- *Making your first figure*
- *Plotting Earth relief*
- *Plotting data points*
- *Plotting datetime charts*
- *Plotting lines*
- *Plotting text*

- *Plotting vectors*
- *Setting the region*
- *Azimuthal Equidistant*
- *General Perspective*
- *General Stereographic*
- *Gnomonic*
- *Lambert Azimuthal Equal Area*
- *Orthographic*
- *Albers Conic Equal Area*
- *Equidistant conic*
- *Lambert Conic Conformal Projection*
- *Polyconic Projection*
- *Cassini Cylindrical*
- *Cylindrical Stereographic*
- *Cylindrical equal-area*
- *Cylindrical equidistant*
- *Mercator*
- *Miller cylindrical*
- *Oblique Mercator*
- *Oblique Mercator*
- *Oblique Mercator*
- *Transverse Mercator*
- *Universal Transverse Mercator*
- *Eckert IV*
- *Eckert VI*
- *Hammer*
- *Mollweide*
- *Robinson*
- *Sinusoidal*
- *Van der Grinten*
- *Winkel Tripel*
- *Cartesian linear*
- *Cartesian logarithmic*
- *Cartesian power*
- *Polar*

pygmt.Figure.psconvert

`Figure.`**`psconvert`**`(`**, crop=None, gs_option=None, dpi=None, prefix=None, icc_gray=None, fmt=None, anti_aliasing=None, verbose=None, **kwargs*`)`

Convert [E]PS file(s) to other formats.

Converts one or more PostScript files to other formats (BMP, EPS, JPEG, PDF, PNG, PPM, SVG, TIFF) using GhostScript.

If no input files are given, will convert the current active figure (see `pygmt.figure`). In this case, an output name must be given using parameter *prefix*.

Full option list at https://docs.generic-mapping-tools.org/latest/psconvert.html

Aliases:

- A = crop
- C = gs_option
- E = dpi
- F = prefix
- I = icc_gray
- Q = anti_aliasing
- T = fmt
- V = verbose

Parameters

- **`crop`** (*`str or bool`*) – Adjust the BoundingBox and HiResBoundingBox to the minimum required by the image content. Append `u` to first remove any GMT-produced time-stamps. Default is True.
- **`gs_option`** (*`str`*) – Specify a single, custom option that will be passed on to GhostScript as is.
- **`dpi`** (*`int`*) – Set raster resolution in dpi. Default = 720 for PDF, 300 for others.
- **`prefix`** (*`str`*) – Force the output file name. By default output names are constructed using the input names as base, which are appended with an appropriate extension. Use this option to provide a different name, but without extension. Extension is still determined automatically.
- **`icc_gray`** (*`bool`*) – Enforce gray-shades by using ICC profiles.
- **`anti_aliasing`** (*`str`*) – [**g**|**p**|**t**][**1**|**2**|**4**]. Set the anti-aliasing options for **g**raphics or **t**ext. Append the size of the subsample box (1, 2, or 4) [4]. [Default is no anti-aliasing (same as bits = 1)].
- **`fmt`** (*`str`*) – Sets the output format, where **b** means BMP, **e** means EPS, **E** means EPS with PageSize command, **f** means PDF, **F** means multi-page PDF, **j** means JPEG, **g** means PNG, **G** means transparent PNG (untouched regions are transparent), **m** means PPM, **s** means SVG, and **t** means TIFF [default is JPEG]. To **b**|**j**|**g**|**t**, optionally append **+m** in order to get a monochrome (grayscale) image. The EPS format can be combined with any of the other formats. For example, **ef** creates both an EPS and a PDF file. Using **F** creates a multi-page PDF file from the list of input PS or PDF files. It requires the `prefix` parameter.
- **`verbose`** (*`bool or str`*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:

- **q** - Quiet, not even fatal error messages are produced
- **e** - Error messages only
- **w** - Warnings [Default]
- **t** - Timings (report runtimes for time-intensive algorithms);
- **i** - Informational messages (same as `verbose=True`)
- **c** - Compatibility warnings
- **d** - Debugging messages

Configuring the display settings:

set_display([method])	Set the display method.

pygmt.set_display

`pygmt.set_display`(*method=None*)

Set the display method.

Parameters **method** (*str or None*) – The method to display an image. Choose from:

- **external**: PDF preview in an external program [default]
- **notebook**: PNG preview [default in Jupyter notebooks]
- **none**: Disable image preview

9.21.3 Data Processing

Operations on tabular data:

blockmean([data, x, y, z, outfile, spacing, ...])	Block average (x,y,z) data tables by mean estimation.
blockmedian([data, x, y, z, outfile, ...])	Block average (x,y,z) data tables by median estimation.
blockmode([data, x, y, z, outfile, spacing, ...])	Block average (x,y,z) data tables by mode estimation.
nearneighbor([data, x, y, z, empty, ...])	Grid table data using a "Nearest neighbor" algorithm
project([data, x, y, z, outfile, azimuth, ...])	Project data onto lines or great circles, or generate tracks.
select([data, outfile, area_thresh, ...])	Select data table subsets based on multiple spatial criteria.
sph2grd(data, *[, outgrid, spacing, region, ...])	Create spherical grid files in tension of data.
sphdistance([data, x, y, single_form, ...])	Create Voronoi distance, node, or natural nearest-neighbor grid on a sphere.
sphinterpolate(data, *[, outgrid, spacing, ...])	Create spherical grid files in tension of data.
surface([data, x, y, z, spacing, region, ...])	Grids table data using adjustable tension continuous curvature splines.
xyz2grd([data, x, y, z, duplicate, outgrid, ...])	Create a grid file from table data.

pygmt.blockmean

`pygmt.blockmean`(*data=None*, *x=None*, *y=None*, *z=None*, *outfile=None*, ***, *spacing=None*, *region=None*, *verbose=None*, *aspatial=None*, *binary=None*, *nodata=None*, *find=None*, *coltypes=None*, *header=None*, *incols=None*, *outcols=None*, *registration=None*, *wrap=None*, ***kwargs*)

Block average (x,y,z) data tables by mean estimation.

Reads arbitrarily located (x,y,z) triples [or optionally weighted quadruples (x,y,z,w)] and writes to the output a mean position and value for every non-empty block in a grid region defined by the `region` and `spacing` parameters.

Takes a matrix, xyz triplets, or a file name as input.

Must provide either `data` or `x`, `y`, and `z`.

Full option list at https://docs.generic-mapping-tools.org/latest/blockmean.html

Aliases:

- I = spacing
- R = region
- V = verbose
- a = aspatial
- b = binary
- d = nodata
- e = find
- f = coltypes
- h = header
- i = incols
- o = outcols
- r = registration
- w = wrap

Parameters

- **data** (*str or numpy.ndarray or pandas.DataFrame or xarray.Dataset or geopandas.GeoDataFrame*) – Pass in (x, y, z) or (longitude, latitude, elevation) values by providing a file name to an ASCII data table, a 2D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **x/y/z** (*1d arrays*) – Arrays of x and y coordinates and values z of the data points.
- **spacing** (*str*) – *xinc*[**+e**|**n**][/*yinc*[**+e**|**n**]]. *x_inc* [and optionally *y_inc*] is the grid spacing.
  - **Geographical (degrees) coordinates**: Optionally, append an increment unit. Choose among **m** to indicate arc minutes or **s** to indicate arc seconds. If one of the units **e**, **f**, **k**, **M**, **n** or **u** is appended instead, the increment is assumed to be given in meter, foot, km, mile, nautical mile or US survey foot, respectively, and will be converted to the equivalent degrees longitude at the middle latitude of the region (the conversion depends on PROJ_ELLIPSOID). If *y_inc* is given but set to 0 it will be reset equal to *x_inc*; otherwise it will be converted to degrees latitude.

  - **All coordinates**: If **+e** is appended then the corresponding max *x* (*east*) or *y* (*north*) may be slightly adjusted to fit exactly the given increment [by default the increment may be adjusted slightly to fit the given domain]. Finally, instead of giving an increment you may specify the *number of nodes* desired by appending **+n** to the supplied integer argument; the increment is then recalculated from the number of nodes, the *registration*, and the domain. The resulting increment value depends on whether you have selected a gridline-registered or pixel-registered grid; see GMT File Formats for details.

  Note: If `region=grdfile` is used then the grid spacing and the registration have already been initialized; use `spacing` and `registration` to override these values.

- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **outfile** (*str*) – The file name for the output ASCII file.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **aspatial** (*bool or str*) – [*col*=]*name*[,…]. Control how aspatial data are handled during input and output. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#aspatial-full.
- **binary** (*bool or str*) – **i**|**o**[*ncols*][*type*][**w**][**+l**|**b**]. Select native binary input (using `binary="i"`) or output (using `binary="o"`), where *ncols* is the number of data columns of *type*, which must be one of:
  - **c** - int8_t (1-byte signed char)
  - **u** - uint8_t (1-byte unsigned char)
  - **h** - int16_t (2-byte signed int)
  - **H** - uint16_t (2-byte unsigned int)
  - **i** - int32_t (4-byte signed int)
  - **I** - uint32_t (4-byte unsigned int)
  - **l** - int64_t (8-byte signed int)
  - **L** - uint64_t (8-byte unsigned int)
  - **f** - 4-byte single-precision float
  - **d** - 8-byte double-precision float
  - **x** - use to skip *ncols* anywhere in the record

  For records with mixed types, append additional comma-separated combinations of *ncols type* (no space). The following modifiers are supported:

  - **w** after any item to force byte-swapping.

  - **+l|b** to indicate that the entire data file should be read as little- or big-endian, respectively.

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#bi-full.

- **nodata** (*str*) – **i|o***nodata*. Substitute specific values with NaN (for tabular data). For example, `d="-9999"` will replace all values equal to -9999 with NaN during input and all NaN values with -9999 during output. Prepend **i** to the *nodata* value for input columns only. Prepend **o** to the *nodata* value for output columns only.
- **find** (*str*) – [**~**]*"pattern"* | [**~**]/*regexp*/[**i**]. Only pass records that match the given *pattern* or regular expressions [Default processes all records]. Prepend **~** to the *pattern* or *regexp* to instead only pass data expressions that do not match the pattern. Append **i** for case insensitive matching. This does not apply to headers or segment headers.
- **incols** (*str or 1d array*) – Specify data columns for primary input in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default reads all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in input order (e.g., `incols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `incols="0:2,4+l"` to input the first three columns followed by the log-transformed 5th column). To read from a given column until the end of the record, leave off *stop* when specifying the column range. To read trailing text, add the column **t**. Append the word number to **t** to ingest only a single word from the trailing text. Instead of specifying columns, use `incols="n"` to simply read numerical input and skip trailing text. Optionally, append one of the following modifiers to any column or column range to transform the input columns:
    - **+l** to take the *log10* of the input values.
    - **+d** to divide the input values by the factor *divisor* [Default is 1].
    - **+s** to multiple the input values by the factor *scale* [Default is 1].
    - **+o** to add the given *offset* to the input values [Default is 0].
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **header** (*str*) – [**i**|**o**][*n*][**+c**][**+d**][**+m***segheader*][**+r***remark*][**+t***title*]. Specify that input and/or output file(s) have *n* header records [Default is 0]. Prepend **i** if only the primary input should have header records. Prepend **o** to control the writing of header records, with the following modifiers supported:
  - **+d** to remove existing header records.
  - **+c** to add a header comment with column names to the output [Default is no column names].
  - **+m** to add a segment header *segheader* to the output after the header block [Default is no segment header].
  - **+r** to add a *remark* comment to the output [Default is no comment]. The *remark* string may contain \n to indicate line-breaks.
  - **+t** to add a *title* comment to the output [Default is no title]. The *title* string may contain \n to indicate line-breaks.

  Blank lines and lines starting with # are always skipped.

- **outcols** (*str or 1d array*) – *cols*[,. . .][,**t**[*word*]]. Specify data columns for primary output in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default writes all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in output order (e.g., `outcols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `outcols="0:2,4"` to output the first three columns followed by the 5th column). To write from a given column until the end of the record, leave off *stop* when specifying the column range. To write trailing text, add the column **t**. Append the word number to **t** to write only a single word from the trailing text. Instead of specifying columns, use `outcols="n"` to simply read numerical input and skip trailing text. Note: if `incols` is also used then the columns given to `outcols` correspond to the order after the `incols` selection has taken place.
- **registration** (*str*) – **g**|**p**. Force gridline (**g**) or pixel (**p**) node registration. [Default is **g**(ridline)].
- **wrap** (*str*) – **y**|**a**|**w**|**d**|**h**|**m**|**s**|**c***period*[/*phase*][+**c***col*]. Convert the input *x*-coordinate to a cyclical coordinate, or a different column if selected via **+c***col*. The following cyclical coordinate transformations are supported:
  - **y** - yearly cycle (normalized)
  - **a** - annual cycle (monthly)
  - **w** - weekly cycle (day)
  - **d** - daily cycle (hour)
  - **h** - hourly cycle (minute)
  - **m** - minute cycle (second)
  - **s** - second cycle (second)
  - **c** - custom cycle (normalized)

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#w-full.

Returns

output (*pandas.DataFrame or None*) – Return type depends on whether the `outfile` parameter is set:

- `pandas.DataFrame` table with (x, y, z) columns if `outfile` is not set.
- None if `outfile` is set (filtered output will be stored in file set by `outfile`).

pygmt.blockmedian

`pygmt.blockmedian`(*data=None*, *x=None*, *y=None*, *z=None*, *outfile=None*, ***, *spacing=None*, *region=None*, *verbose=None*, *aspatial=None*, *binary=None*, *nodata=None*, *find=None*, *coltypes=None*, *header=None*, *incols=None*, *outcols=None*, *registration=None*, *wrap=None*, ***kwargs*)

Block average (x,y,z) data tables by median estimation.

Reads arbitrarily located (x,y,z) triples [or optionally weighted quadruples (x,y,z,w)] and writes to the output a median position and value for every non-empty block in a grid region defined by the `region` and `spacing` parameters.

Takes a matrix, xyz triplets, or a file name as input.

Must provide either `data` or `x`, `y`, and `z`.

Full option list at https://docs.generic-mapping-tools.org/latest/blockmedian.html

Aliases:

- I = spacing
- R = region
- V = verbose
- a = aspatial
- b = binary
- d = nodata
- e = find
- f = coltypes
- h = header
- i = incols
- o = outcols
- r = registration
- w = wrap

Parameters

- **data** (*str or numpy.ndarray or pandas.DataFrame or xarray.Dataset or geopandas.GeoDataFrame*) – Pass in (x, y, z) or (longitude, latitude, elevation) values by providing a file name to an ASCII data table, a 2D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **x/y/z** (*1d arrays*) – Arrays of x and y coordinates and values z of the data points.
- **spacing** (*str*) – *xinc*[**+e**|**n**][/*yinc*[**+e**|**n**]]. *x_inc* [and optionally *y_inc*] is the grid spacing.
  - **Geographical (degrees) coordinates**: Optionally, append an increment unit. Choose among **m** to indicate arc minutes or **s** to indicate arc seconds. If one of the units **e**, **f**, **k**, **M**, **n** or **u** is appended instead, the increment is assumed to be given in meter, foot, km, mile, nautical mile or US survey foot, respectively, and will be converted to the equivalent degrees longitude at the middle latitude of the region (the conversion depends on PROJ_ELLIPSOID). If *y_inc* is given but set to 0 it will be reset equal to *x_inc*; otherwise it will be converted to degrees latitude.
  - **All coordinates**: If **+e** is appended then the corresponding max *x* (*east*) or *y* (*north*) may be slightly adjusted to fit exactly the given increment [by default the increment may be adjusted slightly to fit the given domain]. Finally, instead of giving an increment you may specify the *number of nodes* desired by appending **+n** to the supplied integer argument; the increment is then recalculated from the number of nodes, the *registration*, and the domain. The resulting increment value depends on whether you have selected a gridline-registered or pixel-registered grid; see GMT File Formats for details.

  Note: If `region=grdfile` is used then the grid spacing and the registration have already been initialized; use `spacing` and `registration` to override these values.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.

- **outfile** (*str*) – The file name for the output ASCII file.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **aspatial** (*bool or str*) – [*col*=]*name*[,. . .]. Control how aspatial data are handled during input and output. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#aspatial-full.
- **binary** (*bool or str*) – **i**|**o**[*ncols*][*type*][**w**][**+l**|**b**]. Select native binary input (using `binary="i"`) or output (using `binary="o"`), where *ncols* is the number of data columns of *type*, which must be one of:
  - **c** - int8_t (1-byte signed char)
  - **u** - uint8_t (1-byte unsigned char)
  - **h** - int16_t (2-byte signed int)
  - **H** - uint16_t (2-byte unsigned int)
  - **i** - int32_t (4-byte signed int)
  - **I** - uint32_t (4-byte unsigned int)
  - **l** - int64_t (8-byte signed int)
  - **L** - uint64_t (8-byte unsigned int)
  - **f** - 4-byte single-precision float
  - **d** - 8-byte double-precision float
  - **x** - use to skip *ncols* anywhere in the record

  For records with mixed types, append additional comma-separated combinations of *ncols type* (no space). The following modifiers are supported:
  - **w** after any item to force byte-swapping.
  - **+l**|**b** to indicate that the entire data file should be read as little- or big-endian, respectively.

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#bi-full.
- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `d="-9999"` will replace all values equal to -9999 with NaN during input and all NaN values with -9999 during output. Prepend **i** to the *nodata* value for input columns only. Prepend **o** to the *nodata* value for output columns only.
- **find** (*str*) – [**~**]*"pattern"* | [**~**]/*regexp*/[**i**]. Only pass records that match the given *pattern* or regular expressions [Default processes all records]. Prepend **~** to the *pattern* or *regexp* to instead only pass data expressions that do not match the pattern. Append **i** for case insensitive matching. This does not apply to headers or segment headers.

- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **header** (*str*) – [**i**|**o**][*n*][**+c**][**+d**][**+m***segheader*][**+r***remark*][**+t***title*]. Specify that input and/or output file(s) have *n* header records [Default is 0]. Prepend **i** if only the primary input should have header records. Prepend **o** to control the writing of header records, with the following modifiers supported:
  - **+d** to remove existing header records.
  - **+c** to add a header comment with column names to the output [Default is no column names].
  - **+m** to add a segment header *segheader* to the output after the header block [Default is no segment header].
  - **+r** to add a *remark* comment to the output [Default is no comment]. The *remark* string may contain \n to indicate line-breaks.
  - **+t** to add a *title* comment to the output [Default is no title]. The *title* string may contain \n to indicate line-breaks.

  Blank lines and lines starting with # are always skipped.
- **incols** (*str or 1d array*) – Specify data columns for primary input in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default reads all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in input order (e.g., `incols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `incols="0:2,4+l"` to input the first three columns followed by the log-transformed 5th column). To read from a given column until the end of the record, leave off *stop* when specifying the column range. To read trailing text, add the column **t**. Append the word number to **t** to ingest only a single word from the trailing text. Instead of specifying columns, use `incols="n"` to simply read numerical input and skip trailing text. Optionally, append one of the following modifiers to any column or column range to transform the input columns:
    - **+l** to take the *log10* of the input values.
    - **+d** to divide the input values by the factor *divisor* [Default is 1].
    - **+s** to multiple the input values by the factor *scale* [Default is 1].
    - **+o** to add the given *offset* to the input values [Default is 0].
- **outcols** (*str or 1d array*) – *cols*[,...][,**t**[*word*]]. Specify data columns for primary output in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default writes all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in output order (e.g., `outcols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `outcols="0:2,4"` to output the first three columns followed by the 5th column). To write from a given column until the end of the record, leave off *stop* when specifying the column range. To write trailing text, add the column **t**. Append the word number to

t to write only a single word from the trailing text. Instead of specifying columns, use `outcols="n"` to simply read numerical input and skip trailing text. Note: if `incols` is also used then the columns given to `outcols` correspond to the order after the `incols` selection has taken place.

- **registration** (*str*) – **g**|**p**. Force gridline (**g**) or pixel (**p**) node registration. [Default is **g**(ridline)].
- **wrap** (*str*) – **y**|**a**|**w**|**d**|**h**|**m**|**s**|**c***period*[/*phase*][**+c***col*]. Convert the input *x*-coordinate to a cyclical coordinate, or a different column if selected via **+c***col*. The following cyclical coordinate transformations are supported:
  - **y** - yearly cycle (normalized)
  - **a** - annual cycle (monthly)
  - **w** - weekly cycle (day)
  - **d** - daily cycle (hour)
  - **h** - hourly cycle (minute)
  - **m** - minute cycle (second)
  - **s** - second cycle (second)
  - **c** - custom cycle (normalized)

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#w-full.

Returns

output (*pandas.DataFrame or None*) – Return type depends on whether the `outfile` parameter is set:

- `pandas.DataFrame` table with (x, y, z) columns if `outfile` is not set.
- None if `outfile` is set (filtered output will be stored in file set by `outfile`).

pygmt.blockmode

`pygmt.blockmode`(*data=None*, *x=None*, *y=None*, *z=None*, *outfile=None*, ***, *spacing=None*, *region=None*, *verbose=None*, *aspatial=None*, *binary=None*, *nodata=None*, *find=None*, *coltypes=None*, *header=None*, *incols=None*, *outcols=None*, *registration=None*, *wrap=None*, ***kwargs*)

Block average (x,y,z) data tables by mode estimation.

Reads arbitrarily located (x,y,z) triples [or optionally weighted quadruples (x,y,z,w)] and writes to the output a mode position and value for every non-empty block in a grid region defined by the `region` and `spacing` parameters.

Takes a matrix, xyz triplets, or a file name as input.

Must provide either `data` or `x`, `y`, and `z`.

Full option list at https://docs.generic-mapping-tools.org/latest/blockmode.html

Aliases:

- I = spacing
- R = region
- V = verbose
- a = aspatial

- b = binary
- d = nodata
- e = find
- f = coltypes
- h = header
- i = incols
- o = outcols
- r = registration
- w = wrap

Parameters

- **`data`** (*`str` or `numpy.ndarray` or `pandas.DataFrame` or `xarray.Dataset` or `geopandas.GeoDataFrame`*) – Pass in (x, y, z) or (longitude, latitude, elevation) values by providing a file name to an ASCII data table, a 2D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **`x/y/z`** (*1d arrays*) – Arrays of x and y coordinates and values z of the data points.
- **`spacing`** (*`str`*) – *xinc*[**+e**|**n**][/*yinc*[**+e**|**n**]]. *x_inc* [and optionally *y_inc*] is the grid spacing.
  - **Geographical (degrees) coordinates**: Optionally, append an increment unit. Choose among **m** to indicate arc minutes or **s** to indicate arc seconds. If one of the units **e**, **f**, **k**, **M**, **n** or **u** is appended instead, the increment is assumed to be given in meter, foot, km, mile, nautical mile or US survey foot, respectively, and will be converted to the equivalent degrees longitude at the middle latitude of the region (the conversion depends on PROJ_ELLIPSOID). If *y_inc* is given but set to 0 it will be reset equal to *x_inc*; otherwise it will be converted to degrees latitude.
  - **All coordinates**: If **+e** is appended then the corresponding max *x* (*east*) or *y* (*north*) may be slightly adjusted to fit exactly the given increment [by default the increment may be adjusted slightly to fit the given domain]. Finally, instead of giving an increment you may specify the *number of nodes* desired by appending **+n** to the supplied integer argument; the increment is then recalculated from the number of nodes, the *registration*, and the domain. The resulting increment value depends on whether you have selected a gridline-registered or pixel-registered grid; see GMT File Formats for details.

  Note: If `region=grdfile` is used then the grid spacing and the registration have already been initialized; use `spacing` and `registration` to override these values.
- **`region`** (*`str` or `list`*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **`outfile`** (*`str`*) – The file name for the output ASCII file.
- **`verbose`** (*`bool` or `str`*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);

  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages

- **aspatial** (*bool or str*) – [*col*=]*name*[,. . .]. Control how aspatial data are handled during input and output. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#aspatial-full.
- **binary** (*bool or str*) – **i**|**o**[*ncols*][*type*][**w**][**+l**|**b**]. Select native binary input (using `binary="i"`) or output (using `binary="o"`), where *ncols* is the number of data columns of *type*, which must be one of:
  - **c** - int8_t (1-byte signed char)
  - **u** - uint8_t (1-byte unsigned char)
  - **h** - int16_t (2-byte signed int)
  - **H** - uint16_t (2-byte unsigned int)
  - **i** - int32_t (4-byte signed int)
  - **I** - uint32_t (4-byte unsigned int)
  - **l** - int64_t (8-byte signed int)
  - **L** - uint64_t (8-byte unsigned int)
  - **f** - 4-byte single-precision float
  - **d** - 8-byte double-precision float
  - **x** - use to skip *ncols* anywhere in the record

  For records with mixed types, append additional comma-separated combinations of *ncols type* (no space). The following modifiers are supported:
  - **w** after any item to force byte-swapping.
  - **+l**|**b** to indicate that the entire data file should be read as little- or big-endian, respectively.

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#bi-full.
- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `d="-9999"` will replace all values equal to -9999 with NaN during input and all NaN values with -9999 during output. Prepend **i** to the *nodata* value for input columns only. Prepend **o** to the *nodata* value for output columns only.
- **find** (*str*) – [**~**]*"pattern"* | [**~**]/*regexp*/[**i**]. Only pass records that match the given *pattern* or regular expressions [Default processes all records]. Prepend **~** to the *pattern* or *regexp* to instead only pass data expressions that do not match the pattern. Append **i** for case insensitive matching. This does not apply to headers or segment headers.
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **header** (*str*) – [**i**|**o**][*n*][**+c**][**+d**][**+m***segheader*][**+r***remark*][**+t***title*]. Specify that input and/or output file(s) have *n* header records [Default is 0]. Prepend **i** if only the primary input should have header records. Prepend **o** to control the writing of header records, with the following modifiers supported:
  - **+d** to remove existing header records.

  - **+c** to add a header comment with column names to the output [Default is no column names].
  - **+m** to add a segment header *segheader* to the output after the header block [Default is no segment header].
  - **+r** to add a *remark* comment to the output [Default is no comment]. The *remark* string may contain \n to indicate line-breaks.
  - **+t** to add a *title* comment to the output [Default is no title]. The *title* string may contain \n to indicate line-breaks.

  Blank lines and lines starting with # are always skipped.

- **incols** (*str or 1d array*) – Specify data columns for primary input in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default reads all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in input order (e.g., `incols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `incols="0:2,4+l"` to input the first three columns followed by the log-transformed 5th column). To read from a given column until the end of the record, leave off *stop* when specifying the column range. To read trailing text, add the column **t**. Append the word number to **t** to ingest only a single word from the trailing text. Instead of specifying columns, use `incols="n"` to simply read numerical input and skip trailing text. Optionally, append one of the following modifiers to any column or column range to transform the input columns:
    - **+l** to take the *log10* of the input values.
    - **+d** to divide the input values by the factor *divisor* [Default is 1].
    - **+s** to multiple the input values by the factor *scale* [Default is 1].
    - **+o** to add the given *offset* to the input values [Default is 0].
- **outcols** (*str or 1d array*) – *cols*[,...][,**t**[*word*]]. Specify data columns for primary output in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default writes all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in output order (e.g., `outcols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `outcols="0:2,4"` to output the first three columns followed by the 5th column). To write from a given column until the end of the record, leave off *stop* when specifying the column range. To write trailing text, add the column **t**. Append the word number to **t** to write only a single word from the trailing text. Instead of specifying columns, use `outcols="n"` to simply read numerical input and skip trailing text. Note: if `incols` is also used then the columns given to `outcols` correspond to the order after the `incols` selection has taken place.
- **registration** (*str*) – **g**|**p**. Force gridline (**g**) or pixel (**p**) node registration. [Default is **g**(ridline)].
- **wrap** (*str*) – **y**|**a**|**w**|**d**|**h**|**m**|**s**|**c***period*[/*phase*][**+c***col*]. Convert the input *x*-coordinate to a cyclical coordinate, or a different column if selected via **+c***col*. The following cyclical coordinate transformations are supported:

- **y** - yearly cycle (normalized)
- **a** - annual cycle (monthly)
- **w** - weekly cycle (day)
- **d** - daily cycle (hour)
- **h** - hourly cycle (minute)
- **m** - minute cycle (second)
- **s** - second cycle (second)
- **c** - custom cycle (normalized)

Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#w-full.

Returns

output (*pandas.DataFrame or None*) – Return type depends on whether the `outfile` parameter is set:

- `pandas.DataFrame` table with (x, y, z) columns if `outfile` is not set.
- None if `outfile` is set (filtered output will be stored in file set by `outfile`).

pygmt.nearneighbor

`pygmt.nearneighbor`(*data=None*, *x=None*, *y=None*, *z=None*, ***, *empty=None*, *outgrid=None*, *spacing=None*, *sectors=None*, *region=None*, *search_radius=None*, *verbose=None*, *aspatial=None*, *binary=None*, *nodata=None*, *find=None*, *coltypes=None*, *gap=None*, *header=None*, *incols=None*, *registration=None*, *wrap=None*, ***kwargs*)

Grid table data using a “Nearest neighbor” algorithm

nearneighbor reads arbitrarily located (*x,y,z*[,*w*]) triples [quadruplets] and uses a nearest neighbor algorithm to assign a weighted average value to each node that has one or more data points within a search radius centered on the node with adequate coverage across a subset of the chosen sectors. The node value is computed as a weighted mean of the nearest point from each sector inside the search radius. The weighting function and the averaging used is given by:

$$w(r_i) = \frac{w_i}{1 + d(r_i)^2}, \quad d(r) = \frac{3r}{R}, \quad \bar{z} = \frac{\sum_i^n w(r_i) z_i}{\sum_i^n w(r_i)}$$

where n is the number of data points that satisfy the selection criteria and r_i is the distance from the node to the i’th data point. If no data weights are supplied then $w_i = 1$.

Fig. 1: Search geometry includes the search radius (R) which limits the points considered and the number of sectors (here 4), which restricts how points inside the search radius contribute to the value at the node. Only the closest point in each sector (red circles) contribute to the weighted estimate.

Takes a matrix, xyz triples, or a file name as input.

Must provide either `data` or `x`, `y`, and `z`.

Full option list at https://docs.generic-mapping-tools.org/latest/nearneighbor.html

Aliases:

- E = empty
- G = outgrid

- I = spacing
- N = sectors
- R = region
- S = search_radius
- V = verbose
- a = aspatial
- b = binary
- d = nodata
- e = find
- f = coltypes
- g = gap
- h = header
- i = incols
- r = registration
- w = wrap

Parameters

- **`data`** (*`str` or `numpy.ndarray` or `pandas.DataFrame` or `xarray.Dataset` or `geopandas.GeoDataFrame`*) – Pass in (x, y, z) or (longitude, latitude, elevation) values by providing a file name to an ASCII data table, a 2D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **`x/y/z`** (*`1d arrays`*) – Arrays of x and y coordinates and values z of the data points.
- **`spacing`** (*`str`*) – *xinc*[**+e**|**n**][/*yinc*[**+e**|**n**]]. *x_inc* [and optionally *y_inc*] is the grid spacing.
  - **Geographical (degrees) coordinates**: Optionally, append an increment unit. Choose among **m** to indicate arc minutes or **s** to indicate arc seconds. If one of the units **e**, **f**, **k**, **M**, **n** or **u** is appended instead, the increment is assumed to be given in meter, foot, km, mile, nautical mile or US survey foot, respectively, and will be converted to the equivalent degrees longitude at the middle latitude of the region (the conversion depends on PROJ_ELLIPSOID). If *y_inc* is given but set to 0 it will be reset equal to *x_inc*; otherwise it will be converted to degrees latitude.
  - **All coordinates**: If **+e** is appended then the corresponding max *x* (*east*) or *y* (*north*) may be slightly adjusted to fit exactly the given increment [by default the increment may be adjusted slightly to fit the given domain]. Finally, instead of giving an increment you may specify the *number of nodes* desired by appending **+n** to the supplied integer argument; the increment is then recalculated from the number of nodes, the *registration*, and the domain. The resulting increment value depends on whether you have selected a gridline-registered or pixel-registered grid; see GMT File Formats for details.

  Note: If `region=grdfile` is used then the grid spacing and the registration have already been initialized; use `spacing` and `registration` to override these values.
- **`region`** (*`str` or `list`*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.

- **search_radius** (*str*) – Sets the search radius that determines which data points are considered close to a node.
- **outgrid** (*str*) – Optional. The file name for the output netcdf file with extension .nc to store the grid in.
- **empty** (*str*) – Optional. Set the value assigned to empty nodes. Defaults to NaN.
- **sectors** (*str*) – *sectors*[**+m***min_sectors*]|**n**. Optional. The circular search area centered on each node is divided into *sectors* sectors. Average values will only be computed if there is *at least* one value inside each of at least *min_sectors* of the sectors for a given node. Nodes that fail this test are assigned the value NaN (but see `empty`). If **+m** is omitted then *min_sectors* is set to be at least 50% of *sectors* (i.e., rounded up to next integer) [Default is a quadrant search with 100% coverage, i.e., *sectors* = *min_sectors* = 4]. Note that only the nearest value per sector enters into the averaging; the more distant points are ignored. Alternatively, use `sectors="n"` to call GDAL's nearest neighbor algorithm instead.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **aspatial** (*bool or str*) – [*col*=]*name*[,...]. Control how aspatial data are handled during input and output. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#aspatial-full.
- **binary** (*bool or str*) – **i**|**o**[*ncols*][*type*][**w**][**+l**|**b**]. Select native binary input (using `binary="i"`) or output (using `binary="o"`), where *ncols* is the number of data columns of *type*, which must be one of:
  - **c** - int8_t (1-byte signed char)
  - **u** - uint8_t (1-byte unsigned char)
  - **h** - int16_t (2-byte signed int)
  - **H** - uint16_t (2-byte unsigned int)
  - **i** - int32_t (4-byte signed int)
  - **I** - uint32_t (4-byte unsigned int)
  - **l** - int64_t (8-byte signed int)
  - **L** - uint64_t (8-byte unsigned int)
  - **f** - 4-byte single-precision float
  - **d** - 8-byte double-precision float
  - **x** - use to skip *ncols* anywhere in the record

  For records with mixed types, append additional comma-separated combinations of *ncols* *type* (no space). The following modifiers are supported:

  - **w** after any item to force byte-swapping.
  - **+l|b** to indicate that the entire data file should be read as little- or big-endian, respectively.

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#bi-full.

- `nodata` (*str*) – **i|o***nodata*. Substitute specific values with NaN (for tabular data). For example, `d="-9999"` will replace all values equal to -9999 with NaN during input and all NaN values with -9999 during output. Prepend **i** to the *nodata* value for input columns only. Prepend **o** to the *nodata* value for output columns only.
- `find` (*str*) – [~]*"pattern"* | [~]/*regexp*/[**i**]. Only pass records that match the given *pattern* or regular expressions [Default processes all records]. Prepend **~** to the *pattern* or *regexp* to instead only pass data expressions that do not match the pattern. Append **i** for case insensitive matching. This does not apply to headers or segment headers.
- `coltypes` (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- `gap` (*str or list*) – [**a**]**x**|**y**|**d**|**X**|**Y**|**D**|[*col*]**z***gap*[**+n**|**p**]. Examine the spacing between consecutive data points in order to impose breaks in the line. To specify multiple criteria, provide a list with each item containing a string describing one set of criteria. Prepend **a** to specify that all the criteria must be met [Default is to impose breaks if any criteria are met]. The following modifiers are supported:
  - **x|X** - define a gap when there is a large enough change in the x coordinates (upper case to use projected coordinates).
  - **y|Y** - define a gap when there is a large enough change in the y coordinates (upper case to use projected coordinates).
  - **d|D** - define a gap when there is a large enough distance between coordinates (upper case to use projected coordinates).
  - [*col*]**z** - define a gap when there is a large enough change in the data in column *col* [default *col* is 2 (i.e., 3rd column)].

  A unit **u** may be appended to the specified *gap*:

  - For geographic data (**x**|**y**|**d**), the unit may be arc **d**(egree), **m**(inute), and **s**(econd), or (m)**e**(ter), **f**(eet), **k**(ilometer), **M**(iles), or **n**(autical miles) [Default is (m)**e**(ter)].
  - For projected data (**X**|**Y**|**D**), the unit may be **i**(nch), **c**(entimeter), or **p**(oint).

  One of the following modifiers can be appended to *gap* [Default imposes breaks based on the absolute value of the difference between the current and previous value]:

  - **+n** - specify that the previous value minus the current column value must exceed *gap* for a break to be imposed.
  - **+p** - specify that the current value minus the previous value must exceed *gap* for a break to be imposed.
- `header` (*str*) – [**i**|**o**][*n*][**+c**][**+d**][**+m***segheader*][**+r***remark*][**+t***title*]. Specify that input and/or output file(s) have *n* header records [Default is 0]. Prepend **i** if only the primary input should have header records. Prepend **o** to control the writing of header records, with the following modifiers supported:
  - **+d** to remove existing header records.
  - **+c** to add a header comment with column names to the output [Default is no column names].

  - **+m** to add a segment header *segheader* to the output after the header block [Default is no segment header].
  - **+r** to add a *remark* comment to the output [Default is no comment]. The *remark* string may contain \n to indicate line-breaks.
  - **+t** to add a *title* comment to the output [Default is no title]. The *title* string may contain \n to indicate line-breaks.

  Blank lines and lines starting with # are always skipped.

- **incols** (*str or 1d array*) – Specify data columns for primary input in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default reads all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in input order (e.g., `incols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `incols="0:2,4+l"` to input the first three columns followed by the log-transformed 5th column). To read from a given column until the end of the record, leave off *stop* when specifying the column range. To read trailing text, add the column **t**. Append the word number to **t** to ingest only a single word from the trailing text. Instead of specifying columns, use `incols="n"` to simply read numerical input and skip trailing text. Optionally, append one of the following modifiers to any column or column range to transform the input columns:
    - **+l** to take the *log10* of the input values.
    - **+d** to divide the input values by the factor *divisor* [Default is 1].
    - **+s** to multiple the input values by the factor *scale* [Default is 1].
    - **+o** to add the given *offset* to the input values [Default is 0].
- **registration** (*str*) – **g**|**p**. Force gridline (**g**) or pixel (**p**) node registration. [Default is **g**(ridline)].
- **wrap** (*str*) – **y**|**a**|**w**|**d**|**h**|**m**|**s**|**c***period*[/*phase*][**+c***col*]. Convert the input *x*-coordinate to a cyclical coordinate, or a different column if selected via **+c***col*. The following cyclical coordinate transformations are supported:
  - **y** - yearly cycle (normalized)
  - **a** - annual cycle (monthly)
  - **w** - weekly cycle (day)
  - **d** - daily cycle (hour)
  - **h** - hourly cycle (minute)
  - **m** - minute cycle (second)
  - **s** - second cycle (second)
  - **c** - custom cycle (normalized)

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#w-full.

Returns

ret (*xarray.DataArray or None*) – Return type depends on whether the `outgrid` parameter is set:

- `xarray.DataArray`: if `outgrid` is not set
- None if `outgrid` is set (grid output will be stored in file set by `outgrid`)

pygmt.project

`pygmt.project`(*data=None*, *x=None*, *y=None*, *z=None*, *outfile=None*, ***, *azimuth=None*, *center=None*, *endpoint=None*, *convention=None*, *generate=None*, *length=None*, *flat_earth=None*, *unit=None*, *sort=None*, *pole=None*, *verbose=None*, *width=None*, *ellipse=None*, *coltypes=None*, ***kwargs*)

Project data onto lines or great circles, or generate tracks.

Project reads arbitrary $(x, y[, z])$ data and returns any combination of (x, y, z, p, q, r, s), where (p, q) are the coordinates in the projection, (r, s) is the position in the (x, y) coordinate system of the point on the profile ($q = 0$ path) closest to (x, y), and z is all remaining columns in the input (beyond the required x and y columns).

Alternatively, `project` may be used to generate (r, s, p) triples at equal increments along a profile using the `generate` parameter. In this case, the value of `data` is ignored (you can use, e.g., `data=None`).

Projections are defined in any (but only) one of three ways:

1. By a `center` and an `azimuth` in degrees clockwise from North.
2. By a `center` and `endpoint` of the projection path.
3. By a `center` and a `pole` position.

To spherically project data along a great circle path, an oblique coordinate system is created which has its equator along that path, and the zero meridian through the Center. Then the oblique longitude (p) corresponds to the distance from the Center along the great circle, and the oblique latitude (q) corresponds to the distance perpendicular to the great circle path. When moving in the increasing (p) direction, (toward B or in the azimuth direction), the positive (q) direction is to your left. If a Pole has been specified, then the positive (q) direction is toward the pole.

To specify an oblique projection, use the `pole` option to set the pole. Then the equator of the projection is already determined and the `center` option is used to locate the $p = 0$ meridian. The center *cx/cy* will be taken as a point through which the $p = 0$ meridian passes. If you do not care to choose a particular point, use the South pole (*cx* = 0, *cy* = -90).

Data can be selectively windowed by using the `length` and `width` options. If `width` is used, the projection width is set to use only data with $w_{min} < q < w_{max}$. If `length` is set, then the length is set to use only those data with $l_{min} < p < l_{max}$. If the `endpoint` option has been used to define the projection, then `length="w"` may be used to window the length of the projection to exactly the span from O to B.

Flat Earth (Cartesian) coordinate transformations can also be made. Set `flat_earth=True` and remember that azimuth is clockwise from North (the y axis), NOT the usual cartesian theta, which is counterclockwise from the x axis. azimuth = 90 - theta.

No assumptions are made regarding the units for $x, y, r, s, p, q, dist, l_{min}, l_{max}, w_{min}, w_{max}$. If -Q is selected, map units are assumed and x, y, r, s must be in degrees and $p, q, dist, l_{min}, l_{max}, w_{min}, w_{max}$ will be in km.

Calculations of specific great-circle and geodesic distances or for back-azimuths or azimuths are better done using https://docs.generic-mapping-tools.org/latest/mapproject as project is strictly spherical.

Aliases:

- A = azimuth
- C = center
- E = endpoint
- F = convention

- G = generate
- L = length
- N = flat_earth
- Q = unit
- S = sort
- T = pole
- V = verbose
- W = width
- Z = ellipse
- f = coltypes

Parameters

- **data** (*str or numpy.ndarray or pandas.DataFrame or xarray.Dataset or geopandas.GeoDataFrame*) – Pass in (x, y, z) or (longitude, latitude, elevation) values by providing a file name to an ASCII data table, a 2D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **center** (*str or list*) – *cx*/*cy*. Set the origin of the projection, in Definition 1 or 2. If Definition 3 is used, then *cx/cy* are the coordinates of a point through which the oblique zero meridian ($p = 0$) should pass. The *cx/cy* is not required to be 90 degrees from the pole.
- **azimuth** (*float or str*) – Define the azimuth of the projection (Definition 1).
- **endpoint** (*str or list*) – *bx*/*by*. Define the end point of the projection path (Definition 2).
- **convention** (*str*) – Specify the desired output using any combination of **xyzpqrs**, in any order [Default is **xypqrsz**]. Do not space between the letters. Use lower case. The output will be columns of values corresponding to your `convention`. The **z** flag is special and refers to all numerical columns beyond the leading **x** and **y** in your input record. The **z** flag also includes any trailing text (which is placed at the end of the record regardless of the order of **z** in `convention`). **Note**: If `generate` is True, then the output order is hardwired to be **rsp** and `convention` is not allowed.
- **generate** (*str*) – *dist* [/*colat*][**+c**|**h**]. Create (r, s, p) output data every *dist* units of p (See *unit* option). Alternatively, append */colat* for a small circle instead [Default is a colatitude of 90, i.e., a great circle]. If setting a pole with `pole` and you want the small circle to go through *cx*/*cy*, append **+c** to compute the required colatitude. Use `center` and `endpoint` to generate a circle that goes through the center and end point. Note, in this case the center and end point cannot be farther apart than $2|\text{colat}|$. Finally, if you append **+h** then we will report the position of the pole as part of the segment header [Default is no header]. Note: No input is read and the value of `data`, `x`, `y`, and `z` is ignored if `generate` is used.
- **length** (*str or list*) – [**w**|*l_min*/*l_max*]. Project only those data whose *p* coordinate is within $l_{min} < p < l_{max}$. If `endpoint` has been set, then you may alternatively use **w** to stay within the distance from `center` to `endpoint`.
- **flat_earth** (*bool*) – Make a Cartesian coordinate transformation in the plane. [Default is `False`; plane created with spherical trigonometry.]
- **unit** (*bool*) – Set units for x, y, r, s degrees and $p, q, dist, l_{min}, l_{max}, w_{min}, w_max$ to km. [Default is `False`; all arguments use the same units]

- **sort** (*bool*) – Sort the output into increasing p order. Useful when projecting random data into a sequential profile.
- **pole** (*str or list*) – *px/py*. Set the position of the rotation pole of the projection. (Definition 3).
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **width** (*str or list*) – *w_min/w_max*. Project only those data whose q coordinate is within $w_{min} < q < w_{max}$.
- **ellipse** (*str*) – *major/minor/azimuth* [**+e**|**n**]. Used in conjunction with `center` (sets its center) and `generate` (sets the distance increment) to create the coordinates of an ellipse with *major* and *minor* axes given in km (unless `flat_earth` is given for a Cartesian ellipse) and the *azimuth* of the major axis in degrees. Append **+e** to adjust the increment set via `generate` so that the the ellipse has equal distance increments [Default uses the given increment and closes the ellipse]. Instead, append **+n** to set a specific number of unique equidistant data via `generate`. For degenerate ellipses you can just supply a single *diameter* instead. A geographic diameter may be specified in any desired unit other than km by appending the unit (e.g., 3d for degrees) [Default is km]; the increment is assumed to be in the same unit. **Note**: For the Cartesian ellipse (which requires `flat_earth`), the *direction* is counter-clockwise from the horizontal instead of an *azimuth*.
- **outfile** (*str*) – The file name for the output ASCII file.
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.

Returns

track (*pandas.DataFrame or None*) – Return type depends on whether the `outfile` parameter is set:

- `pandas.DataFrame` table with (x, y, ..., newcolname) if `outfile` is not set
- None if `outfile` is set (output will be stored in file set by `outfile`)

pygmt.select

`pygmt.select`(*data=None*, *outfile=None*, ***, *area_thresh=None*, *resolution=None*, *gridmask=None*, *reverse=None*, *projection=None*, *mask=None*, *region=None*, *verbose=None*, *z_subregion=None*, *binary=None*, *nodata=None*, *find=None*, *coltypes=None*, *gap=None*, *header=None*, *incols=None*, *outcols=None*, *registration=None*, *skiprows=None*, *wrap=None*, ***kwargs*)

Select data table subsets based on multiple spatial criteria.

This is a filter that reads (x, y) or (longitude, latitude) positions from the first 2 columns of *data* and uses a combination of 1-7 criteria to pass or reject the records. Records can be selected based on whether or not they are:

1. inside a rectangular region (**region** [and **projection**])
2. within *dist* km of any point in *pointfile*
3. within *dist* km of any line in *linefile*
4. inside one of the polygons in the *polygonfile*
5. inside geographical features (based on coastlines)
6. has z-values within a given range, or
7. inside bins of a grid mask whose nodes are non-zero

The sense of the tests can be reversed for each of these 7 criteria by using the **reverse** option.

Full option list at https://docs.generic-mapping-tools.org/latest/gmtselect.html

Aliases:

- A = area_thresh
- D = resolution
- G = gridmask
- I = reverse
- J = projection
- N = mask
- R = region
- V = verbose
- Z = z_subregion
- b = binary
- d = nodata
- e = find
- f = coltypes
- g = gap
- h = header
- i = incols
- o = outcols
- r = registration
- s = skiprows

- w = wrap

Parameters

- **data** (*str or numpy.ndarray or pandas.DataFrame or xarray.Dataset or geopandas.GeoDataFrame*) – Pass in either a file name to an ASCII data table, a 2D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **outfile** (*str*) – The file name for the output ASCII file.
- **area_thresh** (*int or float or str*) – *min_area*[/*min_level*/*max_level*][**+a**[**g**|**i**][**s**|**S**]][**+l**|**r**][**+p***percent*]. Features with an area smaller than *min_area* in km^2 or of hierarchical level that is lower than *min_level* or higher than *max_level* will not be plotted [Default is 0/0/4 (all features)].
- **resolution** (*str*) – *resolution*[**+f**]. Ignored unless **mask** is set. Selects the resolution of the coastline data set to use ((**f**)ull, (**h**)igh, (**i**)ntermediate, (**l**)ow, or (**c**)rude). The resolution drops off by ~80% between data sets. [Default is **l**]. Append (**+f**) to automatically select a lower resolution should the one requested not be available [Default is abort if not found]. Note that because the coastlines differ in details it is not guaranteed that a point will remain inside [or outside] when a different resolution is selected.
- **gridmask** (*str*) – Pass all locations that are inside the valid data area of the grid *gridmask*. Nodes that are outside are either NaN or zero.
- **reverse** (*str*) – [**cflrsz**]. Reverses the sense of the test for each of the criteria specified:
  - **c** select records NOT inside any point's circle of influence.
  - **f** select records NOT inside any of the polygons.
  - **g** will pass records inside the cells with z equal zero of the grid mask in **gridmask**.
  - **l** select records NOT within the specified distance of any line.
  - **r** select records NOT inside the specified rectangular region.
  - **s** select records NOT considered inside as specified by **mask** (and **area_thresh**, **resolution**).
  - **z** select records NOT within the range specified by **z_subregion**.
- **projection** (*str*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- **mask** (*str or list*) – Pass all records whose location is inside specified geographical features. Specify if records should be skipped (s) or kept (k) using 1 of 2 formats:
  - *wet/dry*.
  - *ocean/land/lake/island/pond*.

  [Default is s/k/s/k/s (i.e., s/k), which passes all points on dry land].
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]

  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages

- **z_subregion** (*str*) – *min*[/*max*][**+a**][**+c***col*][**+i**]. Pass all records whose 3rd column (*z*; *col* = 2) lies within the given range or is NaN (use **skiprows** to skip NaN records). If *max* is omitted then we test if *z* equals *min* instead. This means equality within 5 ULPs (unit of least precision; http://en.wikipedia.org/wiki/Unit_in_the_last_place). Input file must have at least three columns. To indicate no limit on min or max, specify a hyphen (-). If your 3rd column is absolute time then remember to supply `coltypes="2T"`. To specify another column, append **+c***col*, and to specify several tests just repeat the **z_subregion** option as many times as you have columns to test. **Note**: When more than one **z_subregion** option is given then the `reverse="z"` option cannot be used. In the case of multiple tests you may use these modifiers as well: **+a** passes any record that passes at least one of your *z* tests [Default is all tests must pass], and **+i** reverses the tests to pass record with *z* value NOT in the given range. Finally, if **+c** is not used then it is automatically incremented for each new **z_subregion** option, starting with 2.

- **binary** (*bool or str*) – **i**|**o**[*ncols*][*type*][**w**][**+l**|**b**]. Select native binary input (using `binary="i"`) or output (using `binary="o"`), where *ncols* is the number of data columns of *type*, which must be one of:

  - **c** - int8_t (1-byte signed char)
  - **u** - uint8_t (1-byte unsigned char)
  - **h** - int16_t (2-byte signed int)
  - **H** - uint16_t (2-byte unsigned int)
  - **i** - int32_t (4-byte signed int)
  - **I** - uint32_t (4-byte unsigned int)
  - **l** - int64_t (8-byte signed int)
  - **L** - uint64_t (8-byte unsigned int)
  - **f** - 4-byte single-precision float
  - **d** - 8-byte double-precision float
  - **x** - use to skip *ncols* anywhere in the record

  For records with mixed types, append additional comma-separated combinations of *ncols* *type* (no space). The following modifiers are supported:

  - **w** after any item to force byte-swapping.
  - **+l**|**b** to indicate that the entire data file should be read as little- or big-endian, respectively.

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#bi-full.

- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `d="-9999"` will replace all values equal to -9999 with NaN during input and all NaN values with -9999 during output. Prepend **i** to the *nodata* value for input columns only. Prepend **o** to the *nodata* value for output columns only.

- **find** (*str*) – [**~**]*"pattern"* | [**~**]/*regexp*/[**i**]. Only pass records that match the given *pattern* or regular expressions [Default processes all records]. Prepend **~** to the *pattern* or *regexp* to instead only pass data expressions that do not match the pattern. Append **i** for case insensitive matching. This does not apply to headers or segment headers.

- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **gap** (*str or list*) – [**a**]**x**|**y**|**d**|**X**|**Y**|**D**|[*col*]**z***gap*[**+n**|**p**]. Examine the spacing between consecutive data points in order to impose breaks in the line. To specify multiple criteria, provide a list with each item containing a string describing one set of criteria. Prepend **a** to specify that all the criteria must be met [Default is to impose breaks if any criteria are met]. The following modifiers are supported:
  - **x**|**X** - define a gap when there is a large enough change in the x coordinates (upper case to use projected coordinates).
  - **y**|**Y** - define a gap when there is a large enough change in the y coordinates (upper case to use projected coordinates).
  - **d**|**D** - define a gap when there is a large enough distance between coordinates (upper case to use projected coordinates).
  - [*col*]**z** - define a gap when there is a large enough change in the data in column *col* [default *col* is 2 (i.e., 3rd column)].

  A unit **u** may be appended to the specified *gap*:
  - For geographic data (**x**|**y**|**d**), the unit may be arc **d**(egree), **m**(inute), and **s**(econd), or (m)**e**(ter), **f**(eet), **k**(ilometer), **M**(iles), or **n**(autical miles) [Default is (m)**e**(ter)].
  - For projected data (**X**|**Y**|**D**), the unit may be **i**(nch), **c**(entimeter), or **p**(oint).

  One of the following modifiers can be appended to *gap* [Default imposes breaks based on the absolute value of the difference between the current and previous value]:
  - **+n** - specify that the previous value minus the current column value must exceed *gap* for a break to be imposed.
  - **+p** - specify that the current value minus the previous value must exceed *gap* for a break to be imposed.
- **header** (*str*) – [**i**|**o**][*n*][**+c**][**+d**][**+m***segheader*][**+r***remark*][**+t***title*]. Specify that input and/or output file(s) have *n* header records [Default is 0]. Prepend **i** if only the primary input should have header records. Prepend **o** to control the writing of header records, with the following modifiers supported:
  - **+d** to remove existing header records.
  - **+c** to add a header comment with column names to the output [Default is no column names].
  - **+m** to add a segment header *segheader* to the output after the header block [Default is no segment header].
  - **+r** to add a *remark* comment to the output [Default is no comment]. The *remark* string may contain \n to indicate line-breaks.
  - **+t** to add a *title* comment to the output [Default is no title]. The *title* string may contain \n to indicate line-breaks.

  Blank lines and lines starting with # are always skipped.

- **incols** (*str or 1d array*) – Specify data columns for primary input in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default reads all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in input order (e.g., `incols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `incols="0:2,4+l"` to input the first three columns followed by the log-transformed 5th column). To read from a given column until the end of the record, leave off *stop* when specifying the column range. To read trailing text, add the column **t**. Append the word number to **t** to ingest only a single word from the trailing text. Instead of specifying columns, use `incols="n"` to simply read numerical input and skip trailing text. Optionally, append one of the following modifiers to any column or column range to transform the input columns:
    - **+l** to take the *log10* of the input values.
    - **+d** to divide the input values by the factor *divisor* [Default is 1].
    - **+s** to multiple the input values by the factor *scale* [Default is 1].
    - **+o** to add the given *offset* to the input values [Default is 0].
- **outcols** (*str or 1d array*) – *cols*[,...][,**t**[*word*]]. Specify data columns for primary output in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default writes all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in output order (e.g., `outcols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `outcols="0:2,4"` to output the first three columns followed by the 5th column). To write from a given column until the end of the record, leave off *stop* when specifying the column range. To write trailing text, add the column **t**. Append the word number to **t** to write only a single word from the trailing text. Instead of specifying columns, use `outcols="n"` to simply read numerical input and skip trailing text. Note: if `incols` is also used then the columns given to `outcols` correspond to the order after the `incols` selection has taken place.
- **registration** (*str*) – **g**|**p**. Force gridline (**g**) or pixel (**p**) node registration. [Default is **g**(ridline)].
- **skiprows** (*bool or str*) – [*cols*][**+a**][**+r**]. Suppress output for records whose *z*-value equals NaN [Default outputs all records]. Optionally, supply a comma-separated list of all columns or column ranges to consider for this NaN test [Default only considers the third data column (i.e., *cols = 2*)]. Column ranges must be given in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified. The following modifiers are supported:
  - **+r** to reverse the suppression, i.e., only output the records whose *z*-value equals NaN.
  - **+a** to suppress the output of the record if just one or more of the columns equal NaN [Default skips record only if values in all specified *cols* equal NaN].
- **wrap** (*str*) – **y**|**a**|**w**|**d**|**h**|**m**|**s**|**c***period*[/*phase*][**+c***col*]. Convert the input *x*-coordinate to a cyclical coordinate, or a different column if selected via **+c***col*. The following cyclical coordinate transformations are supported:
  - **y** - yearly cycle (normalized)

  - **a** - annual cycle (monthly)
  - **w** - weekly cycle (day)
  - **d** - daily cycle (hour)
  - **h** - hourly cycle (minute)
  - **m** - minute cycle (second)
  - **s** - second cycle (second)
  - **c** - custom cycle (normalized)

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#w-full.

Returns

output (*pandas.DataFrame or None*) – Return type depends on whether the `outfile` parameter is set:

- `pandas.DataFrame` table if `outfile` is not set.
- None if `outfile` is set (filtered output will be stored in file set by `outfile`).

pygmt.sph2grd

`pygmt.sph2grd`(*data*, ***, *outgrid=None*, *spacing=None*, *region=None*, *verbose=None*, *binary=None*, *header=None*, *incols=None*, *registration=None*, *cores=None*, ***kwargs*)

Create spherical grid files in tension of data.

Reads a spherical harmonics coefficient table with records of L, M, C[L,M], S[L,M] and evaluates the spherical harmonic model on the specified grid.

Full option list at https://docs.generic-mapping-tools.org/latest/sph2grd.html

Aliases:

- G = outgrid
- I = spacing
- R = region
- V = verbose
- b = binary
- h = header
- i = incols
- r = registration
- x = cores

Parameters

- **data** (*`str` or `numpy.ndarray` or `pandas.DataFrame` or `xarray.Dataset` or `geopandas.GeoDataFrame`*) – Pass in data with L, M, C[L,M], S[L,M] values by providing a file name to an ASCII data table, a 2D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **outgrid** (*`str` or `None`*) – The name of the output netCDF file with extension .nc to store the grid in.

- `spacing` (*str*) – *xinc*[**+e**|**n**][/*yinc*[**+e**|**n**]]. *x_inc* [and optionally *y_inc*] is the grid spacing.
  - **Geographical (degrees) coordinates**: Optionally, append an increment unit. Choose among **m** to indicate arc minutes or **s** to indicate arc seconds. If one of the units **e**, **f**, **k**, **M**, **n** or **u** is appended instead, the increment is assumed to be given in meter, foot, km, mile, nautical mile or US survey foot, respectively, and will be converted to the equivalent degrees longitude at the middle latitude of the region (the conversion depends on PROJ_ELLIPSOID). If *y_inc* is given but set to 0 it will be reset equal to *x_inc*; otherwise it will be converted to degrees latitude.
  - **All coordinates**: If **+e** is appended then the corresponding max *x* (*east*) or *y* (*north*) may be slightly adjusted to fit exactly the given increment [by default the increment may be adjusted slightly to fit the given domain]. Finally, instead of giving an increment you may specify the *number of nodes* desired by appending **+n** to the supplied integer argument; the increment is then recalculated from the number of nodes, the *registration*, and the domain. The resulting increment value depends on whether you have selected a gridline-registered or pixel-registered grid; see GMT File Formats for details.

  Note: If `region=grdfile` is used then the grid spacing and the registration have already been initialized; use `spacing` and `registration` to override these values.
- `region` (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- `verbose` (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- `binary` (*bool or str*) – **i**|**o**[*ncols*][*type*][**w**][**+l**|**b**]. Select native binary input (using `binary="i"`) or output (using `binary="o"`), where *ncols* is the number of data columns of *type*, which must be one of:
  - **c** - int8_t (1-byte signed char)
  - **u** - uint8_t (1-byte unsigned char)
  - **h** - int16_t (2-byte signed int)
  - **H** - uint16_t (2-byte unsigned int)
  - **i** - int32_t (4-byte signed int)
  - **I** - uint32_t (4-byte unsigned int)
  - **l** - int64_t (8-byte signed int)
  - **L** - uint64_t (8-byte unsigned int)
  - **f** - 4-byte single-precision float
  - **d** - 8-byte double-precision float
  - **x** - use to skip *ncols* anywhere in the record

For records with mixed types, append additional comma-separated combinations of *ncols* *type* (no space). The following modifiers are supported:

- **w** after any item to force byte-swapping.
- **+l|b** to indicate that the entire data file should be read as little- or big-endian, respectively.

Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#bi-full.

- **header** (*str*) – [**i**|**o**][*n*][**+c**][**+d**][**+m***segheader*][**+r***remark*][**+t***title*]. Specify that input and/or output file(s) have *n* header records [Default is 0]. Prepend **i** if only the primary input should have header records. Prepend **o** to control the writing of header records, with the following modifiers supported:
  - **+d** to remove existing header records.
  - **+c** to add a header comment with column names to the output [Default is no column names].
  - **+m** to add a segment header *segheader* to the output after the header block [Default is no segment header].
  - **+r** to add a *remark* comment to the output [Default is no comment]. The *remark* string may contain \n to indicate line-breaks.
  - **+t** to add a *title* comment to the output [Default is no title]. The *title* string may contain \n to indicate line-breaks.

  Blank lines and lines starting with # are always skipped.
- **incols** (*str or 1d array*) – Specify data columns for primary input in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default reads all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in input order (e.g., `incols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `incols="0:2,4+l"` to input the first three columns followed by the log-transformed 5th column). To read from a given column until the end of the record, leave off *stop* when specifying the column range. To read trailing text, add the column **t**. Append the word number to **t** to ingest only a single word from the trailing text. Instead of specifying columns, use `incols="n"` to simply read numerical input and skip trailing text. Optionally, append one of the following modifiers to any column or column range to transform the input columns:
    - **+l** to take the *log10* of the input values.
    - **+d** to divide the input values by the factor *divisor* [Default is 1].
    - **+s** to multiple the input values by the factor *scale* [Default is 1].
    - **+o** to add the given *offset* to the input values [Default is 0].
- **registration** (*str*) – **g**|**p**. Force gridline (**g**) or pixel (**p**) node registration. [Default is **g**(ridline)].
- **cores** (*bool or int*) – [[**-**]*n*]. Limit the number of cores to be used in any OpenMP-enabled multi-threaded algorithms. By default we try to use all available cores. Set a number *n* to only use n cores (if too large it will be truncated to the maximum cores available). Finally, give a negative number *-n* to select (all - *n*) cores (or at least 1 if *n* equals or exceeds all).

Returns

ret (*xarray.DataArray or None*) – Return type depends on whether the `outgrid` parameter is set:

- `xarray.DataArray` if `outgrid` is not set
- None if `outgrid` is set (grid output will be stored in file set by `outgrid`)

pygmt.sphdistance

`pygmt.sphdistance`(*data=None*, *x=None*, *y=None*, ***, *single_form=None*, *duplicate=None*, *quantity=None*, *outgrid=None*, *spacing=None*, *unit=None*, *node_table=None*, *voronoi=None*, *region=None*, *verbose=None*, ***kwargs*)

Create Voronoi distance, node, or natural nearest-neighbor grid on a sphere.

Reads a table containing *lon, lat* columns and performs the construction of Voronoi polygons. These polygons are then processed to calculate the nearest distance to each node of the lattice and written to the specified grid.

Full option list at :gmt-docs:`sphdistance.html

Aliases:

- C = single_form
- D = duplicate
- E = quantity
- G = outgrid
- I = spacing
- L = unit
- N = node_table
- Q = voronoi
- R = region
- V = verbose

Parameters

- **data** (*str or numpy.ndarray or pandas.DataFrame or xarray.Dataset or geopandas.GeoDataFrame*) – Pass in (x, y) or (longitude, latitude) values by providing a file name to an ASCII data table, a 2D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **x/y** (*1d arrays*) – Arrays of x and y coordinates.
- **outgrid** (*str or None*) – The name of the output netCDF file with extension .nc to store the grid in.
- **spacing** (*str*) – *xinc*[**+e**|**n**][/*yinc*[**+e**|**n**]]. *x_inc* [and optionally *y_inc*] is the grid spacing.
  - **Geographical (degrees) coordinates**: Optionally, append an increment unit. Choose among **m** to indicate arc minutes or **s** to indicate arc seconds. If one of the units **e**, **f**, **k**, **M**, **n** or **u** is appended instead, the increment is assumed to be given in meter, foot, km, mile, nautical mile or US survey foot, respectively, and will be converted to the equivalent degrees longitude at the middle latitude of the region (the conversion depends on

PROJ_ELLIPSOID). If *y_inc* is given but set to 0 it will be reset equal to *x_inc*; otherwise it will be converted to degrees latitude.

– **All coordinates**: If **+e** is appended then the corresponding max *x* (*east*) or *y* (*north*) may be slightly adjusted to fit exactly the given increment [by default the increment may be adjusted slightly to fit the given domain]. Finally, instead of giving an increment you may specify the *number of nodes* desired by appending **+n** to the supplied integer argument; the increment is then recalculated from the number of nodes, the *registration*, and the domain. The resulting increment value depends on whether you have selected a gridline-registered or pixel-registered grid; see GMT File Formats for details.

Note: If `region=grdfile` is used then the grid spacing and the registration have already been initialized; use `spacing` and `registration` to override these values.

- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  – **q** - Quiet, not even fatal error messages are produced
  – **e** - Error messages only
  – **w** - Warnings [Default]
  – **t** - Timings (report runtimes for time-intensive algorithms);
  – **i** - Informational messages (same as `verbose=True`)
  – **c** - Compatibility warnings
  – **d** - Debugging messages
- **single_form** (*bool*) – For large data sets you can save some memory (at the expense of more processing) by only storing one form of location coordinates (geographic or Cartesian 3-D vectors) at any given time, translating from one form to the other when necessary [Default keeps both arrays in memory]. Not applicable with `voronoi`.
- **duplicate** (*bool*) – Used to skip duplicate points since the algorithm cannot handle them. [Default assumes there are no duplicates].
- **quantity** (*str*) – **d**|**n**|**z**[*dist*]. Specify the quantity that should be assigned to the grid nodes [Default is **d**]:
  – **d** - compute distances to the nearest data point
  – **n** - assign the ID numbers of the Voronoi polygons that each grid node is inside
  – **z** - assign all nodes inside the polygon the z-value of the center node for a natural nearest-neighbor grid.

  Optionally, append the resampling interval along Voronoi arcs in spherical degrees.
- **unit** (*str*) – Specify the unit used for distance calculations. Choose among **d** (spherical degree), **e** (m), **f** (feet), **k** (km), **M** (mile), **n** (nautical mile) or **u** survey foot.
- **node_table** (*str*) – Read the information pertaining to each Voronoi polygon (the unique node lon, lat and polygon area) from a separate file [Default acquires this information from the ASCII segment headers of the output file]. Required if binary input via *voronoi* is used.
- **voronoi** (*str*) – Append the name of a file with pre-calculated Voronoi polygons [Default performs the Voronoi construction on input data].

Returns

ret (*xarray.DataArray or None*) – Return type depends on whether the `outgrid` parameter is set:

- `xarray.DataArray` if `outgrid` is not set
- None if `outgrid` is set (grid output will be stored in file set by `outgrid`)

pygmt.sphinterpolate

`pygmt.sphinterpolate`(*data*, ***, *outgrid=None*, *spacing=None*, *region=None*, *verbose=None*, ***kwargs*)

Create spherical grid files in tension of data.

Reads a table containing *lon, lat, z* columns and performs a Delaunay triangulation to set up a spherical interpolation in tension. Several options may be used to affect the outcome, such as choosing local versus global gradient estimation or optimize the tension selection to satisfy one of four criteria.

Full option list at https://docs.generic-mapping-tools.org/latest/sphinterpolate.html

Aliases:

- G = outgrid
- I = spacing
- R = region
- V = verbose

Parameters

- **data** (*str or numpy.ndarray or pandas.DataFrame or xarray.Dataset or geopandas.GeoDataFrame*) – Pass in (x, y, z) or (longitude, latitude, elevation) values by providing a file name to an ASCII data table, a 2D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **outgrid** (*str or None*) – The name of the output netCDF file with extension .nc to store the grid in.
- **spacing** (*str*) – *xinc*[**+e**|**n**][/*yinc*[**+e**|**n**]]. *x_inc* [and optionally *y_inc*] is the grid spacing.
  - **Geographical (degrees) coordinates**: Optionally, append an increment unit. Choose among **m** to indicate arc minutes or **s** to indicate arc seconds. If one of the units **e**, **f**, **k**, **M**, **n** or **u** is appended instead, the increment is assumed to be given in meter, foot, km, mile, nautical mile or US survey foot, respectively, and will be converted to the equivalent degrees longitude at the middle latitude of the region (the conversion depends on PROJ_ELLIPSOID). If *y_inc* is given but set to 0 it will be reset equal to *x_inc*; otherwise it will be converted to degrees latitude.
  - **All coordinates**: If **+e** is appended then the corresponding max *x* (*east*) or *y* (*north*) may be slightly adjusted to fit exactly the given increment [by default the increment may be adjusted slightly to fit the given domain]. Finally, instead of giving an increment you may specify the *number of nodes* desired by appending **+n** to the supplied integer argument; the increment is then recalculated from the number of nodes, the *registration*, and the domain. The resulting increment value depends on whether you have selected a gridline-registered or pixel-registered grid; see GMT File Formats for details.

  Note: If `region=grdfile` is used then the grid spacing and the registration have already been initialized; use `spacing` and `registration` to override these values.

- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages

Returns

ret (*xarray.DataArray or None*) – Return type depends on whether the `outgrid` parameter is set:

- `xarray.DataArray` if `outgrid` is not set
- None if `outgrid` is set (grid output will be stored in file set by `outgrid`)

pygmt.surface

`pygmt.surface`(*data=None*, *x=None*, *y=None*, *z=None*, ***, *spacing=None*, *region=None*, *outgrid=None*, *verbose=None*, *aspatial=None*, *binary=None*, *nodata=None*, *find=None*, *coltypes=None*, *header=None*, *incols=None*, *registration=None*, *wrap=None*, ***kwargs*)

Grids table data using adjustable tension continuous curvature splines.

Surface reads randomly-spaced (x,y,z) triples and produces gridded values z(x,y) by solving:

$$(1 - t)\nabla^2(z) + t\nabla(z) = 0$$

where t is a tension factor between 0 and 1, and ∇ indicates the Laplacian operator.

Takes a matrix, xyz triples, or a file name as input.

Must provide either `data` or `x`, `y`, and `z`.

Full option list at https://docs.generic-mapping-tools.org/latest/surface.html

Aliases:

- G = outgrid
- I = spacing
- R = region
- V = verbose
- a = aspatial
- b = binary
- d = nodata
- e = find

- f = coltypes
- h = header
- i = incols
- r = registration
- w = wrap

Parameters

- **data** (*str or numpy.ndarray or pandas.DataFrame or xarray.Dataset or geopandas.GeoDataFrame*) – Pass in (x, y, z) or (longitude, latitude, elevation) values by providing a file name to an ASCII data table, a 2D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **x/y/z** (*1d arrays*) – Arrays of x and y coordinates and values z of the data points.
- **spacing** (*str*) – *xinc*[**+e**|**n**][/*yinc*[**+e**|**n**]]. *x_inc* [and optionally *y_inc*] is the grid spacing.
  - **Geographical (degrees) coordinates**: Optionally, append an increment unit. Choose among **m** to indicate arc minutes or **s** to indicate arc seconds. If one of the units **e**, **f**, **k**, **M**, **n** or **u** is appended instead, the increment is assumed to be given in meter, foot, km, mile, nautical mile or US survey foot, respectively, and will be converted to the equivalent degrees longitude at the middle latitude of the region (the conversion depends on PROJ_ELLIPSOID). If *y_inc* is given but set to 0 it will be reset equal to *x_inc*; otherwise it will be converted to degrees latitude.
  - **All coordinates**: If **+e** is appended then the corresponding max *x* (*east*) or *y* (*north*) may be slightly adjusted to fit exactly the given increment [by default the increment may be adjusted slightly to fit the given domain]. Finally, instead of giving an increment you may specify the *number of nodes* desired by appending **+n** to the supplied integer argument; the increment is then recalculated from the number of nodes, the *registration*, and the domain. The resulting increment value depends on whether you have selected a gridline-registered or pixel-registered grid; see GMT File Formats for details.

  Note: If `region=grdfile` is used then the grid spacing and the registration have already been initialized; use `spacing` and `registration` to override these values.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **outgrid** (*str*) – Optional. The file name for the output netcdf file with extension .nc to store the grid in.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages

- **aspatial** (*bool or str*) – [*col*=]*name*[,…]. Control how aspatial data are handled during input and output. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#aspatial-full.
- **binary** (*bool or str*) – **i**|**o**[*ncols*][*type*][**w**][**+l**|**b**]. Select native binary input (using `binary="i"`) or output (using `binary="o"`), where *ncols* is the number of data columns of *type*, which must be one of:
  - **c** - int8_t (1-byte signed char)
  - **u** - uint8_t (1-byte unsigned char)
  - **h** - int16_t (2-byte signed int)
  - **H** - uint16_t (2-byte unsigned int)
  - **i** - int32_t (4-byte signed int)
  - **I** - uint32_t (4-byte unsigned int)
  - **l** - int64_t (8-byte signed int)
  - **L** - uint64_t (8-byte unsigned int)
  - **f** - 4-byte single-precision float
  - **d** - 8-byte double-precision float
  - **x** - use to skip *ncols* anywhere in the record

  For records with mixed types, append additional comma-separated combinations of *ncols type* (no space). The following modifiers are supported:

  - **w** after any item to force byte-swapping.
  - **+l**|**b** to indicate that the entire data file should be read as little- or big-endian, respectively.

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#bi-full.
- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `d="-9999"` will replace all values equal to -9999 with NaN during input and all NaN values with -9999 during output. Prepend **i** to the *nodata* value for input columns only. Prepend **o** to the *nodata* value for output columns only.
- **find** (*str*) – [**~**]*"pattern"* | [**~**]/*regexp*/[**i**]. Only pass records that match the given *pattern* or regular expressions [Default processes all records]. Prepend **~** to the *pattern* or *regexp* to instead only pass data expressions that do not match the pattern. Append **i** for case insensitive matching. This does not apply to headers or segment headers.
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **header** (*str*) – [**i**|**o**][*n*][**+c**][**+d**][**+m***segheader*][**+r***remark*][**+t***title*]. Specify that input and/or output file(s) have *n* header records [Default is 0]. Prepend **i** if only the primary input should have header records. Prepend **o** to control the writing of header records, with the following modifiers supported:
  - **+d** to remove existing header records.
  - **+c** to add a header comment with column names to the output [Default is no column names].
  - **+m** to add a segment header *segheader* to the output after the header block [Default is no segment header].

- **+r** to add a *remark* comment to the output [Default is no comment]. The *remark* string may contain \n to indicate line-breaks.
- **+t** to add a *title* comment to the output [Default is no title]. The *title* string may contain \n to indicate line-breaks.

Blank lines and lines starting with # are always skipped.

- **incols** (*str or 1d array*) – Specify data columns for primary input in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default reads all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in input order (e.g., `incols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `incols="0:2,4+l"` to input the first three columns followed by the log-transformed 5th column). To read from a given column until the end of the record, leave off *stop* when specifying the column range. To read trailing text, add the column **t**. Append the word number to **t** to ingest only a single word from the trailing text. Instead of specifying columns, use `incols="n"` to simply read numerical input and skip trailing text. Optionally, append one of the following modifiers to any column or column range to transform the input columns:
    - **+l** to take the *log10* of the input values.
    - **+d** to divide the input values by the factor *divisor* [Default is 1].
    - **+s** to multiple the input values by the factor *scale* [Default is 1].
    - **+o** to add the given *offset* to the input values [Default is 0].
- **registration** (*str*) – **g**|**p**. Force gridline (**g**) or pixel (**p**) node registration. [Default is **g**(ridline)].
- **wrap** (*str*) – **y**|**a**|**w**|**d**|**h**|**m**|**s**|**c***period*[/*phase*][**+c***col*]. Convert the input *x*-coordinate to a cyclical coordinate, or a different column if selected via **+c***col*. The following cyclical coordinate transformations are supported:
  - **y** - yearly cycle (normalized)
  - **a** - annual cycle (monthly)
  - **w** - weekly cycle (day)
  - **d** - daily cycle (hour)
  - **h** - hourly cycle (minute)
  - **m** - minute cycle (second)
  - **s** - second cycle (second)
  - **c** - custom cycle (normalized)

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#w-full.

Returns

ret (*xarray.DataArray or None*) – Return type depends on whether the `outgrid` parameter is set:

- `xarray.DataArray`: if `outgrid` is not set
- None if `outgrid` is set (grid output will be stored in file set by `outgrid`)

pygmt.xyz2grd

`pygmt.xyz2grd`(*data=None*, *x=None*, *y=None*, *z=None*, ***, *duplicate=None*, *outgrid=None*, *spacing=None*, *projection=None*, *region=None*, *verbose=None*, *convention=None*, *binary=None*, *nodata=None*, *find=None*, *coltypes=None*, *header=None*, *incols=None*, *registration=None*, *wrap=None*, ***kwargs*)

Create a grid file from table data.

Reads one or more tables with *x, y, z* columns and creates a binary grid file. xyz2grd will report if some of the nodes are not filled in with data. Such unconstrained nodes are set to a value specified by the user [Default is NaN]. Nodes with more than one value will be set to the mean value.

Full option list at https://docs.generic-mapping-tools.org/latest/xyz2grd.html

Aliases:

- A = duplicate
- G = outgrid
- I = spacing
- J = projection
- R = region
- V = verbose
- Z = convention
- b = binary
- d = nodata
- e = find
- f = coltypes
- h = header
- i = incols
- r = registration
- w = wrap

Parameters

- **data** (*str or numpy.ndarray or pandas.DataFrame or xarray.Dataset or geopandas.GeoDataFrame*) – Pass in (x, y, z) or (longitude, latitude, elevation) values by providing a file name to an ASCII data table, a 2D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **x/y/z** (*1d arrays*) – The arrays of x and y coordinates and z data points.
- **outgrid** (*str or None*) – Optional. The name of the output netCDF file with extension .nc to store the grid in.
- **duplicate** (*str*) – [**d**|**f**|**l**|**m**|**n**|**r**|**S**|**s**|**u**|**z**]. By default we will calculate mean values if multiple entries fall on the same node. Use **-A** to change this behavior, except it is ignored if **-Z** is given. Append **f** or **s** to simply keep the first or last data point that was assigned to each node. Append **l** or **u** or **d** to find the lowest (minimum) or upper (maximum) value or the difference between the maximum and miminum value at each node, respectively. Append **m** or **r** or **S** to compute mean or RMS value or standard deviation at each node, respectively.

Append **n** to simply count the number of data points that were assigned to each node (this only requires two input columns *x* and *y* as *z* is not consulted). Append **z** to sum multiple values that belong to the same node.

- `spacing` (*str*) – *xinc*[**+e**|**n**][/*yinc*[**+e**|**n**]]. *x_inc* [and optionally *y_inc*] is the grid spacing.
  - **Geographical (degrees) coordinates**: Optionally, append an increment unit. Choose among **m** to indicate arc minutes or **s** to indicate arc seconds. If one of the units **e**, **f**, **k**, **M**, **n** or **u** is appended instead, the increment is assumed to be given in meter, foot, km, mile, nautical mile or US survey foot, respectively, and will be converted to the equivalent degrees longitude at the middle latitude of the region (the conversion depends on PROJ_ELLIPSOID). If *y_inc* is given but set to 0 it will be reset equal to *x_inc*; otherwise it will be converted to degrees latitude.
  - **All coordinates**: If **+e** is appended then the corresponding max *x* (*east*) or *y* (*north*) may be slightly adjusted to fit exactly the given increment [by default the increment may be adjusted slightly to fit the given domain]. Finally, instead of giving an increment you may specify the *number of nodes* desired by appending **+n** to the supplied integer argument; the increment is then recalculated from the number of nodes, the *registration*, and the domain. The resulting increment value depends on whether you have selected a gridline-registered or pixel-registered grid; see GMT File Formats for details.

  Note: If `region=grdfile` is used then the grid spacing and the registration have already been initialized; use `spacing` and `registration` to override these values.
- `projection` (*str*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- `region` (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- `verbose` (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- `convention` (*str*) – [*flags*]. Read a 1-column ASCII [or binary] table. This assumes that all the nodes are present and sorted according to specified ordering convention contained in *flags*. If incoming data represents rows, make *flags* start with **T**(op) if first row is y = ymax or **B**(ottom) if first row is y = ymin. Then, append **L** or **R** to indicate that first element is at left or right end of row. Likewise for column formats: start with **L** or **R** to position first column, and then append **T** or **B** to position first element in a row. **Note**: These two row/column indicators are only required for grids; for other tables they do not apply. For gridline registered grids: If data are periodic in x but the incoming data do not contain the (redundant) column at x = xmax, append **x**. For data periodic in y without redundant row at y = ymax, append **y**. Append **s***n* to skip the first *n* number of bytes (probably a header). If the byte-order or the words needs to be swapped, append **w**. Select one of several data types (all binary except **a**):
  - **A** ASCII representation of one or more floating point values per record

- **a** ASCII representation of a single item per record
- **c** int8_t, signed 1-byte character
- **u** uint8_t, unsigned 1-byte character
- **h** int16_t, signed 2-byte integer
- **H** uint16_t, unsigned 2-byte integer
- **i** int32_t, signed 4-byte integer
- **I** uint32_t, unsigned 4-byte integer
- **l** int64_t, long (8-byte) integer
- **L** uint64_t, unsigned long (8-byte) integer
- **f** 4-byte floating point single precision
- **d** 8-byte floating point double precision

Default format is scanline orientation of ASCII numbers: **-ZTLa**. The difference between **A** and **a** is that the latter can decode both *date***T***clock* and *ddd:mm:ss[.xx]* formats but expects each input record to have a single value, while the former can handle multiple values per record but can only parse regular floating point values. Translate incoming *z*-values via the `incols` parameter.

- **binary** (*bool or str*) – **i**|**o**[*ncols*][*type*][**w**][**+l**|**b**]. Select native binary input (using `binary="i"`) or output (using `binary="o"`), where *ncols* is the number of data columns of *type*, which must be one of:

  - **c** - int8_t (1-byte signed char)
  - **u** - uint8_t (1-byte unsigned char)
  - **h** - int16_t (2-byte signed int)
  - **H** - uint16_t (2-byte unsigned int)
  - **i** - int32_t (4-byte signed int)
  - **I** - uint32_t (4-byte unsigned int)
  - **l** - int64_t (8-byte signed int)
  - **L** - uint64_t (8-byte unsigned int)
  - **f** - 4-byte single-precision float
  - **d** - 8-byte double-precision float
  - **x** - use to skip *ncols* anywhere in the record

  For records with mixed types, append additional comma-separated combinations of *ncols type* (no space). The following modifiers are supported:

  - **w** after any item to force byte-swapping.
  - **+l**|**b** to indicate that the entire data file should be read as little- or big-endian, respectively.

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#bi-full.

- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `d="-9999"` will replace all values equal to -9999 with NaN during input and all NaN values with -9999 during output. Prepend **i** to the *nodata* value for input columns only. Prepend **o** to the *nodata* value for output columns only.

- **find** (*str*) – [**~**]*"pattern"* | [**~**]/*regexp*/[**i**]. Only pass records that match the given *pattern* or regular expressions [Default processes all records]. Prepend **~** to the *pattern* or *regexp* to instead only pass data expressions that do not match the pattern. Append **i** for case insensitive matching. This does not apply to headers or segment headers.
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **header** (*str*) – [**i**|**o**][*n*][**+c**][**+d**][**+m***segheader*][**+r***remark*][**+t***title*]. Specify that input and/or output file(s) have *n* header records [Default is 0]. Prepend **i** if only the primary input should have header records. Prepend **o** to control the writing of header records, with the following modifiers supported:
  - **+d** to remove existing header records.
  - **+c** to add a header comment with column names to the output [Default is no column names].
  - **+m** to add a segment header *segheader* to the output after the header block [Default is no segment header].
  - **+r** to add a *remark* comment to the output [Default is no comment]. The *remark* string may contain \n to indicate line-breaks.
  - **+t** to add a *title* comment to the output [Default is no title]. The *title* string may contain \n to indicate line-breaks.

  Blank lines and lines starting with # are always skipped.
- **incols** (*str or 1d array*) – Specify data columns for primary input in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default reads all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in input order (e.g., `incols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `incols="0:2,4+l"` to input the first three columns followed by the log-transformed 5th column). To read from a given column until the end of the record, leave off *stop* when specifying the column range. To read trailing text, add the column **t**. Append the word number to **t** to ingest only a single word from the trailing text. Instead of specifying columns, use `incols="n"` to simply read numerical input and skip trailing text. Optionally, append one of the following modifiers to any column or column range to transform the input columns:
    - **+l** to take the *log10* of the input values.
    - **+d** to divide the input values by the factor *divisor* [Default is 1].
    - **+s** to multiple the input values by the factor *scale* [Default is 1].
    - **+o** to add the given *offset* to the input values [Default is 0].
- **registration** (*str*) – **g**|**p**. Force gridline (**g**) or pixel (**p**) node registration. [Default is **g**(ridline)].
- **wrap** (*str*) – **y**|**a**|**w**|**d**|**h**|**m**|**s**|**c***period*[/*phase*][**+c***col*]. Convert the input *x*-coordinate to a cyclical coordinate, or a different column if selected via **+c***col*. The following cyclical coordinate transformations are supported:
  - **y** - yearly cycle (normalized)

- **a** - annual cycle (monthly)
- **w** - weekly cycle (day)
- **d** - daily cycle (hour)
- **h** - hourly cycle (minute)
- **m** - minute cycle (second)
- **s** - second cycle (second)
- **c** - custom cycle (normalized)

Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#w-full.

Returns

ret (*xarray.DataArray or None*) – Return type depends on whether the `outgrid` parameter is set:

- `xarray.DataArray`: if `outgrid` is not set
- None if `outgrid` is set (grid output will be stored in file set by `outgrid`)

Operations on grids:

grd2xyz(grid[, output_type, outfile, ...])	Convert grid to data table.
grdclip(grid, *[, outgrid, region, above, ...])	Sets values in a grid that meet certain criteria to a new value.
grdcut(grid, *[, outgrid, region, ...])	Extract subregion from a grid.
grdfill(grid, *[, mode, outgrid, region, ...])	Fill blank areas from a grid file.
grdfilter(grid, *[, distance, filter, ...])	Filter a grid in the space (or time) domain.
grdgradient(grid, *[, azimuth, direction, ...])	Compute the directional derivative of the vector gradient of the data.
grdlandmask(*[, area_thresh, resolution, ...])	Create a grid file with set values for land and water.
grdproject(grid, *[, center, spacing, dpi, ...])	Change projection of gridded data between geographical and rectangular.
grdsample(grid, *[, outgrid, projection, ...])	Change the registration, spacing, or nodes in a grid file.
grdtrack(points, grid[, newcolname, ...])	Sample grids at specified (x,y) locations.
grdvolume(grid[, output_type, outfile, ...])	Determine the volume between the surface of a grid and a plane.

pygmt.grd2xyz

pygmt.**grd2xyz**(*grid*, *output_type='pandas'*, *outfile=None*, ***, *cstyle=None*, *region=None*, *verbose=None*, *weight=None*, *convention=None*, *binary=None*, *nodata=None*, *coltypes=None*, *header=None*, *outcols=None*, *skiprows=None*, ***kwargs*)

Convert grid to data table.

Read a grid and output xyz-triplets as a `numpy.ndarray`, `pandas.DataFrame`, or ASCII file.

Full option list at https://docs.generic-mapping-tools.org/latest/grd2xyz.html

Aliases:

- C = cstyle
- R = region
- V = verbose

- W = weight
- Z = convention
- b = binary
- d = nodata
- f = coltypes
- h = header
- o = outcols
- s = skiprows

Parameters

- **grid** (*str or xarray.DataArray*) – The file name of the input grid or the grid loaded as a `xarray.DataArray`. This is the only required parameter.
- **output_type** (*str*) – Determine the format the xyz data will be returned in [Default is `pandas`]:
  - `numpy` - `numpy.ndarray`
  - `pandas`- `pandas.DataFrame`
  - `file` - ASCII file (requires `outfile`)
- **outfile** (*str*) – The file name for the output ASCII file.
- **cstyle** (*str*) – [**f**|**i**]. Replace the x- and y-coordinates on output with the corresponding column and row numbers. These start at 0 (C-style counting); append **f** to start at 1 (Fortran-style counting). Alternatively, append **i** to write just the two columns *index* and *z*, where *index* is the 1-D indexing that GMT uses when referring to grid nodes.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest. Adding `region` will select a subsection of the grid. If this subsection exceeds the boundaries of the grid, only the common region will be output.
- **weight** (*str*) – [**a**[**+u***unit*]|*weight*]. Write out *x,y,z,w*, where *w* is the supplied *weight* (or 1 if not supplied) [Default writes *x,y,z* only]. Choose **a** to compute weights equal to the area each node represents. For Cartesian grids this is simply the product of the *x* and *y* increments (except for gridline-registered grids at all sides [half] and corners [quarter]). For geographic grids we default to a length unit of **k**. Change this by appending **+u***unit*. For such grids, the area varies with latitude and also sees special cases for gridline-registered layouts at sides, corners, and poles.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages

- **convention** (*str*) – [*flags*]. Write a 1-column ASCII [or binary] table. Output will be organized according to the specified ordering convention contained in *flags*. If data should be written by rows, make *flags* start with **T** (op) if first row is y = ymax or **B** (ottom) if first row is y = ymin. Then, append **L** or **R** to indicate that first element should start at left or right end of row. Likewise for column formats: start with **L** or **R** to position first column, and then append **T** or **B** to position first element in a row. For gridline registered grids: If grid is periodic in x but the written data should not contain the (redundant) column at x = xmax, append **x**. For grid periodic in y, skip writing the redundant row at y = ymax by appending **y**. If the byte-order needs to be swapped, append **w**. Select one of several data types (all binary except **a**):

  – **a** ASCII representation of a single item per record

  – **c** int8_t, signed 1-byte character

  – **u** uint8_t, unsigned 1-byte character

  – **h** int16_t, short 2-byte integer

  – **H** uint16_t, unsigned short 2-byte integer

  – **i** int32_t, 4-byte integer

  – **I** uint32_t, unsigned 4-byte integer

  – **l** int64_t, long (8-byte) integer

  – **L** uint64_t, unsigned long (8-byte) integer

  – **f** 4-byte floating point single precision

  – **d** 8-byte floating point double precision

  Default format is scanline orientation of ASCII numbers: **TLa**.

- **binary** (*bool or str*) – **i**|**o**[*ncols*][*type*][**w**][**+l**|**b**]. Select native binary input (using `binary="i"`) or output (using `binary="o"`), where *ncols* is the number of data columns of *type*, which must be one of:

  – **c** - int8_t (1-byte signed char)

  – **u** - uint8_t (1-byte unsigned char)

  – **h** - int16_t (2-byte signed int)

  – **H** - uint16_t (2-byte unsigned int)

  – **i** - int32_t (4-byte signed int)

  – **I** - uint32_t (4-byte unsigned int)

  – **l** - int64_t (8-byte signed int)

  – **L** - uint64_t (8-byte unsigned int)

  – **f** - 4-byte single-precision float

  – **d** - 8-byte double-precision float

  – **x** - use to skip *ncols* anywhere in the record

  For records with mixed types, append additional comma-separated combinations of *ncols* *type* (no space). The following modifiers are supported:

  – **w** after any item to force byte-swapping.

  – **+l**|**b** to indicate that the entire data file should be read as little- or big-endian, respectively.

Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#bi-full.

- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `d="-9999"` will replace all values equal to -9999 with NaN during input and all NaN values with -9999 during output. Prepend **i** to the *nodata* value for input columns only. Prepend **o** to the *nodata* value for output columns only.
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **header** (*str*) – [**i**|**o**][*n*][**+c**][**+d**][**+m***segheader*][**+r***remark*][**+t***title*]. Specify that input and/or output file(s) have *n* header records [Default is 0]. Prepend **i** if only the primary input should have header records. Prepend **o** to control the writing of header records, with the following modifiers supported:
  - **+d** to remove existing header records.
  - **+c** to add a header comment with column names to the output [Default is no column names].
  - **+m** to add a segment header *segheader* to the output after the header block [Default is no segment header].
  - **+r** to add a *remark* comment to the output [Default is no comment]. The *remark* string may contain \n to indicate line-breaks.
  - **+t** to add a *title* comment to the output [Default is no title]. The *title* string may contain \n to indicate line-breaks.

  Blank lines and lines starting with # are always skipped.
- **outcols** (*str or 1d array*) – *cols*[,...][,**t**[*word*]]. Specify data columns for primary output in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default writes all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in output order (e.g., `outcols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `outcols="0:2,4"` to output the first three columns followed by the 5th column). To write from a given column until the end of the record, leave off *stop* when specifying the column range. To write trailing text, add the column **t**. Append the word number to **t** to write only a single word from the trailing text. Instead of specifying columns, use `outcols="n"` to simply read numerical input and skip trailing text. Note: if `incols` is also used then the columns given to `outcols` correspond to the order after the `incols` selection has taken place.
- **skiprows** (*bool or str*) – [*cols*][**+a**][**+r**]. Suppress output for records whose *z*-value equals NaN [Default outputs all records]. Optionally, supply a comma-separated list of all columns or column ranges to consider for this NaN test [Default only considers the third data column (i.e., *cols = 2*)]. Column ranges must be given in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified. The following modifiers are supported:
  - **+r** to reverse the suppression, i.e., only output the records whose *z*-value equals NaN.
  - **+a** to suppress the output of the record if just one or more of the columns equal NaN [Default skips record only if values in all specified *cols* equal NaN].

Returns

ret (*pandas.DataFrame or numpy.ndarray or None*) – Return type depends on `outfile` and `output_type`:

- None if `outfile` is set (output will be stored in file set by `outfile`)
- `pandas.DataFrame` or `numpy.ndarray` if `outfile` is not set (depends on `output_type`)

pygmt.grdclip

`pygmt.`**`grdclip`**(*grid*, ***, *outgrid=None*, *region=None*, *above=None*, *below=None*, *between=None*, *new=None*, *verbose=None*, ***kwargs*)

Sets values in a grid that meet certain criteria to a new value.

Produce a clipped `outgrid` or `xarray.DataArray` version of the input `grid` file.

The parameters `above` and `below` allow for a given value to be set for values above or below a set amount, respectively. This allows for extreme values in a grid, such as points below a certain depth when plotting Earth relief, to all be set to the same value.

Full option list at https://docs.generic-mapping-tools.org/latest/grdclip.html

Aliases:

- G = outgrid
- R = region
- Sa = above
- Sb = below
- Si = between
- Sr = new
- V = verbose

Parameters

- **grid** (`str` *or* `xarray.DataArray`) – The file name of the input grid or the grid loaded as a DataArray.
- **outgrid** (`str` *or* `None`) – The name of the output netCDF file with extension .nc to store the grid in.
- **region** (`str` *or* `list`) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **above** (`str` *or* `list` *or* `tuple`) – [*high*, *above*]. Set all data[i] > *high* to *above*.
- **below** (`str` *or* `list` *or* `tuple`) – [*low*, *below*]. Set all data[i] < *low* to *below*.
- **between** (`str` *or* `list` *or* `tuple`) – [*low*, *high*, *between*]. Set all data[i] >= *low* and <= *high* to *between*.
- **new** (`str` *or* `list` *or* `tuple`) – [*old*, *new*]. Set all data[i] == *old* to *new*. This is mostly useful when your data are known to be integer values.
- **verbose** (`bool` *or* `str`) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only

- **w** - Warnings [Default]
- **t** - Timings (report runtimes for time-intensive algorithms);
- **i** - Informational messages (same as `verbose=True`)
- **c** - Compatibility warnings
- **d** - Debugging messages

Returns

ret (*xarray.DataArray or None*) – Return type depends on whether the `outgrid` parameter is set:

- `xarray.DataArray` if `outgrid` is not set
- None if `outgrid` is set (grid output will be stored in file set by `outgrid`)

Examples using `pygmt.grdclip`

- *Clipping grid values*

pygmt.grdcut

`pygmt.grdcut`(*grid*, ***, *outgrid=None*, *region=None*, *projection=None*, *extend=None*, *circ_subregion=None*, *verbose=None*, *z_subregion=None*, *coltypes=None*, ***kwargs*)

Extract subregion from a grid.

Produce a new `outgrid` file which is a subregion of `grid`. The subregion is specified with `region`; the specified range must not exceed the range of `grid` (but see `extend`). If in doubt, run *`pygmt.grdinfo`* to check range. Alternatively, define the subregion indirectly via a range check on the node values or via distances from a given point. Finally, you can give `projection` for oblique projections to determine the corresponding rectangular `region` that will give a grid that fully covers the oblique domain.

Full option list at https://docs.generic-mapping-tools.org/latest/grdcut.html

Aliases:

- G = outgrid
- J = projection
- N = extend
- R = region
- S = circ_subregion
- V = verbose
- Z = z_subregion
- f = coltypes

Parameters

- **grid** (`str or xarray.DataArray`) – The file name of the input grid or the grid loaded as a DataArray.
- **outgrid** (`str or None`) – The name of the output netCDF file with extension .nc to store the grid in.

- **projection** (*str*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **extend** (*bool or int or float*) – Allow grid to be extended if new `region` exceeds existing boundaries. Give a value to initialize nodes outside current region.
- **circ_subregion** (*str*) – *lon/lat/radius*[*unit*][**+n**]. Specify an origin (*lon* and *lat*) and *radius*; append a distance *unit* and we determine the corresponding rectangular region so that all grid nodes on or inside the circle are contained in the subset. If **+n** is appended we set all nodes outside the circle to NaN.
- **z_subregion** (*str*) – [*min/max*][**+n**|**N**|**r**]. Determine a new rectangular region so that all nodes outside this region are also outside the given z-range [-inf/+inf]. To indicate no limit on *min* or *max* only, specify a hyphen (-). Normally, any NaNs encountered are simply skipped and not considered in the range-decision. Append **+n** to consider a NaN to be outside the given z-range. This means the new subset will be NaN-free. Alternatively, append **+r** to consider NaNs to be within the data range. In this case we stop shrinking the boundaries once a NaN is found [Default simply skips NaNs when making the range decision]. Finally, if your core subset grid is surrounded by rows and/or columns that are all NaNs, append **+N** to strip off such columns before (optionally) considering the range of the core subset for further reduction of the area.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.

Returns

ret (*xarray.DataArray or None*) – Return type depends on whether the `outgrid` parameter is set:

- `xarray.DataArray` if `outgrid` is not set
- None if `outgrid` is set (grid output will be stored in file set by `outgrid`)

pygmt.grdfill

`pygmt.grdfill`(*grid*, ***, *mode=None*, *outgrid=None*, *region=None*, *verbose=None*, ***kwargs*)

Fill blank areas from a grid file.

Read a grid that presumably has unfilled holes that the user wants to fill in some fashion. Holes are identified by NaN values but this criteria can be changed. There are several different algorithms that can be used to replace the hole values.

Full option list at https://docs.generic-mapping-tools.org/latest/grdfill.html

Aliases:

- A = mode
- G = outgrid
- R = region
- V = verbose

Parameters

- **grid** (*str or xarray.DataArray*) – The file name of the input grid or the grid loaded as a DataArray.
- **outgrid** (*str or None*) – The name of the output netCDF file with extension .nc to store the grid in.
- **mode** (*str*) – Specify the hole-filling algorithm to use. Choose from **c** for constant fill and append the constant value, **n** for nearest neighbor (and optionally append a search radius in pixels [default radius is $r^2 = \sqrt{X^2 + Y^2}$, where (X, Y) are the node dimensions of the grid]), or **s** for bicubic spline (optionally append a *tension* parameter [Default is no tension]).
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages

Returns

ret (*xarray.DataArray or None*) – Return type depends on whether the `outgrid` parameter is set:

- `xarray.DataArray` if `outgrid` is not set
- None if `outgrid` is set (grid output will be stored in file set by `outgrid`)

pygmt.grdfilter

`pygmt.grdfilter`(*grid*, ***, *distance=None*, *filter=None*, *outgrid=None*, *spacing=None*, *nans=None*, *region=None*, *toggle=None*, *verbose=None*, *coltypes=None*, *registration=None*, ***kwargs*)

Filter a grid in the space (or time) domain.

Filter a grid file in the time domain using one of the selected convolution or non-convolution isotropic or rectangular filters and compute distances using Cartesian or Spherical geometries. The output grid file can optionally be generated as a sub-region of the input (via `region`) and/or with new increment (via `spacing`) or registration (via `toggle`). In this way, one may have "extra space" in the input data so that the edges will not be used and the output can be within one half-width of the input edges. If the filter is low-pass, then the output may be less frequently sampled than the input.

Full option list at https://docs.generic-mapping-tools.org/latest/grdfilter.html

Aliases:

- D = distance
- F = filter
- G = outgrid
- I = spacing
- N = nans
- R = region
- T = toggle
- V = verbose
- f = coltypes
- r = registration

Parameters

- **grid** (*str or xarray.DataArray*) – The file name of the input grid or the grid loaded as a DataArray.
- **outgrid** (*str or None*) – The name of the output netCDF file with extension .nc to store the grid in.
- **filter** (*str*) – **b|c|g|o|m|p|h***xwidth*[/*width2*][*modifiers*]. Name of filter type you which to apply, followed by the width:

  b: Box Car

  c: Cosine Arch

  g: Gaussian

  o: Operator

  m: Median

  p: Maximum Likelihood probability

  h: histogram
- **distance** (*str*) – Distance *flag* tells how grid (x,y) relates to filter width as follows:

  p: grid (px,py) with *width* an odd number of pixels; Cartesian distances.

  0: grid (x,y) same units as *width*, Cartesian distances.

1: grid (x,y) in degrees, *width* in kilometers, Cartesian distances.

2: grid (x,y) in degrees, *width* in km, dx scaled by cos(middle y), Cartesian distances.

The above options are fastest because they allow weight matrix to be computed only once. The next three options are slower because they recompute weights for each latitude.

3: grid (x,y) in degrees, *width* in km, dx scaled by cosine(y), Cartesian distance calculation.

4: grid (x,y) in degrees, *width* in km, Spherical distance calculation.

5: grid (x,y) in Mercator `projection='m1'` img units, *width* in km, Spherical distance calculation.

- **`spacing`** (*`str`*) – *xinc*[**+e**|**n**][/*yinc*[**+e**|**n**]]. *x_inc* [and optionally *y_inc*] is the grid spacing.
  - **Geographical (degrees) coordinates**: Optionally, append an increment unit. Choose among **m** to indicate arc minutes or **s** to indicate arc seconds. If one of the units **e**, **f**, **k**, **M**, **n** or **u** is appended instead, the increment is assumed to be given in meter, foot, km, mile, nautical mile or US survey foot, respectively, and will be converted to the equivalent degrees longitude at the middle latitude of the region (the conversion depends on PROJ_ELLIPSOID). If *y_inc* is given but set to 0 it will be reset equal to *x_inc*; otherwise it will be converted to degrees latitude.
  - **All coordinates**: If **+e** is appended then the corresponding max *x* (*east*) or *y* (*north*) may be slightly adjusted to fit exactly the given increment [by default the increment may be adjusted slightly to fit the given domain]. Finally, instead of giving an increment you may specify the *number of nodes* desired by appending **+n** to the supplied integer argument; the increment is then recalculated from the number of nodes, the *registration*, and the domain. The resulting increment value depends on whether you have selected a gridline-registered or pixel-registered grid; see GMT File Formats for details.

  Note: If `region=grdfile` is used then the grid spacing and the registration have already been initialized; use `spacing` and `registration` to override these values.
- **`nans`** (*`str or float`*) – **i**|**p**|**r**. Determine how NaN-values in the input grid affects the filtered output.
- **`region`** (*`str or list`*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **`toggle`** (*`bool`*) – Toggle the node registration for the output grid so as to become the opposite of the input grid. [Default gives the same registration as the input grid].
- **`verbose`** (*`bool or str`*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **`coltypes`** (*`str`*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.

- **registration** (*str*) – **g**|**p**. Force gridline (**g**) or pixel (**p**) node registration. [Default is **g**(ridline)].

Returns

ret (*xarray.DataArray or None*) – Return type depends on whether the `outgrid` parameter is set:

- `xarray.DataArray` if `outgrid` is not set
- None if `outgrid` is set (grid output will be stored in file set by `outgrid`)

Examples

```
>>> import os
>>> import pygmt
```

```
>>> # Apply a filter of 600km (full width) to the @earth_relief_30m file
>>> # and return a filtered field (saved as netcdf)
>>> pygmt.grdfilter(
...     grid="@earth_relief_30m",
...     filter="m600",
...     distance="4",
...     region=[150, 250, 10, 40],
...     spacing=0.5,
...     outgrid="filtered_pacific.nc",
... )
>>> os.remove("filtered_pacific.nc")  # cleanup file
```

```
>>> # Apply a gaussian smoothing filter of 600 km in the input data array,
>>> # and returns a filtered data array with the smoothed field.
>>> grid = pygmt.datasets.load_earth_relief()
>>> smooth_field = pygmt.grdfilter(grid=grid, filter="g600", distance="4")
```

pygmt.grdgradient

`pygmt.grdgradient`(*grid*, ***, *azimuth=None*, *direction=None*, *radiance=None*, *outgrid=None*, *normalize=None*, *tiles=None*, *region=None*, *slope_file=None*, *verbose=None*, *coltypes=None*, *interpolation=None*, ***kwargs*)

Compute the directional derivative of the vector gradient of the data.

Can accept `azimuth`, `direction`, and `radiance` input to create the resulting gradient.

Full option list at https://docs.generic-mapping-tools.org/latest/grdgradient.html

Aliases:

- A = azimuth
- D = direction
- E = radiance
- G = outgrid
- N = normalize
- Q = tiles

- R = region
- S = slope_file
- V = verbose
- f = coltypes
- n = interpolation

Parameters

- **grid** (*str or xarray.DataArray*) – The file name of the input grid or the grid loaded as a DataArray.
- **outgrid** (*str or None*) – The name of the output netCDF file with extension .nc to store the grid in.
- **azimuth** (*int or float or str or list*) – *azim*[/*azim2*]. Azimuthal direction for a directional derivative; *azim* is the angle in the x,y plane measured in degrees positive clockwise from north (the +y direction) toward east (the +x direction). The negative of the directional derivative, $-(\frac{dz}{dx}\sin(\text{azim}) + \frac{dz}{dy}\cos(\text{azim}))$, is found; negation yields positive values when the slope of $z(x, y)$ is downhill in the *azim* direction, the correct sense for shading the illumination of an image by a light source above the x,y plane shining from the *azim* direction. Optionally, supply two azimuths, *azim*/*azim2*, in which case the gradients in each of these directions are calculated and the one larger in magnitude is retained; this is useful for illuminating data with two directions of lineated structures, e.g., *0/270* illuminates from the north (top) and west (left). Finally, if *azim* is a file it must be a grid of the same domain, spacing and registration as *grid* that will update the azimuth at each output node when computing the directional derivatives.
- **direction** (*str*) – [**a**][**c**][**o**][**n**]. Find the direction of the positive (up-slope) gradient of the data. The following options are supported:
  - **a** - Find the aspect (i.e., the down-slope direction)
  - **c** - Use the conventional Cartesian angles measured counterclockwise from the positive x (east) direction.
  - **o** - Report orientations (0-180) rather than directions (0-360).
  - **n** - Add 90 degrees to all angles (e.g., to give local strikes of the surface).
- **radiance** (*str or list*) – [**m**|**s**|**p**]*azim/elev*[**+a***ambient*][**+d***diffuse*][**+p***specular*][**+s***shine*]. Compute Lambertian radiance appropriate to use with *pygmt.Figure.grdimage* and *pygmt.Figure.grdview*. The Lambertian Reflection assumes an ideal surface that reflects all the light that strikes it and the surface appears equally bright from all viewing directions. Here, *azim* and *elev* are the azimuth and elevation of the light vector. Optionally, supply *ambient* [0.55], *diffuse* [0.6], *specular* [0.4], or *shine* [10], which are parameters that control the reflectance properties of the surface. Default values are given in the brackets. Use **s** for a simpler Lambertian algorithm. Note that with this form you only have to provide azimuth and elevation. Alternatively, use **p** for the Peucker piecewise linear approximation (simpler but faster algorithm; in this case *azim* and *elev* are hardwired to 315 and 45 degrees. This means that even if you provide other values they will be ignored.).
- **normalize** (*str or bool*) – [**e**|**t**][*amp*][**+a***ambient*][**+s***sigma*][**+o***offset*]. The actual gradients g are offset and scaled to produce normalized gradients g_n with a maximum output magnitude of *amp*. If *amp* is not given, default *amp* = 1. If *offset* is not given, it is set to the average of g. The following forms are supported:
  - **True** - Normalize using $g_n = \text{amp}\ (\frac{g - \text{offset}}{max(|g - \text{offset}|)})$

  - **e** - Normalize using a cumulative Laplace distribution yielding: $g_n = \text{amp}(1 - \exp(\sqrt{2}\frac{g - \text{offset}}{\sigma}))$, where σ is estimated using the L1 norm of $(g - \text{offset})$ if it is not given.
  - **t** - Normalize using a cumulative Cauchy distribution yielding: $g_n = \frac{2(\text{amp})}{\pi}(\tan^{-1}(\frac{g - \text{offset}}{\sigma}))$ where σ is estimated using the L2 norm of $(g - \text{offset})$ if it is not given.

  As a final option, you may add **+a***ambient* to add *ambient* to all nodes after gradient calculations are completed.

- **tiles** (*str*) – **c**|**r**|**R**. Controls how normalization via `normalize` is carried out. When multiple grids should be normalized the same way (i.e., with the same *offset* and/or *sigma*), we must pass these values via `normalize`. However, this is inconvenient if we compute these values from a grid. Use **c** to save the results of *offset* and *sigma* to a statistics file; if grid output is not needed for this run then do not specify `outgrid`. For subsequent runs, just use **r** to read these values. Using **R** will read then delete the statistics file.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **slope_file** (*str*) – Name of output grid file with scalar magnitudes of gradient vectors. Requires `direction` but makes `outgrid` optional.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **interpolation** (*str*) – [**b**|**c**|**l**|**n**][**+a**][**+b***BC*][**+c**][**+t***threshold*]. Select interpolation mode for grids. You can select the type of spline used:
  - **b** for B-spline
  - **c** for bicubic [Default]
  - **l** for bilinear
  - **n** for nearest-neighbor

Returns

ret (*xarray.DataArray or None*) – Return type depends on whether the `outgrid` parameter is set:

- `xarray.DataArray` if `outgrid` is not set
- None if `outgrid` is set (grid output will be stored in file set by `outgrid`)

Examples using `pygmt.grdgradient`

- *Calculating grid gradient and radiance*

pygmt.grdlandmask

`pygmt.grdlandmask`(***, *area_thresh=None*, *resolution=None*, *bordervalues=None*, *outgrid=None*, *spacing=None*, *maskvalues=None*, *region=None*, *verbose=None*, *registration=None*, ***kwargs*)

Create a grid file with set values for land and water.

Read the selected shoreline database and create a grid to specify which nodes in the specified grid are over land or over water. The nodes defined by the selected region and lattice spacing will be set according to one of two criteria: (1) land vs water, or (2) the more detailed (hierarchical) ocean vs land vs lake vs island vs pond.

Full option list at https://docs.generic-mapping-tools.org/latest/grdlandmask.html

Aliases:

- A = area_thresh
- D = resolution
- E = bordervalues
- G = outgrid
- I = spacing
- N = maskvalues
- R = region
- V = verbose
- r = registration

Parameters

- **outgrid** (*str or None*) – The name of the output netCDF file with extension .nc to store the grid in.
- **spacing** (*str*) – *xinc*[**+e**|**n**][/*yinc*[**+e**|**n**]]. *x_inc* [and optionally *y_inc*] is the grid spacing.
  - **Geographical (degrees) coordinates**: Optionally, append an increment unit. Choose among **m** to indicate arc minutes or **s** to indicate arc seconds. If one of the units **e**, **f**, **k**, **M**, **n** or **u** is appended instead, the increment is assumed to be given in meter, foot, km, mile, nautical mile or US survey foot, respectively, and will be converted to the equivalent degrees longitude at the middle latitude of the region (the conversion depends on PROJ_ELLIPSOID). If *y_inc* is given but set to 0 it will be reset equal to *x_inc*; otherwise it will be converted to degrees latitude.
  - **All coordinates**: If **+e** is appended then the corresponding max *x* (*east*) or *y* (*north*) may be slightly adjusted to fit exactly the given increment [by default the increment may be adjusted slightly to fit the given domain]. Finally, instead of giving an increment you may specify the *number of nodes* desired by appending **+n** to the supplied integer argument; the increment is then recalculated from the number of nodes, the *registration*, and the domain. The resulting increment value depends on whether you have selected a gridline-registered or pixel-registered grid; see GMT File Formats for details.

  Note: If `region=grdfile` is used then the grid spacing and the registration have already been initialized; use `spacing` and `registration` to override these values.

- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **area_thresh** (*int or float or str*) – *min_area*[/*min_level*/*max_level*][**+a**[**g**|**i**][**s**|**S**]][**+l**|**r**][**+p***percent*]. Features with an area smaller than *min_area* in km^2 or of hierarchical level that is lower than *min_level* or higher than *max_level* will not be plotted [Default is 0/0/4 (all features)].
- **resolution** (*str*) – *res*[**+f**]. Selects the resolution of the data set to use ((**f**)ull, (**h**)igh, (**i**)ntermediate, (**l**)ow, or (**c**)rude). The resolution drops off by ~80% between data sets. [Default is **l**]. Append **+f** to automatically select a lower resolution should the one requested not be available [abort if not found]. Alternatively, choose (**a**)uto to automatically select the best resolution given the chosen region. Note that because the coastlines differ in details a node in a mask file using one resolution is not guaranteed to remain inside [or outside] when a different resolution is selected.
- **bordervalues** (*bool or str or float or list*) – Nodes that fall exactly on a polygon boundary should be considered to be outside the polygon [Default considers them to be inside]. Alternatively, append either a list of four values [*cborder*, *lborder*, *iborder*, *pborder*] or just the single value *bordervalue* (for the case when they should all be the same value). This turns on the line-tracking mode. Now, after setting the mask values specified via `maskvalues` we trace the lines and change the node values for all cells traversed by a line to the corresponding border value. Here, *cborder* is used for cells traversed by the coastline, *lborder* for cells traversed by a lake outline, *iborder* for islands-in-lakes outlines, and *pborder* for ponds-in-islands-in-lakes outlines [Default is no line tracing].
- **maskvalues** (*str or list*) – [*wet*, *dry*] or [*ocean*, *land*, *lake*, *island*, *pond*]. Sets the values that will be assigned to nodes. Values can be any number, including the textstring NaN [Default is [0, 1, 0, 1, 0] (i.e., [0, 1])]. Also select `bordervalues` to let nodes exactly on feature boundaries be considered outside [Default is inside].
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **registration** (*str*) – **g**|**p**. Force gridline (**g**) or pixel (**p**) node registration. [Default is **g**(ridline)].

Returns

ret (*xarray.DataArray or None*) – Return type depends on whether the `outgrid` parameter is set:

- `xarray.DataArray` if `outgrid` is not set
- None if `outgrid` is set (grid output will be stored in file set by `outgrid`)

Examples using `pygmt.grdlandmask`

- *Create 'wet-dry' mask grid*

pygmt.grdproject

`pygmt.grdproject`(*grid*, ***, *center=None*, *spacing=None*, *dpi=None*, *scaling=None*, *outgrid=None*, *projection=None*, *inverse=None*, *unit=None*, *region=None*, *verbose=None*, *interpolation=None*, *registration=None*, ***kwargs*)

Change projection of gridded data between geographical and rectangular.

This module will project a geographical gridded data set onto a rectangular grid. If `inverse` is `True`, it will project a rectangular coordinate system to a geographic system. To obtain the value at each new node, its location is inversely projected back onto the input grid after which a value is interpolated between the surrounding input grid values. By default bi-cubic interpolation is used. Aliasing is avoided by also forward projecting the input grid nodes. If two or more nodes are projected onto the same new node, their average will dominate in the calculation of the new node value. Interpolation and aliasing is controlled with the `interpolation` option. The new node spacing may be determined in one of several ways by specifying the grid spacing, number of nodes, or resolution. Nodes not constrained by input data are set to NaN. The `region` parameter can be used to select a map region larger or smaller than that implied by the extent of the grid file.

Aliases:

- C = center
- D = spacing
- E = dpi
- F = scaling
- G = outgrid
- I = inverse
- J = projection
- M = unit
- R = region
- V = verbose
- n = interpolation
- r = registration

Parameters

- **grid** (*str or xarray.DataArray*) – The file name of the input grid or the grid loaded as a DataArray.
- **outgrid** (*str or None*) – The name of the output netCDF file with extension .nc to store the grid in.
- **inverse** (*bool*) – When set to `True` transforms grid from rectangular to geographical [Default is False].
- **projection** (*str*) – *projcode*[*projparams*/]*width*. Select map *projection*.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.

- **center** (*str or list*) – [*dx*, *dy*]. Let projected coordinates be relative to projection center [Default is relative to lower left corner]. Optionally, add offsets in the projected units to be added (or subtracted when `inverse` is set) to (from) the projected coordinates, such as false eastings and northings for particular projection zones [0/0].
- **spacing** (*str*) – *xinc*[**+e**|**n**][/*yinc*[**+e**|**n**]]. *x_inc* [and optionally *y_inc*] is the grid spacing.
  - **Geographical (degrees) coordinates**: Optionally, append an increment unit. Choose among **m** to indicate arc minutes or **s** to indicate arc seconds. If one of the units **e**, **f**, **k**, **M**, **n** or **u** is appended instead, the increment is assumed to be given in meter, foot, km, mile, nautical mile or US survey foot, respectively, and will be converted to the equivalent degrees longitude at the middle latitude of the region (the conversion depends on PROJ_ELLIPSOID). If *y_inc* is given but set to 0 it will be reset equal to *x_inc*; otherwise it will be converted to degrees latitude.
  - **All coordinates**: If **+e** is appended then the corresponding max *x* (*east*) or *y* (*north*) may be slightly adjusted to fit exactly the given increment [by default the increment may be adjusted slightly to fit the given domain]. Finally, instead of giving an increment you may specify the *number of nodes* desired by appending **+n** to the supplied integer argument; the increment is then recalculated from the number of nodes, the *registration*, and the domain. The resulting increment value depends on whether you have selected a gridline-registered or pixel-registered grid; see GMT File Formats for details.

  Note: If `region=grdfile` is used then the grid spacing and the registration have already been initialized; use `spacing` and `registration` to override these values.
- **dpi** (*int*) – Set the resolution for the new grid in dots per inch.
- **scaling** (*str*) – [**c**|**i**|**p**|**e**|**f**|**k**|**M**|**n**|**u**]. Force 1:1 scaling, i.e., output or output data are in actual projected meters [**e**]. To specify other units, append **f** (foot), **k** (km), **M** (statute mile), **n** (nautical mile), **u** (US survey foot), **i** (inch), **c** (cm), or **p** (point).
- **unit** (*str*) – Append **c**, **i**, or **p** to indicate that cm, inch, or point should be the projected measure unit. Cannot be used with `scaling`.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **interpolation** (*str*) – [**b**|**c**|**l**|**n**][**+a**][**+b***BC*][**+c**][**+t***threshold*]. Select interpolation mode for grids. You can select the type of spline used:
  - **b** for B-spline
  - **c** for bicubic [Default]
  - **l** for bilinear
  - **n** for nearest-neighbor

- **registration** (*str*) – **g**|**p**. Force gridline (**g**) or pixel (**p**) node registration. [Default is **g**(ridline)].

Returns

ret (*xarray.DataArray or None*) – Return type depends on whether the `outgrid` parameter is set:

- `xarray.DataArray` if `outgrid` is not set
- None if `outgrid` is set (grid output will be stored in file set by `outgrid`)

pygmt.grdsample

`pygmt.grdsample`(*grid*, ***, *outgrid=None*, *projection=None*, *spacing=None*, *region=None*, *translate=None*, *verbose=None*, *coltypes=None*, *interpolation=None*, *registration=None*, *cores=None*, ***kwargs*)

Change the registration, spacing, or nodes in a grid file.

This reads a grid file and interpolates it to create a new grid file. It can change the registration with `translate` or `registration`, change the grid-spacing or number of nodes with `spacing`, and set a new sub-region using `region`. A bicubic [Default], bilinear, B-spline or nearest-neighbor interpolation is set with `interpolation`.

When `region` is omitted, the output grid will cover the same region as the input grid. When `spacing` is omitted, the grid spacing of the output grid will be the same as the input grid. Either `registration` or `translate` can be used to change the grid registration. When omitted, the output grid will have the same registration as the input grid.

Aliases:

- G = outgrid
- I = spacing
- J = projection
- R = region
- T = translate
- V = verbose
- f = coltypes
- n = interpolation
- r = registration
- x = cores

Parameters

- **grid** (*str or xarray.DataArray*) – The file name of the input grid or the grid loaded as a DataArray.
- **outgrid** (*str or None*) – The name of the output netCDF file with extension .nc to store the grid in.
- **spacing** (*str*) – *xinc*[+**e**|**n**][/*yinc*[+**e**|**n**]]. *x_inc* [and optionally *y_inc*] is the grid spacing.
  - **Geographical (degrees) coordinates**: Optionally, append an increment unit. Choose among **m** to indicate arc minutes or **s** to indicate arc seconds. If one of the units **e**, **f**, **k**, **M**, **n** or **u** is appended instead, the increment is assumed to be given in meter, foot, km,

mile, nautical mile or US survey foot, respectively, and will be converted to the equivalent degrees longitude at the middle latitude of the region (the conversion depends on PROJ_ELLIPSOID). If *y_inc* is given but set to 0 it will be reset equal to *x_inc*; otherwise it will be converted to degrees latitude.

  - **All coordinates**: If **+e** is appended then the corresponding max *x* (*east*) or *y* (*north*) may be slightly adjusted to fit exactly the given increment [by default the increment may be adjusted slightly to fit the given domain]. Finally, instead of giving an increment you may specify the *number of nodes* desired by appending **+n** to the supplied integer argument; the increment is then recalculated from the number of nodes, the *registration*, and the domain. The resulting increment value depends on whether you have selected a gridline-registered or pixel-registered grid; see GMT File Formats for details.

  Note: If `region=grdfile` is used then the grid spacing and the registration have already been initialized; use `spacing` and `registration` to override these values.

- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **translate** (*bool*) – Translate between grid and pixel registration; if the input is grid-registered, the output will be pixel-registered and vice-versa.
- **registration** (*str or bool*) – [**g**|**p**]. Set registration to **g**ridline or **p**ixel.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **interpolation** (*str*) – [**b**|**c**|**l**|**n**][**+a**][**+b***BC*][**+c**][**+t***threshold*]. Select interpolation mode for grids. You can select the type of spline used:
  - **b** for B-spline
  - **c** for bicubic [Default]
  - **l** for bilinear
  - **n** for nearest-neighbor
- **cores** (*bool or int*) – [[**-**]*n*]. Limit the number of cores to be used in any OpenMP-enabled multi-threaded algorithms. By default we try to use all available cores. Set a number *n* to only use n cores (if too large it will be truncated to the maximum cores available). Finally, give a negative number *-n* to select (all - *n*) cores (or at least 1 if *n* equals or exceeds all).

Returns

ret (*xarray.DataArray or None*) – Return type depends on whether the `outgrid` parameter is set:

- `xarray.DataArray` if `outgrid` is not set
- None if `outgrid` is set (grid output will be stored in file set by `outgrid`)

pygmt.grdtrack

`pygmt.grdtrack`(*points*, *grid*, *newcolname=None*, *outfile=None*, ***, *resample=None*, *crossprofile=None*, *dfile=None*, *profile=None*, *critical=None*, *region=None*, *no_skip=None*, *stack=None*, *radius=None*, *verbose=None*, *z_only=None*, *aspatial=None*, *binary=None*, *nodata=None*, *find=None*, *coltypes=None*, *gap=None*, *header=None*, *incols=None*, *distcalc=None*, *interpolation=None*, *outcols=None*, *skiprows=None*, *wrap=None*, ***kwargs*)

Sample grids at specified (x,y) locations.

Reads one or more grid files and a table (from file or an array input; but see `profile` for exception) with (x,y) [or (lon,lat)] positions in the first two columns (more columns may be present). It interpolates the grid(s) at the positions in the table and writes out the table with the interpolated values added as (one or more) new columns. Alternatively (`crossprofile`), the input is considered to be line-segments and we create orthogonal cross-profiles at each data point or with an equidistant separation and sample the grid(s) along these profiles. A bicubic [Default], bilinear, B-spline or nearest-neighbor interpolation is used, requiring boundary conditions at the limits of the region (see `interpolation`; Default uses "natural" conditions (second partial derivative normal to edge is zero) unless the grid is automatically recognized as periodic.)

Full option list at https://docs.generic-mapping-tools.org/latest/grdtrack.html

Aliases:

- A = resample
- C = crossprofile
- D = dfile
- E = profile
- F = critical
- N = no_skip
- R = region
- S = stack
- T = radius
- V = verbose
- Z = z_only
- a = aspatial
- b = binary
- d = nodata
- e = find
- f = coltypes
- g = gap
- h = header
- i = incols
- j = distcalc

- n = interpolation
- o = outcols
- s = skiprows
- w = wrap

Parameters

- **points** (*str or numpy.ndarray or pandas.DataFrame or xarray.Dataset or geopandas.GeoDataFrame*) – Pass in either a file name to an ASCII data table, a 2D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **grid** (*xarray.DataArray or str*) – Gridded array from which to sample values from, or a filename (netcdf format).
- **newcolname** (*str*) – Required if `points` is a `pandas.DataFrame`. The name for the new column in the track `pandas.DataFrame` table where the sampled values will be placed.
- **outfile** (*str*) – The file name for the output ASCII file.
- **resample** (*str*) – **f**|**p**|**m**|**r**|**R**[**+l**] For track resampling (if `crossprofile` or `profile` are set) we can select how this is to be performed. Append **f** to keep original points, but add intermediate points if needed [Default], **m** as **f**, but first follow meridian (along y) then parallel (along x), **p** as **f**, but first follow parallel (along y) then meridian (along x), **r** to resample at equidistant locations; input points are not necessarily included in the output, and **R** as **r**, but adjust given spacing to fit the track length exactly. Finally, append **+l** if geographic distances should be measured along rhumb lines (loxodromes) instead of great circles. Ignored unless `crossprofile` is used.
- **crossprofile** (*str*) – *length*/*ds*[/*spacing*][**+a**|**+v**][**l**|**r**]. Use input line segments to create an equidistant and (optionally) equally-spaced set of crossing profiles along which we sample the grid(s) [Default simply samples the grid(s) at the input locations]. Specify two length scales that control how the sampling is done: *length* sets the full length of each cross-profile, while *ds* is the sampling spacing along each cross-profile. Optionally, append /*spacing* for an equidistant spacing between cross-profiles [Default erects cross-profiles at the input coordinates]; see `resample` for how resampling the input track is controlled. By default, all cross-profiles have the same direction (left to right as we look in the direction of the input line segment). Append **+a** to alternate the direction of cross-profiles, or **v** to enforce either a "west-to-east" or "south-to-north" view. By default the entire profiles are output. Choose to only output the left or right halves of the profiles by appending **+l** or **+r**, respectively. Append suitable units to *length*; it sets the unit used for *ds* [and *spacing*] (See Units). The default unit for geographic grids is meter while Cartesian grids implies the user unit. The output columns will be *lon*, *lat*, *dist*, *azimuth*, *z1*, *z2*, . . . , *zn* (The *zi* are the sampled values for each of the *n* grids).
- **dfile** (*str*) – In concert with `crossprofile` we can save the (possibly resampled) original lines to *dfile* [Default only saves the cross-profiles]. The columns will be *lon*, *lat*, *dist*, *azimuth*, *z1*, *z2*, . . . (sampled value for each grid).
- **profile** (*str*) – *line*[,*line*,. . .][**+a***az*][**+c**][**+d**][**+g**][**+i***inc*][**+l***length*][**+n***np*][**+o***az*][**+r***radius*]. Instead of reading input track coordinates, specify profiles via coordinates and modifiers. The format of each *line* is *start*/*stop*, where *start* or *stop* are either *lon*/*lat* (*x*/*y* for Cartesian data) or a 2-character XY key that uses the text-style justification format to specify a point on the map as [LCR][BMT]. Each line will be a separate segment unless **+c** is used which will connect segments with shared joints into a single segment. In addition to line coordinates, you can use Z-, Z+ to mean the global minimum and maximum locations in

the grid (only available if a single grid is given via **outfile**). You may append **+i***inc* to set the sampling interval; if not given then we default to half the minimum grid interval. For a *line* along parallels or meridians you can add **+g** to report degrees of longitude or latitude instead of great circle distances starting at zero. Instead of two coordinates you can specify an origin and one of **+a**, **+o**, or **+r**. The **+a** sets the azimuth of a profile of given length starting at the given origin, while **+o** centers the profile on the origin; both require **+l**. For circular sampling specify **+r** to define a circle of given radius centered on the origin; this option requires either **+n** or **+i**. The **+n***np* modifier sets the desired number of points, while **+l***length* gives the total length of the profile. Append **+d** to output the along-track distances after the coordinates. **Note**: No track file will be read. Also note that only one distance unit can be chosen. Giving different units will result in an error. If no units are specified we default to great circle distances in km (if geographic). If working with geographic data you can use `distcalc` to control distance calculation mode [Default is Great Circle]. **Note**: If `crossprofile` is set and *spacing* is given then that sampling scheme overrules any modifier set in `profile`.

- **critical** (*str*) – [**+b**][**+n**][**+r**][**+z***z0*]. Find critical points along each cross-profile as a function of along-track distance. Requires `crossprofile` and a single input grid (*z*). We examine each cross-profile generated and report (*dist*, *lonc*, *latc*, *distc*, *azimuthc*, *zc*) at the center peak of maximum *z* value, (*lonl*, *latl*, *distl*) and (*lonr*, *latr*, *distr*) at the first and last non-NaN point whose *z*-value exceeds *z0*, respectively, and the *width* based on the two extreme points found. Here, *dist* is the distance along the original input `points` and the other 12 output columns are a function of that distance. When searching for the center peak and the extreme first and last values that exceed the threshold we assume the profile is positive up. If we instead are looking for a trough then you must use **+n** to temporarily flip the profile to positive. The threshold *z0* value is always given as >= 0; use **+z** to change it [Default is 0]. Alternatively, use **+b** to determine the balance point and standard deviation of the profile; this is the weighted mean and weighted standard deviation of the distances, with *z* acting as the weight. Finally, use **+r** to obtain the weighted rms about the cross-track center (*distc* == 0). **Note**: We round the exact results to the nearest distance nodes along the cross-profiles. We write 13 output columns per track: *dist, lonc, latc, distc, azimuthc, zc, lonl, latl, distl, lonr, latr, distr, width.*
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **no_skip** (*bool*) – Do *not* skip points that fall outside the domain of the grid(s) [Default only output points within grid domain].
- **stack** (*str or list*) – *method/modifiers*. In conjunction with `crossprofile`, compute a single stacked profile from all profiles across each segment. Choose how stacking should be computed [Default method is **a**]:

  – **a** = mean (average)

  – **m** = median

  – **p** = mode (maximum likelihood)

  – **l** = lower

  – **L** = lower but only consider positive values

  – **u** = upper

  – **U** = upper but only consider negative values.

  The *modifiers* control the output; choose one or more among these choices:

  – **+a** : Append stacked values to all cross-profiles.

- **+d** : Append stack deviations to all cross-profiles.
- **+r** : Append data residuals (data - stack) to all cross-profiles.
- **+s**[*file*] : Save stacked profile to *file* [Default filename is grdtrack_stacked_profile.txt].
- **+c***fact* : Compute envelope on stacked profile as ±*fact* **deviation* [Default fact value is 2].

Notes:

1. Deviations depend on *method* and are st.dev (**a**), L1 scale, i.e., 1.4826 * median absolute deviation (MAD) (for **m** and **p**), or half-range (upper-lower)/2.
2. The stacked profile file contains a leading column plus groups of 4-6 columns, with one group for each sampled grid. The leading column holds cross distance, while the first four columns in a group hold stacked value, deviation, min value, and max value, respectively. If *method* is one of **a**|**m**|**p** then we also write the lower and upper confidence bounds (see **+c**). When one or more of **+a**, **+d**, and **+r** are used then we also append the stacking results to the end of each row, for all cross-profiles. The order is always stacked value (**+a**), followed by deviations (**+d**) and finally residuals (**+r**). When more than one grid is sampled this sequence of 1-3 columns is repeated for each grid.

- **radius** (*bool or int or float or str*) – [*radius*][**+e**|**p**]. To be used with normal grid sampling, and limited to a single, non-IMG grid. If the nearest node to the input point is NaN, search outwards until we find the nearest non-NaN node and report that value instead. Optionally specify a search radius which limits the consideration to points within this distance from the input point. To report the location of the nearest node and its distance from the input point, append **+e**. The default unit for geographic grid distances is spherical degrees. Use *radius* to change the unit and give *radius* = 0 if you do not want to limit the radius search. To instead replace the input point with the coordinates of the nearest node, append **+p**.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **z_only** (*bool*) – Only write out the sampled z-values [Default writes all columns].
- **aspatial** (*bool or str*) – [*col*=]*name*[,. . .]. Control how aspatial data are handled during input and output. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#aspatial-full.
- **binary** (*bool or str*) – **i**|**o**[*ncols*][*type*][**w**][**+l**|**b**]. Select native binary input (using `binary="i"`) or output (using `binary="o"`), where *ncols* is the number of data columns of *type*, which must be one of:
  - **c** - int8_t (1-byte signed char)
  - **u** - uint8_t (1-byte unsigned char)
  - **h** - int16_t (2-byte signed int)

  - **H** - uint16_t (2-byte unsigned int)
  - **i** - int32_t (4-byte signed int)
  - **I** - uint32_t (4-byte unsigned int)
  - **l** - int64_t (8-byte signed int)
  - **L** - uint64_t (8-byte unsigned int)
  - **f** - 4-byte single-precision float
  - **d** - 8-byte double-precision float
  - **x** - use to skip *ncols* anywhere in the record

  For records with mixed types, append additional comma-separated combinations of *ncols* *type* (no space). The following modifiers are supported:

  - **w** after any item to force byte-swapping.
  - **+l|b** to indicate that the entire data file should be read as little- or big-endian, respectively.

  Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#bi-full.

- **nodata** (*str*) – **i|o***nodata*. Substitute specific values with NaN (for tabular data). For example, `d="-9999"` will replace all values equal to -9999 with NaN during input and all NaN values with -9999 during output. Prepend **i** to the *nodata* value for input columns only. Prepend **o** to the *nodata* value for output columns only.
- **find** (*str*) – [**~**]*"pattern"* | [**~**]/*regexp*/[**i**]. Only pass records that match the given *pattern* or regular expressions [Default processes all records]. Prepend **~** to the *pattern* or *regexp* to instead only pass data expressions that do not match the pattern. Append **i** for case insensitive matching. This does not apply to headers or segment headers.
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **gap** (*str or list*) – [**a**]**x**|**y**|**d**|**X**|**Y**|**D**|[*col*]**z***gap*[**+n**|**p**]. Examine the spacing between consecutive data points in order to impose breaks in the line. To specify multiple criteria, provide a list with each item containing a string describing one set of criteria. Prepend **a** to specify that all the criteria must be met [Default is to impose breaks if any criteria are met]. The following modifiers are supported:

  - **x**|**X** - define a gap when there is a large enough change in the x coordinates (upper case to use projected coordinates).
  - **y**|**Y** - define a gap when there is a large enough change in the y coordinates (upper case to use projected coordinates).
  - **d**|**D** - define a gap when there is a large enough distance between coordinates (upper case to use projected coordinates).
  - [*col*]**z** - define a gap when there is a large enough change in the data in column *col* [default *col* is 2 (i.e., 3rd column)].

  A unit **u** may be appended to the specified *gap*:

  - For geographic data (**x**|**y**|**d**), the unit may be arc **d**(egree), **m**(inute), and **s**(econd), or (m)**e**(ter), **f**(eet), **k**(ilometer), **M**(iles), or **n**(autical miles) [Default is (m)**e**(ter)].
  - For projected data (**X**|**Y**|**D**), the unit may be **i**(nch), **c**(entimeter), or **p**(oint).

One of the following modifiers can be appended to *gap* [Default imposes breaks based on the absolute value of the difference between the current and previous value]:

- **+n** - specify that the previous value minus the current column value must exceed *gap* for a break to be imposed.
- **+p** - specify that the current value minus the previous value must exceed *gap* for a break to be imposed.

- `header` (*str*) – [**i**|**o**][*n*][**+c**][**+d**][**+m***segheader*][**+r***remark*][**+t***title*]. Specify that input and/or output file(s) have *n* header records [Default is 0]. Prepend **i** if only the primary input should have header records. Prepend **o** to control the writing of header records, with the following modifiers supported:
  - **+d** to remove existing header records.
  - **+c** to add a header comment with column names to the output [Default is no column names].
  - **+m** to add a segment header *segheader* to the output after the header block [Default is no segment header].
  - **+r** to add a *remark* comment to the output [Default is no comment]. The *remark* string may contain \n to indicate line-breaks.
  - **+t** to add a *title* comment to the output [Default is no title]. The *title* string may contain \n to indicate line-breaks.

  Blank lines and lines starting with # are always skipped.
- `incols` (*str or 1d array*) – Specify data columns for primary input in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default reads all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in input order (e.g., `incols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `incols="0:2,4+l"` to input the first three columns followed by the log-transformed 5th column). To read from a given column until the end of the record, leave off *stop* when specifying the column range. To read trailing text, add the column **t**. Append the word number to **t** to ingest only a single word from the trailing text. Instead of specifying columns, use `incols="n"` to simply read numerical input and skip trailing text. Optionally, append one of the following modifiers to any column or column range to transform the input columns:
    - **+l** to take the *log10* of the input values.
    - **+d** to divide the input values by the factor *divisor* [Default is 1].
    - **+s** to multiple the input values by the factor *scale* [Default is 1].
    - **+o** to add the given *offset* to the input values [Default is 0].
- `distcalc` (*str*) – **e**|**f**|**g**. Determine how spherical distances are calculated.
  - **e** - Ellipsoidal (or geodesic) mode
  - **f** - Flat Earth mode
  - **g** - Great circle distance [Default]

All spherical distance calculations depend on the current ellipsoid (PROJ_ELLIPSOID), the definition of the mean radius (PROJ_MEAN_RADIUS), and the specification of latitude type (PROJ_AUX_LATITUDE). Geodesic distance calculations is also controlled by method (PROJ_GEODESIC).

- **interpolation** (*str*) – [**b**|**c**|**l**|**n**][**+a**][**+b***BC*][**+c**][**+t***threshold*]. Select interpolation mode for grids. You can select the type of spline used:
  - **b** for B-spline
  - **c** for bicubic [Default]
  - **l** for bilinear
  - **n** for nearest-neighbor
- **outcols** (*str or 1d array*) – *cols*[,...][,**t**[*word*]]. Specify data columns for primary output in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default writes all columns in order, starting with the first (i.e., column 0)].
  - For *1d array*: specify individual columns in output order (e.g., `outcols=[1,0]` for the 2nd column followed by the 1st column).
  - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `outcols="0:2,4"` to output the first three columns followed by the 5th column). To write from a given column until the end of the record, leave off *stop* when specifying the column range. To write trailing text, add the column **t**. Append the word number to **t** to write only a single word from the trailing text. Instead of specifying columns, use `outcols="n"` to simply read numerical input and skip trailing text. Note: if `incols` is also used then the columns given to `outcols` correspond to the order after the `incols` selection has taken place.
- **skiprows** (*bool or str*) – [*cols*][**+a**][**+r**]. Suppress output for records whose *z*-value equals NaN [Default outputs all records]. Optionally, supply a comma-separated list of all columns or column ranges to consider for this NaN test [Default only considers the third data column (i.e., *cols = 2*)]. Column ranges must be given in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified. The following modifiers are supported:
  - **+r** to reverse the suppression, i.e., only output the records whose *z*-value equals NaN.
  - **+a** to suppress the output of the record if just one or more of the columns equal NaN [Default skips record only if values in all specified *cols* equal NaN].
- **wrap** (*str*) – **y**|**a**|**w**|**d**|**h**|**m**|**s**|**c***period*[/*phase*][**+c***col*]. Convert the input *x*-coordinate to a cyclical coordinate, or a different column if selected via **+c***col*. The following cyclical coordinate transformations are supported:
  - **y** - yearly cycle (normalized)
  - **a** - annual cycle (monthly)
  - **w** - weekly cycle (day)
  - **d** - daily cycle (hour)
  - **h** - hourly cycle (minute)
  - **m** - minute cycle (second)
  - **s** - second cycle (second)
  - **c** - custom cycle (normalized)

Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#w-full.

Returns

track (*pandas.DataFrame or None*) – Return type depends on whether the `outfile` parameter is set:

- `pandas.DataFrame` table with (x, y, ..., newcolname) if `outfile` is not set
- None if `outfile` is set (track output will be stored in file set by `outfile`)

Examples using `pygmt.grdtrack`

- *Sampling along tracks*

pygmt.grdvolume

`pygmt.`**`grdvolume`**(*grid*, *output_type='pandas'*, *outfile=None*, ***, *contour=None*, *region=None*, *unit=None*, *verbose=None*, ***kwargs*)

Determine the volume between the surface of a grid and a plane.

Read a 2-D grid file and calculate the volume contained below the surface and above the plane specified by the given contour (or zero if not given) and return the contour, area, volume, and maximum mean height (volume/area). Alternatively, a range of contours can be specified to return the volume and area inside the contour for all contour values.

Full option list at https://docs.generic-mapping-tools.org/latest/grdvolume.html

Aliases:

- C = contour
- R = region
- S = unit
- V = verbose

Parameters

- **grid** (*str or xarray.DataArray*) – The file name of the input grid or the grid loaded as a DataArray.
- **output_type** (*str*) – Determine the format the output data will be returned in [Default is `pandas`]:
  - `numpy` - `numpy.ndarray`
  - `pandas`- `pandas.DataFrame`
  - `file` - ASCII file (requires `outfile`)
- **outfile** (*str*) – The file name for the output ASCII file.
- **contour** (*str or int or float or list*) – *cval*|*low/high/delta*|**r***low/high*|**r***cval*. Find area, volume and mean height (volume/area) inside and above the *cval* contour. Alternatively, search using all contours from *low* to *high* in steps of *delta*. [Default returns area, volume and mean height of the entire grid]. The area is measured in the plane of the contour. Adding the **r** prefix computes the volume below the grid surface and above the planes defined by *low* and *high*, or below *cval* and grid's minimum. Note that this is an *outside* volume whilst the other forms compute an *inside* (below the surface) area volume. Use this form to compute

for example the volume of water between two contours. If no *contour* is given then there is no contour and the entire grid area, volume and the mean height is returned and *cval* will be reported as 0.

- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages

Returns

ret (*pandas.DataFrame or numpy.ndarray or None*) – Return type depends on `outfile` and `output_type`:

- None if `outfile` is set (output will be stored in file set by `outfile`)
- `pandas.DataFrame` or `numpy.ndarray` if `outfile` is not set (depends on `output_type` [Default is class:*pandas.DataFrame*])

Crossover analysis with x2sys:

x2sys_init(tag, *[, fmtfile, suffix, force, ...])	Initialize a new x2sys track database.
x2sys_cross([tracks, outfile, combitable, ...])	Calculate crossovers between track data files.

pygmt.x2sys_init

pygmt.**x2sys_init**(*tag*, *, *fmtfile=None*, *suffix=None*, *force=None*, *discontinuity=None*, *spacing=None*, *units=None*, *region=None*, *verbose=None*, *gap=None*, *distcalc=None*, ***kwargs*)

Initialize a new x2sys track database.

Serves as the starting point for x2sys and initializes a set of data bases that are particular to one kind of track data. These data, their associated data bases, and key parameters are given a short-hand notation called an x2sys TAG. The TAG keeps track of settings such as file format, whether the data are geographic or not, and the binning resolution for track indices.

Before you can run *pygmt.x2sys_init* you must set the environmental parameter X2SYS_HOME to a directory where you have write permission, which is where x2sys can keep track of your settings.

Full option list at https://docs.generic-mapping-tools.org/latest/supplements/x2sys/x2sys_init.html

Aliases:

- D = fmtfile
- E = suffix
- F = force

- G = discontinuity
- I = spacing
- N = units
- R = region
- V = verbose
- W = gap
- j = distcalc

Parameters

- **tag** (*str*) – The unique name of this data type x2sys TAG.
- **fmtfile** (*str*) – Format definition file prefix for this data set (see GMT's Format Definition Files for more information). Specify full path if the file is not in the current directory.

  Some file formats already have definition files premade. These include:

  – **mgd77** (for plain ASCII MGD77 data files)

  – **mgd77+** (for enhanced MGD77+ netCDF files)

  – **gmt** (for old mgg supplement binary files)

  – **xy** (for plain ASCII x, y tables)

  – **xyz** (same, with one z-column)

  – **geo** (for plain ASCII longitude, latitude files)

  – **geoz** (same, with one z-column).
- **suffix** (*str*) – Specifies the file extension (suffix) for these data files. If not given we use the format definition file prefix as the suffix (see `fmtfile`).
- **discontinuity** (*str*) – **d**|**g**. Selects geographical coordinates. Append **d** for discontinuity at the Dateline (makes longitude go from -180 to +180) or **g** for discontinuity at Greenwich (makes longitude go from 0 to 360 [Default]). If not given we assume the data are Cartesian.
- **spacing** (*str or list*) – *dx*[/*dy*]. *dx* and optionally *dy* is the grid spacing. Append **m** to indicate minutes or **s** to indicate seconds for geographic data. These spacings refer to the binning used in the track bin-index data base.
- **units** (*str or list*) – **d**|**s***unit*. Sets the units used for distance and speed when requested by other programs. Append **d** for distance or **s** for speed, then give the desired *unit* as:

  – **c** - Cartesian userdist or userdist/usertime

  – **e** - meters or m/s

  – **f** - feet or feet/s

  – **k** - km or km/hr

  – **m** - miles or miles/hr

  – **n** - nautical miles or knots

  – **u** - survey feet or survey feet/s

  [Default is `units=["dk", "se"]` (km and m/s) if `discontinuity` is set, and `units=["dc", "sc"]` otherwise (e.g., for Cartesian units)].

- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **gap** (*str or list*) – **t**|**d***gap*. Give **t** or **d** and append the corresponding maximum time gap (in user units; this is typically seconds [Default is infinity]), or distance (for units, see `units`) gap [Default is infinity]) allowed between the two data points immediately on either side of a crossover. If these limits are exceeded then a data gap is assumed and no COE will be determined.
- **distcalc** (*str*) – **e**|**f**|**g**. Determine how spherical distances are calculated.
  - **e** - Ellipsoidal (or geodesic) mode
  - **f** - Flat Earth mode
  - **g** - Great circle distance [Default]

  All spherical distance calculations depend on the current ellipsoid (PROJ_ELLIPSOID), the definition of the mean radius (PROJ_MEAN_RADIUS), and the specification of latitude type (PROJ_AUX_LATITUDE). Geodesic distance calculations is also controlled by method (PROJ_GEODESIC).

pygmt.x2sys_cross

pygmt.**x2sys_cross**(*tracks=None*, *outfile=None*, ***, *combitable=None*, *runtimes=None*, *override=None*, *interpolation=None*, *region=None*, *speed=None*, *tag=None*, *coe=None*, *verbose=None*, *numpoints=None*, *trackvalues=None*, ***kwargs*)

Calculate crossovers between track data files.

Determines all intersections between ("external cross-overs") or within ("internal cross-overs") tracks (Cartesian or geographic), and report the time, position, distance along track, heading and speed along each track segment, and the crossover error (COE) and mean values for all observables. By default, *`pygmt.x2sys_cross`* will look for both external and internal COEs. As an option, you may choose to project all data using one of the map projections prior to calculating the COE.

Full option list at https://docs.generic-mapping-tools.org/latest/supplements/x2sys/x2sys_cross.html

Aliases:

- A = combitable
- C = runtimes
- D = override
- I = interpolation

- Q = coe
- R = region
- S = speed
- T = tag
- V = verbose
- W = numpoints
- Z = trackvalues

Parameters

- **`tracks`** (*`pandas.DataFrame` or `str` or `list`*) – A table or a list of tables with (x, y) or (lon, lat) values in the first two columns. Track(s) can be provided as pandas DataFrame tables or file names. Supported file formats are ASCII, native binary, or COARDS netCDF 1-D data. More columns may also be present.

  If the filenames are missing their file extension, we will append the suffix specified for this TAG. Track files will be searched for first in the current directory and second in all directories listed in $X2SYS_HOME/TAG/TAG_paths.txt (if it exists). [If $X2SYS_HOME is not set it will default to $GMT_SHAREDIR/x2sys]. (Note: MGD77 files will also be looked for via $MGD77_HOME/mgd77_paths.txt and .gmt files will be searched for via $GMT_SHAREDIR/mgg/gmtfile_paths).

- **`outfile`** (*`str`*) – Optional. The file name for the output ASCII txt file to store the table in.
- **`tag`** (*`str`*) – Specify the x2sys TAG which identifies the attributes of this data type.
- **`combitable`** (*`str`*) – Only process the pair-combinations found in the file *combitable* [Default process all possible combinations among the specified files]. The file *combitable* is created by x2sys_get's -L option.
- **`runtimes`** (*`bool` or `str`*) – Compute and append the processing run-time for each pair to the progress message (use `runtimes=True`). Pass in a filename (e.g. `runtimes="file.txt"`) to save these run-times to file. The idea here is to use the knowledge of run-times to split the main process in a number of sub-processes that can each be launched in a different processor of your multi-core machine. See the MATLAB function split_file4coes.m.
- **`override`** (*`bool` or `str`*) – **S**|**N**. Control how geographic coordinates are handled (Cartesian data are unaffected). By default, we determine if the data are closer to one pole than the other, and then we use a cylindrical polar conversion to avoid problems with longitude jumps. You can turn this off entirely with `override` and then the calculations uses the original data (we have protections against longitude jumps). However, you can force the selection of the pole for the projection by appending **S** or **N** for the south or north pole, respectively. The conversion is used because the algorithm used to find crossovers is inherently a Cartesian algorithm that can run into trouble with data that has large longitudinal range at higher latitudes.
- **`interpolation`** (*`str`*) – **l**|**a**|**c**. Sets the interpolation mode for estimating values at the crossover. Choose among:
  - **l** - Linear interpolation [Default].
  - **a** - Akima spline interpolation.
  - **c** - Cubic spline interpolation.
- **`coe`** (*`str`*) – Use **e** for external COEs only, and **i** for internal COEs only [Default is all COEs].

- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **speed** (*str or list*) – **l**|**u**|**h***speed*. Defines window of track speeds. If speeds are outside this window we do not calculate a COE. Specify:
  - **l** sets lower speed [Default is 0].
  - **u** sets upper speed [Default is infinity].
  - **h** does not limit the speed but sets a lower speed below which headings will not be computed (i.e., set to NaN) [Default calculates headings regardless of speed].

  For example, you can use `speed=["l0", "u10", "h5"]` to set a lower speed of 0, upper speed of 10, and disable heading calculations for speeds below 5.
- **verbose** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **numpoints** (*int*) – Give the maximum number of data points on either side of the crossover to use in the spline interpolation [Default is 3].
- **trackvalues** (*bool*) – Report the values of each track at the crossover [Default reports the crossover value and the mean value].

Returns

crossover_errors (`pandas.DataFrame` or None) – Table containing crossover error information. Return type depends on whether the `outfile` parameter is set:

- `pandas.DataFrame` with (x, y, ..., etc) if `outfile` is not set
- None if `outfile` is set (track output will be stored in the set in `outfile`)

9.21.4 Input/output

load_dataarray(filename_or_obj, **kwargs)	Open, load into memory, and close a DataArray from a file or file-like object containing a single data variable.

pygmt.load_dataarray

`pygmt.load_dataarray(filename_or_obj, **kwargs)`

Open, load into memory, and close a DataArray from a file or file-like object containing a single data variable.

This is a thin wrapper around `xarray.open_dataarray`. It differs from `xarray.open_dataarray` in that it loads the DataArray into memory, gets GMT specific metadata about the grid via `GMTDataArrayAccessor`, closes the file, and returns the DataArray. In contrast, `xarray.open_dataarray` keeps the file handle open and lazy loads its contents. All parameters are passed directly to `xarray.open_dataarray`. See that documentation for further details.

Parameters **filename_or_obj** (*str or pathlib.Path or file-like or DataStore*) – Strings and Path objects are interpreted as a path to a netCDF file or an OpenDAP URL and opened with python-netCDF4, unless the filename ends with .gz, in which case the file is gunzipped and opened with scipy.io.netcdf (only netCDF3 supported). Byte-strings or file-like objects are opened by scipy.io.netcdf (netCDF3) or h5py (netCDF4/HDF).

Returns **datarray** (*xarray.DataArray*) – The newly created DataArray.

See also:

`xarray.open_dataarray`

9.21.5 GMT Defaults

Operations on GMT defaults:

config(**kwargs)	Set GMT defaults globally or locally.

pygmt.config

`class pygmt.config(**kwargs)`

Set GMT defaults globally or locally.

Change GMT defaults globally:

```
pygmt.config(PARAMETER=value)
```

Change GMT defaults locally by using it as a context manager:

```
with pygmt.config(PARAMETER=value):
    ...
```

Full GMT defaults list at https://docs.generic-mapping-tools.org/latest/gmt.conf.html

Methods Summary

Examples using `pygmt.config`

- *Calculating grid gradient and radiance*
- *Double Y axes graph*
- *Configuring PyGMT defaults*
- *Plotting datetime charts*
- *Plotting text*
- *Plotting vectors*
- *Polar*

9.21.6 Metadata

Getting metadata from tabular or grid data:

GMTDataArrayAccessor(xarray_obj)	This is the GMT extension for `xarray.DataArray`.
info(data, *[, per_column, spacing, ...])	Get information about data tables.
grdinfo(grid, *[, per_column, tiles, ...])	Get information about a grid.

pygmt.GMTDataArrayAccessor

class pygmt.**GMTDataArrayAccessor**(*xarray_obj*)

This is the GMT extension for `xarray.DataArray`.

You can access various GMT specific metadata about your grid as follows:

```
>>> from pygmt.datasets import load_earth_relief
>>> # Use the global Earth relief grid with 1 degree spacing
>>> grid = load_earth_relief(resolution="01d")
```

```
>>> # See if grid uses Gridline (0) or Pixel (1) registration
>>> grid.gmt.registration
1
>>> # See if grid uses Cartesian (0) or Geographic (1) coordinate system
>>> grid.gmt.gtype
1
```

Methods Summary

---

pygmt.info

`pygmt.info`(*data*, ***, *per_column=None*, *spacing=None*, *nearest_multiple=None*, *verbose=None*, *aspatial=None*, *coltypes=None*, *incols=None*, *registration=None*, ***kwargs*)

Get information about data tables.

Reads from files and finds the extreme values in each of the columns reported as min/max pairs. It recognizes NaNs and will print warnings if the number of columns vary from record to record. As an option, it will find the extent of the first two columns rounded up and down to the nearest multiple of the supplied increments given by `spacing`. Such output will be in a numpy.ndarray form [*w*, *e*, *s*, *n*], which can be used directly as the `region` parameter for other modules (hence only *dx* and *dy* are needed). If the `per_column` parameter is combined with `spacing`, then the numpy.ndarray output will be rounded up/down for as many columns as there are increments provided in `spacing`. A similar parameter `nearest_multiple` will provide a numpy.ndarray in the form of [*zmin*, *zmax*, *dz*] for makecpt.

Full option list at https://docs.generic-mapping-tools.org/latest/gmtinfo.html

Aliases:

- C = per_column
- I = spacing
- T = nearest_multiple
- V = verbose
- a = aspatial
- f = coltypes
- i = incols
- r = registration

Parameters

- **data** (*str* or *numpy.ndarray* or *pandas.DataFrame* or *xarray.Dataset* or *geopandas.GeoDataFrame*) – Pass in either a file name to an ASCII data table, a 1D/2D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **per_column** (*bool*) – Report the min/max values per column in separate columns.
- **spacing** (*str*) – [**b**|**p**|**f**|**s**]*dx*[/*dy*[/*dz*...]]. Compute the min/max values of the first n columns to the nearest multiple of the provided increments [default is 2 columns]. By default, output results in the form `[w, e, s, n]`, unless `per_column` is set in which case we output each min and max value in separate output columns.
- **nearest_multiple** (*str*) – **dz**[**+c***col*]. Report the min/max of the first (0'th) column to the nearest multiple of dz and output this in the form `[zmin, zmax, dz]`.
- **verbose** (*bool* or *str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced

    - **e** - Error messages only
    - **w** - Warnings [Default]
    - **t** - Timings (report runtimes for time-intensive algorithms);
    - **i** - Informational messages (same as `verbose=True`)
    - **c** - Compatibility warnings
    - **d** - Debugging messages

- **aspatial** (*bool or str*) – [*col*=]*name*[,. . .]. Control how aspatial data are handled during input and output. Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#aspatial-full.
- **incols** (*str or 1d array*) – Specify data columns for primary input in arbitrary order. Columns can be repeated and columns not listed will be skipped [Default reads all columns in order, starting with the first (i.e., column 0)].
    - For *1d array*: specify individual columns in input order (e.g., `incols=[1,0]` for the 2nd column followed by the 1st column).
    - For `str`: specify individual columns or column ranges in the format *start*[:*inc*]:*stop*, where *inc* defaults to 1 if not specified, with columns and/or column ranges separated by commas (e.g., `incols="0:2,4+l"` to input the first three columns followed by the log-transformed 5th column). To read from a given column until the end of the record, leave off *stop* when specifying the column range. To read trailing text, add the column **t**. Append the word number to **t** to ingest only a single word from the trailing text. Instead of specifying columns, use `incols="n"` to simply read numerical input and skip trailing text. Optionally, append one of the following modifiers to any column or column range to transform the input columns:
        * **+l** to take the *log10* of the input values.
        * **+d** to divide the input values by the factor *divisor* [Default is 1].
        * **+s** to multiple the input values by the factor *scale* [Default is 1].
        * **+o** to add the given *offset* to the input values [Default is 0].
- **coltypes** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.
- **registration** (*str*) – **g**|**p**. Force gridline (**g**) or pixel (**p**) node registration. [Default is **g**(ridline)].

Returns

output (*np.ndarray or str*) – Return type depends on whether any of the `per_column`, `spacing`, or `nearest_multiple` parameters are set.

- `numpy.ndarray` if either of the above parameters are used.
- str if none of the above parameters are used.

Examples using `pygmt.info`

- *Color points by categories*
- *3D Scatter plots*
- *Plotting datetime charts*

pygmt.grdinfo

`pygmt.`**`grdinfo`**(*grid*, ***, *per_column=None*, *tiles=None*, *geographic=None*, *spacing=None*, *force_scan=None*, *minmax_pos=None*, *region=None*, *nearest_multiple=None*, *verbose=None*, *coltypes=None*, ***kwargs*)

Get information about a grid.

Can read the grid from a file or given as an xarray.DataArray grid.

Full option list at https://docs.generic-mapping-tools.org/latest/grdinfo.html

Aliases:

- C = per_column
- D = tiles
- F = geographic
- I = spacing
- L = force_scan
- M = minmax_pos
- R = region
- T = nearest_multiple
- V = verbose
- f = coltypes

Parameters

- **`grid`** (*`str` or `xarray.DataArray`*) – The file name of the input grid or the grid loaded as a DataArray. This is the only required parameter.
- **`region`** (*`str` or `list`*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **`per_column`** (*`str` or `bool`*) – **n**|**t**. Formats the report using tab-separated fields on a single line. The output is name *w e s n z0 z1 dx dy nx ny* [*x0 y0 x1 y1*] [*med scale*] [*mean std rms*] [*n_nan*] *registration gtype*. The data in brackets are outputted depending on the `force_scan` and `minmax_pos` parameters. Use **t** to place file name at the end of the output record or, **n** or `True` to only output numerical columns. The registration is either 0 (gridline) or 1 (pixel), while gtype is either 0 (Cartesian) or 1 (geographic). The default value is `False`. This cannot be called if `geographic` is also set.
- **`tiles`** (*`str` or `list`*) – *xoff*[/*yoff*][**+i**]. Divide a single grid's domain (or the `region` domain, if no grid given) into tiles of size dx times dy (set via `spacing`). You can specify overlap between tiles by appending *xoff*[/*yoff*]. If the single grid is given you may use the modifier **+i** to ignore tiles that have no data within each tile subregion. Default output is text

region strings. Use `per_column` to instead report four columns with xmin xmax ymin ymax per tile, or use `per_column="t"` to also have the region string appended as trailing text.

- **`geographic`** (*bool*) – Report grid domain and x/y-increments in world mapping format The default value is `False`. This cannot be called if `per_column` is also set.
- **`spacing`** (*str or list*) – *dx*[/*dy*]|**b**|**i**|**r**. Report the min/max of the region to the nearest multiple of dx and dy, and output this in the form w/e/s/n (unless `per_column` is set). To report the actual grid region, append **r**. For a grid produced by the img supplement (a Cartesian Mercator grid), the exact geographic region is given with **i** (if not found then we return the actual grid region instead). If no argument is given then we report the grid increment in the form *xinc*[/*yinc*]. If **b** is given we write each grid's bounding box polygon instead. Finally, if `tiles` is in effect then *dx* and *dy* are the dimensions of the desired tiles.
- **`force_scan`** (*int or str*) – **0**|**1**|**2**|**p**|**a**.

  0: Report range of z after actually scanning the data, not just reporting what the header says. **1**: Report median and L1 scale of z (L1 scale = 1.4826 * Median Absolute Deviation (MAD)). **2**: Report mean, standard deviation, and root-mean-square (rms) of z. **p**: Report mode (LMS) and LMS scale of z. **a**: Include all of the above.
- **`minxmax_pos`** (*bool*) – Include the x/y values at the location of the minimum and maximum z-values.
- **`nearest_multiple`** (*str*) – [*dz*][**+a**[*alpha*]][**+s**]. Determine min and max z-value. If *dz* is provided then we first round these values off to multiples of *dz*. To exclude the two tails of the distribution when determining the min and max you can add **+a** to set the *alpha* value (in percent): We then sort the grid, exclude the data in the 0.5**alpha* and 100 - 0.5**alpha* tails, and revise the min and max. To force a symmetrical range about zero, using minus/plus the max absolute value of the two extremes, append **+s**. We report the result via the text string *zmin/zmax* or *zmin/zmax/dz* (if *dz* was given) as expected by *`pygmt.makecpt`*.
- **`verbose`** (*bool or str*) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages
- **`coltypes`** (*str*) – [**i**|**o**]*colinfo*. Specify data types of input and/or output columns (time or geographical data). Full documentation is at https://docs.generic-mapping-tools.org/latest/gmt.html#f-full.

Returns **info** (*str*) – A string with information about the grid.

9.21.7 Miscellaneous

which(fname, *[, download, verbose])	Find the full path to specified files.
test([doctest, verbose, coverage, figures])	Run the test suite.
print_clib_info()	Print information about the GMT shared library that we can find.
show_versions()	Prints various dependency versions useful when submitting bug reports.

pygmt.which

pygmt.**which**(*fname*, *, *download=None*, *verbose=None*, ***kwargs*)

Find the full path to specified files.

Reports the full paths to the files given through *fname*. We look for the file in (1) the current directory, (2) in $GMT_USERDIR (if defined), (3) in $GMT_DATADIR (if defined), or (4) in $GMT_CACHEDIR (if defined).

fname can also be a downloadable file (either a full URL, a *@file* special file for downloading from the GMT Site Cache, or *@earth_relief_** topography grids). In these cases, use option *download* to set the desired behavior. If *download* is not used (or False), the file will not be found.

Full option list at https://docs.generic-mapping-tools.org/latest/gmtwhich.html

Aliases:

- G = download
- V = verbose

Parameters

- **fname** (`str or list`) – One or more file names of any data type (grids, tables, etc.).
- **download** (`bool or str`) – [**a**|**c**|**l**|**u**]. If the fname argument is a downloadable file (either a complete URL, an @file for downloading from the GMT data server, or @earth_relief_xxy) we will try to download the file if it is not found in your local data or cache dirs. By default [`download=True` or `download="l"`] we download to the current directory. Use **a** to place files in the appropriate folder under the user directory (this is where GMT normally places downloaded files), **c** to place it in the user cache directory, or **u** for the user data directory instead (i.e., ignoring any subdirectory structure).
- **verbose** (`bool or str`) – Select verbosity level [Default is **w**], which modulates the messages written to stderr. Choose among 7 levels of verbosity:
  - **q** - Quiet, not even fatal error messages are produced
  - **e** - Error messages only
  - **w** - Warnings [Default]
  - **t** - Timings (report runtimes for time-intensive algorithms);
  - **i** - Informational messages (same as `verbose=True`)
  - **c** - Compatibility warnings
  - **d** - Debugging messages

Returns **path** (*str or list*) – The path(s) to the file(s), depending on the options used.

Raises `FileNotFoundError` – If the file is not found.

pygmt.test

`pygmt.test(doctest=True, verbose=True, coverage=False, figures=True)`

Run the test suite.

Uses pytest to discover and run the tests. If you haven't already, you can install it with conda or pip.

Parameters

- **doctest** (*bool*) – If `True`, will run the doctests as well (code examples that start with a >>> in the docs).
- **verbose** (*bool*) – If `True`, will print extra information during the test run.
- **coverage** (*bool*) – If `True`, will run test coverage analysis on the code as well. Requires `pytest-cov`.
- **figures** (*bool*) – If `True`, will test generated figures against saved baseline figures. Requires `pytest-mpl` and `matplotlib`.

Raises `AssertionError` – If pytest returns a non-zero error code indicating that some tests have failed.

pygmt.print_clib_info

`pygmt.print_clib_info()`

Print information about the GMT shared library that we can find.

Includes the GMT version, default values for parameters, the path to the `libgmt` shared library, and GMT directories.

pygmt.show_versions

`pygmt.show_versions()`

Prints various dependency versions useful when submitting bug reports. This includes information about:

- PyGMT itself
- System information (Python version, Operating System)
- Core dependency versions (Numpy, Pandas, Xarray, etc)
- GMT library information

9.21.8 Datasets

PyGMT provides access to GMT's datasets through the *pygmt.datasets* package. These functions will download the datasets automatically the first time they are used and store them in the GMT cache folder.

datasets.load_earth_age([resolution, ...])	Load Earth seafloor crustal ages in various resolutions.
datasets.load_earth_relief([resolution, ...])	Load Earth relief grids (topography and bathymetry) in various resolutions.
datasets.load_fractures_compilation()	Load a table of fracture lengths and azimuths as hypothetically digitized from geological maps as a pandas.DataFrame.

continues on next page

Table 17 – continued from previous page

datasets.load_hotspots()	Load a table with the locations, names, and suggested symbol sizes of hotspots.
datasets.load_japan_quakes()	Load a table of earthquakes around Japan as a pandas.DataFrame.
datasets.load_mars_shape()	Load a table of data for the shape of Mars.
datasets.load_ocean_ridge_points()	Load a table of ocean ridge points for the entire world as a pandas.DataFrame.
datasets.load_sample_bathymetry()	Load a table of ship observations of bathymetry off Baja California as a pandas.DataFrame.
datasets.load_usgs_quakes()	Load a table of global earthquakes form the USGS as a pandas.DataFrame.

pygmt.datasets.load_earth_age

pygmt.datasets.**load_earth_age**(*resolution='01d'*, *region=None*, *registration=None*)

Load Earth seafloor crustal ages in various resolutions.

The grids are downloaded to a user data directory (usually `~/.gmt/server/earth/earth_age/`) the first time you invoke this function. Afterwards, it will load the grid from the data directory. So you'll need an internet connection the first time around.

These grids can also be accessed by passing in the file name **@earth_age**_*res*[_*reg*] to any grid plotting/processing function. *res* is the grid resolution (see below), and *reg* is grid registration type (**p** for pixel registration or **g** for gridline registration).

Refer to https://docs.generic-mapping-tools.org/latest/datasets/remote-data.html#global-earth-seafloor-crustal-age-grids for more details.

Parameters

- **resolution** (*str*) – The grid resolution. The suffix `d` and `m` stand for arc-degree, arc-minute and arc-second. It can be `'01d'`, `'30m'`, `'20m'`, `'15m'`, `'10m'`, `'06m'`, `'05m'`, `'04m'`, `'03m'`, `'02m'`, or `'01m'`.
- **region** (*str or list*) – The subregion of the grid to load, in the forms of a list [*xmin*, *xmax*, *ymin*, *ymax*] or a string *xmin/xmax/ymin/ymax*. Required for grids with resolutions higher than 5 arc-minute (i.e., `05m`).
- **registration** (*str*) – Grid registration type. Either `pixel` for pixel registration or `gridline` for gridline registration. Default is `None`, where a pixel-registered grid is returned unless only the gridline-registered grid is available.

Returns **grid** (`xarray.DataArray`) – The Earth seafloor crustal age grid. Coordinates are latitude and longitude in degrees. Age is in millions of years (Myr).

Notes

The `xarray.DataArray` grid doesn't support slice operation, for Earth seafloor crustal age with resolutions of 5 arc-minutes or higher, which are stored as smaller tiles.

pygmt.datasets.load_earth_relief

`pygmt.datasets.load_earth_relief`(*resolution='01d'*, *region=None*, *registration=None*, *use_srtm=False*)

Load Earth relief grids (topography and bathymetry) in various resolutions.

The grids are downloaded to a user data directory (usually `~/.gmt/server/earth/earth_relief/`) the first time you invoke this function. Afterwards, it will load the grid from the data directory. So you'll need an internet connection the first time around.

These grids can also be accessed by passing in the file name **@earth_relief**_*res*[_*reg*] to any grid plotting/processing function. *res* is the grid resolution (see below), and *reg* is grid registration type (**p** for pixel registration or **g** for gridline registration).

Refer to https://docs.generic-mapping-tools.org/latest/datasets/remote-data.html#global-earth-relief-grids for more details.

Parameters

- **resolution** (*str*) – The grid resolution. The suffix `d`, `m` and `s` stand for arc-degree, arc-minute and arc-second. It can be `'01d'`, `'30m'`, `'20m'`, `'15m'`, `'10m'`, `'06m'`, `'05m'`, `'04m'`, `'03m'`, `'02m'`, `'01m'`, `'30s'`, `'15s'`, `'03s'`, or `'01s'`.
- **region** (*str or list*) – The subregion of the grid to load, in the forms of a list [*xmin*, *xmax*, *ymin*, *ymax*] or a string *xmin/xmax/ymin/ymax*. Required for Earth relief grids with resolutions higher than 5 arc-minute (i.e., `05m`).
- **registration** (*str*) – Grid registration type. Either `pixel` for pixel registration or `gridline` for gridline registration. Default is `None`, where a pixel-registered grid is returned unless only the gridline-registered grid is available.
- **use_srtm** (*bool*) – By default, the land-only SRTM tiles from NASA are used to generate the `'03s'` and `'01s'` grids, and the missing ocean values are filled by up-sampling the SRTM15+V2.1 tiles which have a resolution of 15 arc-second (i.e., `'15s'`). If True, will only load the original land-only SRTM tiles.

Returns **grid** (`xarray.DataArray`) – The Earth relief grid. Coordinates are latitude and longitude in degrees. Relief is in meters.

Notes

The `xarray.DataArray` grid doesn't support slice operation, for Earth relief data with resolutions of 5 arc-minutes or higher, which are stored as smaller tiles.

Examples

```
>>> # load the default grid (pixel-registered 01d grid)
>>> grid = load_earth_relief()
>>> # load the 30m grid with "gridline" registration
>>> grid = load_earth_relief("30m", registration="gridline")
>>> # load high-resolution grid for a specific region
>>> grid = load_earth_relief(
...     "05m", region=[120, 160, 30, 60], registration="gridline"
... )
>>> # load the original 3 arc-second land-only SRTM tiles from NASA
>>> grid = load_earth_relief(
...     "03s",
```

(continues on next page)

(continued from previous page)

```
...     region=[135, 136, 35, 36],
...     registration="gridline",
...     use_srtm=True,
... )
```

Examples using `pygmt.datasets.load_earth_relief`

- *Calculating grid gradient and radiance*
- *Clipping grid values*
- *Sampling along tracks*
- *Multiple colormaps*
- *Creating a 3D perspective image*
- *Creating a map with contour lines*
- *Plotting Earth relief*

pygmt.datasets.load_fractures_compilation

`pygmt.datasets.`**`load_fractures_compilation`**`()`

Load a table of fracture lengths and azimuths as hypothetically digitized from geological maps as a pandas.DataFrame.

This is the `@fractures_06.txt` dataset used in the GMT tutorials.

The data are downloaded to a cache directory (usually `~/.gmt/cache`) the first time you invoke this function. Afterwards, it will load the data from the cache. So you'll need an internet connection the first time around.

Returns data (*pandas.DataFrame*) – The data table. Use `print(data.describe())` to see the available columns.

Examples using `pygmt.datasets.load_fractures_compilation`

- *Rose diagram*

pygmt.datasets.load_hotspots

`pygmt.datasets.`**`load_hotspots`**`()`

Load a table with the locations, names, and suggested symbol sizes of hotspots.

This is the `@hotspots.txt` dataset used in the GMT tutorials, with data from Mueller, Royer, and Lawver, 1993, Geology, vol. 21, pp. 275-278. The main 5 hotspots used by Doubrovine et al. [2012] have symbol sizes twice the size of all other hotspots.

The data are downloaded to a cache directory (usually `~/.gmt/cache`) the first time you invoke this function. Afterwards, it will load the data from the cache. So you'll need an internet connection the first time around.

Returns data (*pandas.DataFrame*) – The data table with columns "longitude", "latitude", "symbol_size", and "placename".

pygmt.datasets.load_japan_quakes

`pygmt.datasets.`**`load_japan_quakes`**`()`

Load a table of earthquakes around Japan as a pandas.DataFrame.

Data is from the NOAA NGDC database. This is the `@tut_quakes.ngdc` dataset used in the GMT tutorials.

The data are downloaded to a cache directory (usually `~/.gmt/cache`) the first time you invoke this function. Afterwards, it will load the data from the cache. So you'll need an internet connection the first time around.

Returns **data** (*pandas.DataFrame*) – The data table. Columns are year, month, day, latitude, longitude, depth (in km), and magnitude of the earthquakes.

Examples using `pygmt.datasets.load_japan_quakes`

- *Plotting data points*

pygmt.datasets.load_mars_shape

`pygmt.datasets.`**`load_mars_shape`**`()`

Load a table of data for the shape of Mars.

This is the `@mars370d.txt` dataset used in GMT examples, with data and information from Smith, D. E., and M. T. Zuber (1996), The shape of Mars and the topographic signature of the hemispheric dichotomy. Data columns are "longitude," "latitude", and "radius (meters)."

The data are downloaded to a cache directory (usually `~/.gmt/cache`) the first time you invoke this function. Afterwards, it will load the data from the cache. So you'll need an internet connection the first time around.

Returns **data** (*pandas.DataFrame*) – The data table with columns "longitude", "latitude", and "radius(m)".

pygmt.datasets.load_ocean_ridge_points

`pygmt.datasets.`**`load_ocean_ridge_points`**`()`

Load a table of ocean ridge points for the entire world as a pandas.DataFrame.

This is the `@ridge.txt` dataset used in the GMT tutorials.

The data are downloaded to a cache directory (usually `~/.gmt/cache`) the first time you invoke this function. Afterwards, it will load the data from the cache. So you'll need an internet connection the first time around.

Returns **data** (*pandas.DataFrame*) – The data table. Columns are longitude and latitude.

Examples using `pygmt.datasets.load_ocean_ridge_points`

- *Sampling along tracks*

pygmt.datasets.load_sample_bathymetry

pygmt.datasets.**load_sample_bathymetry**()

Load a table of ship observations of bathymetry off Baja California as a pandas.DataFrame.

This is the `@tut_ship.xyz` dataset used in the GMT tutorials.

The data are downloaded to a cache directory (usually `~/.gmt/cache`) the first time you invoke this function. Afterwards, it will load the data from the cache. So you'll need an internet connection the first time around.

Returns data (*pandas.DataFrame*) – The data table. Columns are longitude, latitude, and bathymetry.

pygmt.datasets.load_usgs_quakes

pygmt.datasets.**load_usgs_quakes**()

Load a table of global earthquakes form the USGS as a pandas.DataFrame.

This is the `@usgs_quakes_22.txt` dataset used in the GMT tutorials.

The data are downloaded to a cache directory (usually `~/.gmt/cache`) the first time you invoke this function. Afterwards, it will load the data from the cache. So you'll need an internet connection the first time around.

Returns data (*pandas.DataFrame*) – The data table. Use `print(data.describe())` to see the available columns.

9.21.9 Exceptions

All custom exceptions are derived from *pygmt.exceptions.GMTError*.

exceptions.GMTError	Base class for all GMT related errors.
exceptions.GMTInvalidInput	Raised when the input of a function/method is invalid.
exceptions.GMTVersionError	Raised when an incompatible version of GMT is being used.
exceptions.GMTOSError	Unsupported operating system.
exceptions.GMTCLibError	Error encountered when running a function from the GMT shared library.
exceptions.GMTCLibNoSessionError	Tried to access GMT API without a currently open GMT session.
exceptions.GMTCLibNotFoundError	Could not find the GMT shared library.

pygmt.exceptions.GMTError

exception pygmt.exceptions.**GMTError**

Base class for all GMT related errors.

pygmt.exceptions.GMTInvalidInput

exception pygmt.exceptions.**GMTInvalidInput**
Raised when the input of a function/method is invalid.

pygmt.exceptions.GMTVersionError

exception pygmt.exceptions.**GMTVersionError**
Raised when an incompatible version of GMT is being used.

pygmt.exceptions.GMTOSError

exception pygmt.exceptions.**GMTOSError**
Unsupported operating system.

pygmt.exceptions.GMTCLibError

exception pygmt.exceptions.**GMTCLibError**
Error encountered when running a function from the GMT shared library.

pygmt.exceptions.GMTCLibNoSessionError

exception pygmt.exceptions.**GMTCLibNoSessionError**
Tried to access GMT API without a currently open GMT session.

pygmt.exceptions.GMTCLibNotFoundError

exception pygmt.exceptions.**GMTCLibNotFoundError**
Could not find the GMT shared library.

9.21.10 GMT C API

The `pygmt.clib` package is a wrapper for the GMT C API built using `ctypes`. Most calls to the C API happen through the `pygmt.clib.Session` class.

clib.Session()	A GMT API session where most operations involving the C API happen.

pygmt.clib.Session

class pygmt.clib.**Session**
A GMT API session where most operations involving the C API happen.

Works as a context manager (for use in a `with` block) to create a GMT C API session and destroy it in the end to clean up memory.

Functions of the shared library are exposed as methods of this class. Most methods MUST be used with an open session (inside a `with` block). If creating GMT data structures to communicate data, put that code inside the same `with` block as the API calls that will use the data.

By default, will let ctypes try to find the GMT shared library (libgmt). If the environment variable GMT_LIBRARY_PATH is set, will look for the shared library in the directory specified by it.

A GMTVersionError exception will be raised if the GMT shared library reports a version older than the required minimum GMT version.

The session_pointer attribute holds a ctypes pointer to the currently open session.

Raises

- ***GMTCLibNotFoundError*** – If there was any problem loading the library (couldn't find it or couldn't access the functions).
- ***GMTCLibNoSessionError*** – If you try to call a method outside of a 'with' block.
- ***GMTVersionError*** – If the minimum required version of GMT is not found.

Examples

```
>>> from pygmt.datasets import load_earth_relief
>>> from pygmt.helpers import GMTTempFile
>>> grid = load_earth_relief()
>>> type(grid)
<class 'xarray.core.dataarray.DataArray'>
>>> # Create a session and destroy it automatically when exiting the "with"
>>> # block.
>>> with Session() as ses:
...     # Create a virtual file and link to the memory block of the grid.
...     with ses.virtualfile_from_grid(grid) as fin:
...         # Create a temp file to use as output.
...         with GMTTempFile() as fout:
...             # Call the grdinfo module with the virtual file as input
...             # and the temp file as output.
...             ses.call_module("grdinfo", f"{fin} -C ->{fout.name}")
...             # Read the contents of the temp file before it's deleted.
...             print(fout.read().strip())
...
-180 180 -90 90 -8182 5651.5 1 1 360 180 1 1
```

Methods Summary

Session.call_module(module, args)	Call a GMT module with the given arguments.
Session.create(name)	Create a new GMT C API session.
Session.create_data(family, geometry, mode, ...)	Create an empty GMT data container.
Session.destroy()	Destroy the currently open GMT API session.
Session.extract_region()	Extract the WESN bounding box of the currently active figure.
Session.get_default(name)	Get the value of a GMT default parameter (library version, paths, etc).
Session.get_libgmt_func(name[, argtypes, ...])	Get a ctypes function from the libgmt shared library.
Session.open_virtual_file(family, geometry, ...)	Open a GMT Virtual File to pass data to and from a module.
Session.put_matrix(dataset, matrix[, pad])	Attach a numpy 2D array to a GMT dataset.

continues on next page

Table 20 – continued from previous page

Session.put_strings(dataset, family, strings)	Attach a numpy 1D array of dtype str as a column on a GMT dataset.
Session.put_vector(dataset, column, vector)	Attach a numpy 1D array as a column on a GMT dataset.
Session.virtualfile_from_data([check_kind, ...])	Store any data inside a virtual file.
Session.virtualfile_from_grid(grid)	Store a grid in a virtual file.
Session.virtualfile_from_matrix(matrix)	Store a 2d array as a table inside a virtual file.
Session.virtualfile_from_vectors(*vectors)	Store 1d arrays as columns of a table inside a virtual file.
Session.write_data(family, geometry, mode, ...)	Write a GMT data container to a file.

GMT modules are executed through the `call_module` method:

clib.Session.call_module(module, args)	Call a GMT module with the given arguments.

pygmt.clib.Session.call_module

`Session.`**`call_module`**(*module*, *args*)

Call a GMT module with the given arguments.

Makes a call to `GMT_Call_Module` from the C API using mode `GMT_MODULE_CMD` (arguments passed as a single string).

Most interactions with the C API are done through this function.

Parameters

- **`module`** (*`str`*) – Module name (`'coast'`, `'basemap'`, etc).
- **`args`** (*`str`*) – String with the command line arguments that will be passed to the module (for example, `'-R0/5/0/10 -JM'`).

Raises ***`GMTCLibError`*** – If the returned status code of the function is non-zero.

Passing memory blocks between Python data objects (e.g. `numpy.ndarray`, `pandas.Series`, `xarray.DataArray`, etc) and GMT happens through *virtual files*. These methods are context managers that automate the conversion of Python variables to GMT virtual files:

clib.Session.virtualfile_from_data([...])	Store any data inside a virtual file.
clib.Session.virtualfile_from_matrix(matrix)	Store a 2d array as a table inside a virtual file.
clib.Session.virtualfile_from_vectors(*vectors)	Store 1d arrays as columns of a table inside a virtual file.
clib.Session.virtualfile_from_grid(grid)	Store a grid in a virtual file.

pygmt.clib.Session.virtualfile_from_data

`Session.`**`virtualfile_from_data`**(*check_kind=None*, *data=None*, *x=None*, *y=None*, *z=None*, *extra_arrays=None*, *required_z=False*)

Store any data inside a virtual file.

This convenience function automatically detects the kind of data passed into it, and produces a virtualfile that can be passed into GMT later on.

Parameters

- **`check_kind`** (*`str`*) – Used to validate the type of data that can be passed in. Choose from 'raster', 'vector' or None. Default is None (no validation).

- **data** (*str or pathlib.Path or xarray.DataArray or numpy.ndarray or pandas.DataFrame or xarray.Dataset or geopandas.GeoDataFrame or None*) – Any raster or vector data format. This could be a file name or path, a raster grid, a vector matrix/arrays, or other supported data input.
- **x/y/z** (*1d arrays or None*) – x, y and z columns as numpy arrays.
- **extra_arrays** (*list of 1d arrays*) – Optional. A list of numpy arrays in addition to x, y and z. All of these arrays must be of the same size as the x/y/z arrays.
- **required_z** (*bool*) – State whether the 'z' column is required.

Returns **file_context** (*contextlib._GeneratorContextManager*) – The virtual file stored inside a context manager. Access the file name of this virtualfile using `with file_context as fname: ...`.

Examples

```
>>> from pygmt.helpers import GMTTempFile
>>> import xarray as xr
>>> data = xr.Dataset(
...     coords=dict(index=[0, 1, 2]),
...     data_vars=dict(
...         x=("index", [9, 8, 7]),
...         y=("index", [6, 5, 4]),
...         z=("index", [3, 2, 1]),
...     ),
... )
>>> with Session() as ses:
...     with ses.virtualfile_from_data(
...         check_kind="vector", data=data
...     ) as fin:
...         # Send the output to a file so that we can read it
...         with GMTTempFile() as fout:
...             ses.call_module("info", fin + " ->" + fout.name)
...             print(fout.read().strip())
...
<vector memory>: N = 3 <7/9> <4/6> <1/3>
```

pygmt.clib.Session.virtualfile_from_matrix

`Session.virtualfile_from_matrix(matrix)`

Store a 2d array as a table inside a virtual file.

Use the virtual file name to pass in the data in your matrix to a GMT module.

Context manager (use in a `with` block). Yields the virtual file name that you can pass as an argument to a GMT module call. Closes the virtual file upon exit of the `with` block.

The virtual file will contain the array as a `GMT_MATRIX` pretending to be a `GMT_DATASET`.

Not meant for creating ``GMT_GRID``. The grid requires more metadata than just the data matrix. Use *pygmt.clib.Session.virtualfile_from_grid* instead.

Use this instead of creating the data container and virtual file by hand with *pygmt.clib.Session.create_data*, *pygmt.clib.Session.put_matrix*, and *pygmt.clib.Session.open_virtual_file*

The matrix must be C contiguous in memory. If it is not (e.g., it is a slice of a larger array), the array will be copied to make sure it is.

Parameters **`matrix`** (*`2d array`*) – The matrix that will be included in the GMT data container.

Yields **fname** (*str*) – The name of virtual file. Pass this as a file name argument to a GMT module.

Examples

```
>>> from pygmt.helpers import GMTTempFile
>>> import numpy as np
>>> data = np.arange(12).reshape((4, 3))
>>> print(data)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>> with Session() as ses:
...     with ses.virtualfile_from_matrix(data) as fin:
...         # Send the output to a file so that we can read it
...         with GMTTempFile() as fout:
...             ses.call_module("info", f"{fin} ->{fout.name}")
...             print(fout.read().strip())
...
<matrix memory>: N = 4 <0/9> <1/10> <2/11>
```

pygmt.clib.Session.virtualfile_from_vectors

`Session.`**`virtualfile_from_vectors`**`(`**vectors*`)`

Store 1d arrays as columns of a table inside a virtual file.

Use the virtual file name to pass in the data in your vectors to a GMT module.

Context manager (use in a `with` block). Yields the virtual file name that you can pass as an argument to a GMT module call. Closes the virtual file upon exit of the `with` block.

Use this instead of creating the data container and virtual file by hand with *`pygmt.clib.Session.create_data`*, *`pygmt.clib.Session.put_vector`*, and *`pygmt.clib.Session.open_virtual_file`*.

If the arrays are C contiguous blocks of memory, they will be passed without copying to GMT. If they are not (e.g., they are columns of a 2D array), they will need to be copied to a contiguous block.

Parameters **`vectors`** (*`1d arrays`*) – The vectors that will be included in the array. All must be of the same size.

Yields **fname** (*str*) – The name of virtual file. Pass this as a file name argument to a GMT module.

Examples

```
>>> from pygmt.helpers import GMTTempFile
>>> import numpy as np
>>> import pandas as pd
>>> x = [1, 2, 3]
>>> y = np.array([4, 5, 6])
>>> z = pd.Series([7, 8, 9])
>>> with Session() as ses:
...     with ses.virtualfile_from_vectors(x, y, z) as fin:
...         # Send the output to a file so that we can read it
...         with GMTTempFile() as fout:
...             ses.call_module("info", f"{fin} ->{fout.name}")
...             print(fout.read().strip())
...
<vector memory>: N = 3 <1/3> <4/6> <7/9>
```

pygmt.clib.Session.virtualfile_from_grid

`Session.`**`virtualfile_from_grid`**(*grid*)

Store a grid in a virtual file.

Use the virtual file name to pass in the data in your grid to a GMT module. Grids must be `xarray.DataArray` instances.

Context manager (use in a `with` block). Yields the virtual file name that you can pass as an argument to a GMT module call. Closes the virtual file upon exit of the `with` block.

The virtual file will contain the grid as a `GMT_MATRIX` with extra metadata.

Use this instead of creating a data container and virtual file by hand with *`pygmt.clib.Session.create_data`*, *`pygmt.clib.Session.put_matrix`*, and *`pygmt.clib.Session.open_virtual_file`*

The grid data matrix must be C contiguous in memory. If it is not (e.g., it is a slice of a larger array), the array will be copied to make sure it is.

Parameters **`grid`** (`xarray.DataArray`) – The grid that will be included in the virtual file.

Yields **fname** (*str*) – The name of virtual file. Pass this as a file name argument to a GMT module.

Examples

```
>>> from pygmt.datasets import load_earth_relief
>>> from pygmt.helpers import GMTTempFile
>>> data = load_earth_relief(resolution="01d")
>>> print(data.shape)
(180, 360)
>>> print(data.lon.values.min(), data.lon.values.max())
-179.5 179.5
>>> print(data.lat.values.min(), data.lat.values.max())
-89.5 89.5
>>> print(data.values.min(), data.values.max())
-8182.0 5651.5
>>> with Session() as ses:
```

(continues on next page)

(continued from previous page)

```
...     with ses.virtualfile_from_grid(data) as fin:
...         # Send the output to a file so that we can read it
...         with GMTTempFile() as fout:
...             args = f"{fin} -L0 -Cn ->{fout.name}"
...             ses.call_module("grdinfo", args)
...             print(fout.read().strip())
...
-180 180 -90 90 -8182 5651.5 1 1 360 180 1 1
>>> # The output is: w e s n z0 z1 dx dy n_columns n_rows reg gtype
```

Low level access (these are mostly used by the `pygmt.clib` package):

clib.Session.create(name)	Create a new GMT C API session.
clib.Session.destroy()	Destroy the currently open GMT API session.
clib.Session.__getitem__(name)	Get the value of a GMT constant (C enum) from gmt_resources.h.
clib.Session.__enter__()	Create a GMT API session and check the libgmt version.
clib.Session.__exit__(exc_type, exc_value, ...)	Destroy the currently open GMT API session.
clib.Session.get_default(name)	Get the value of a GMT default parameter (library version, paths, etc).
clib.Session.create_data(family, geometry, ...)	Create an empty GMT data container.
clib.Session.put_matrix(dataset, matrix[, pad])	Attach a numpy 2D array to a GMT dataset.
clib.Session.put_strings(dataset, family, ...)	Attach a numpy 1D array of dtype str as a column on a GMT dataset.
clib.Session.put_vector(dataset, column, vector)	Attach a numpy 1D array as a column on a GMT dataset.
clib.Session.write_data(family, geometry, ...)	Write a GMT data container to a file.
clib.Session.open_virtual_file(family, ...)	Open a GMT Virtual File to pass data to and from a module.
clib.Session.extract_region()	Extract the WESN bounding box of the currently active figure.
clib.Session.get_libgmt_func(name[, ...])	Get a ctypes function from the libgmt shared library.

pygmt.clib.Session.create

`Session.`**`create`**`(`*name*`)`

Create a new GMT C API session.

This is required before most other methods of *pygmt.clib.Session* can be called.

> **Warning:** Usage of *pygmt.clib.Session* as a context manager in a `with` block is preferred over calling *pygmt.clib.Session.create* and *pygmt.clib.Session.destroy* manually.

Calls `GMT_Create_Session` and generates a new `GMTAPI_CTRL` struct, which is a `ctypes.c_void_p` pointer. Sets the `session_pointer` attribute to this pointer.

Remember to terminate the current session using *pygmt.clib.Session.destroy* before creating a new one.

Parameters **name** (*str*) – A name for this session. Doesn't really affect the outcome.

pygmt.clib.Session.destroy

`Session.`**`destroy`**`()`

Destroy the currently open GMT API session.

> **Warning:** Usage of *`pygmt.clib.Session`* as a context manager in a `with` block is preferred over calling *`pygmt.clib.Session.create`* and *`pygmt.clib.Session.destroy`* manually.

Calls `GMT_Destroy_Session` to terminate and free the memory of a registered `GMTAPI_CTRL` session (the pointer for this struct is stored in the `session_pointer` attribute).

Always use this method after you are done using a C API session. The session needs to be destroyed before creating a new one. Otherwise, some of the configuration files might be left behind and can influence subsequent API calls.

Sets the `session_pointer` attribute to `None`.

pygmt.clib.Session.__getitem__

`Session.`**`__getitem__`**`(`*name*`)`

Get the value of a GMT constant (C enum) from gmt_resources.h.

Used to set configuration values for other API calls. Wraps `GMT_Get_Enum`.

Parameters **`name`** (*`str`*) – The name of the constant (e.g., `"GMT_SESSION_EXTERNAL"`)

Returns **constant** (*int*) – Integer value of the constant. Do not rely on this value because it might change.

Raises ***`GMTCLibError`*** – If the constant doesn't exist.

pygmt.clib.Session.__enter__

`Session.`**`__enter__`**`()`

Create a GMT API session and check the libgmt version.

Calls *`pygmt.clib.Session.create`*.

Raises ***`GMTVersionError`*** – If the version reported by libgmt is less than `Session.required_version`. Will destroy the session before raising the exception.

pygmt.clib.Session.__exit__

`Session.`**`__exit__`**`(`*exc_type*, *exc_value*, *traceback*`)`

Destroy the currently open GMT API session.

Calls *`pygmt.clib.Session.destroy`*.

pygmt.clib.Session.get_default

`Session.get_default(name)`

Get the value of a GMT default parameter (library version, paths, etc).

Possible default parameter names include:

- `"API_VERSION"`: The GMT version
- `"API_PAD"`: The grid padding setting
- `"API_BINDIR"`: The binary file directory
- `"API_SHAREDIR"`: The share directory
- `"API_DATADIR"`: The data directory
- `"API_PLUGINDIR"`: The plugin directory
- `"API_LIBRARY"`: The core library path
- `"API_CORES"`: The number of cores
- `"API_IMAGE_LAYOUT"`: The image/band layout
- `"API_GRID_LAYOUT"`: The grid layout

Parameters **name** (*str*) – The name of the default parameter (e.g., `"API_VERSION"`)

Returns **value** (*str*) – The default value for the parameter.

Raises ***GMTCLibError*** – If the parameter doesn't exist.

pygmt.clib.Session.create_data

`Session.create_data(family, geometry, mode, **kwargs)`

Create an empty GMT data container.

Parameters

- **family** (*str*) – A valid GMT data family name (e.g., `'GMT_IS_DATASET'`). See the `FAMILIES` attribute for valid names.
- **geometry** (*str*) – A valid GMT data geometry name (e.g., `'GMT_IS_POINT'`). See the `GEOMETRIES` attribute for valid names.
- **mode** (*str*) – A valid GMT data mode (e.g., `'GMT_IS_OUTPUT'`). See the `MODES` attribute for valid names.
- **dim** (*list of 4 integers*) – The dimensions of the dataset. See the documentation for the GMT C API function `GMT_Create_Data` (`src/gmt_api.c`) for the full range of options regarding 'dim'. If `None`, will pass in the NULL pointer.
- **ranges** (*list of 4 floats*) – The dataset extent. Also a bit of a complicated argument. See the C function documentation. It's called `range` in the C function but it would conflict with the Python built-in `range` function.
- **inc** (*list of 2 floats*) – The increments between points of the dataset. See the C function documentation.
- **registration** (*str*) – The node registration (what the coordinates mean). Can be `'GMT_GRID_PIXEL_REG'` or `'GMT_GRID_NODE_REG'`. Defaults to `'GMT_GRID_NODE_REG'`.

- **pad** (*int*) – The grid padding. Defaults to `GMT_PAD_DEFAULT`.

Returns **data_ptr** (*int*) – A ctypes pointer (an integer) to the allocated `GMT_Dataset` object.

pygmt.clib.Session.put_matrix

`Session.put_matrix`(*dataset*, *matrix*, *pad=0*)

Attach a numpy 2D array to a GMT dataset.

Use this function to attach numpy array data to a GMT dataset and pass it to GMT modules. Wraps `GMT_Put_Matrix`.

The dataset must be created by *pygmt.clib.Session.create_data* first. Use `|GMT_VIA_MATRIX'` in the family.

Not at all numpy dtypes are supported, only: float64, float32, int64, int32, uint64, and uint32.

Warning: The numpy array must be C contiguous in memory. Use `numpy.ascontiguousarray` to make sure your vector is contiguous (it won't copy if it already is).

Parameters

- **dataset** (`ctypes.c_void_p`) – The ctypes void pointer to a `GMT_Dataset`. Create it with *pygmt.clib.Session.create_data*.
- **matrix** (*numpy 2d-array*) – The array that will be attached to the dataset. Must be a 2d C contiguous array.
- **pad** (*int*) – The amount of padding that should be added to the matrix. Use when creating grids for modules that require padding.

Raises ***GMTCLibError*** – If given invalid input or `GMT_Put_Matrix` exits with status != 0.

pygmt.clib.Session.put_strings

`Session.put_strings`(*dataset*, *family*, *strings*)

Attach a numpy 1D array of dtype str as a column on a GMT dataset.

Use this function to attach string type numpy array data to a GMT dataset and pass it to GMT modules. Wraps `GMT_Put_Strings`.

The dataset must be created by *pygmt.clib.Session.create_data* first.

Warning: The numpy array must be C contiguous in memory. If it comes from a column slice of a 2d array, for example, you will have to make a copy. Use `numpy.ascontiguousarray` to make sure your vector is contiguous (it won't copy if it already is).

Parameters

- **dataset** (`ctypes.c_void_p`) – The ctypes void pointer to a `GMT_Dataset`. Create it with *pygmt.clib.Session.create_data*.
- **family** (*str*) – The family type of the dataset. Can be either `GMT_IS_VECTOR` or `GMT_IS_MATRIX`.

- **strings** (*numpy 1d-array*) – The array that will be attached to the dataset. Must be a 1d C contiguous array.

Raises ***GMTCLibError*** – If given invalid input or `GMT_Put_Strings` exits with status != 0.

pygmt.clib.Session.put_vector

`Session.`**`put_vector`**(*dataset*, *column*, *vector*)

Attach a numpy 1D array as a column on a GMT dataset.

Use this function to attach numpy array data to a GMT dataset and pass it to GMT modules. Wraps `GMT_Put_Vector`.

The dataset must be created by *`pygmt.clib.Session.create_data`* first. Use `family='GMT_IS_DATASET|GMT_VIA_VECTOR'`.

Not at all numpy dtypes are supported, only: float64, float32, int64, int32, uint64, uint32, datetime64 and str_.

> **Warning:** The numpy array must be C contiguous in memory. If it comes from a column slice of a 2d array, for example, you will have to make a copy. Use `numpy.ascontiguousarray` to make sure your vector is contiguous (it won't copy if it already is).

Parameters

- **dataset** (`ctypes.c_void_p`) – The ctypes void pointer to a `GMT_Dataset`. Create it with *`pygmt.clib.Session.create_data`*.
- **column** (*`int`*) – The column number of this vector in the dataset (starting from 0).
- **vector** (*numpy 1d-array*) – The array that will be attached to the dataset. Must be a 1d C contiguous array.

Raises ***GMTCLibError*** – If given invalid input or `GMT_Put_Vector` exits with status != 0.

pygmt.clib.Session.write_data

`Session.`**`write_data`**(*family*, *geometry*, *mode*, *wesn*, *output*, *data*)

Write a GMT data container to a file.

The data container should be created by *`pygmt.clib.Session.create_data`*.

Wraps `GMT_Write_Data` but only allows writing to a file. So the `method` argument is omitted.

Parameters

- **family** (*`str`*) – A valid GMT data family name (e.g., `'GMT_IS_DATASET'`). See the `FAMILIES` attribute for valid names. Don't use the `GMT_VIA_VECTOR` or `GMT_VIA_MATRIX` constructs for this. Use `GMT_IS_VECTOR` and `GMT_IS_MATRIX` instead.
- **geometry** (*`str`*) – A valid GMT data geometry name (e.g., `'GMT_IS_POINT'`). See the `GEOMETRIES` attribute for valid names.
- **mode** (*`str`*) – How the data is to be written to the file. This option varies depending on the given family. See the GMT API documentation for details.
- **wesn** (*`list` or numpy array*) – [xmin, xmax, ymin, ymax, zmin, zmax] of the data. Must have 6 elements.
- **output** (*`str`*) – The output file name.

- **data** (ctypes.c_void_p) – Pointer to the data container created by *pygmt.clib.Session.create_data*.

Raises *GMTCLibError* – For invalid input arguments or if the GMT API functions returns a non-zero status code.

pygmt.clib.Session.open_virtual_file

Session.**open_virtual_file**(*family*, *geometry*, *direction*, *data*)

Open a GMT Virtual File to pass data to and from a module.

GMT uses a virtual file scheme to pass in data to API modules. Use it to pass in your GMT data structure (created using *pygmt.clib.Session.create_data*) to a module that expects an input or output file.

Use in a `with` block. Will automatically close the virtual file when leaving the `with` block. Because of this, no wrapper for `GMT_Close_VirtualFile` is provided.

Parameters

- **family** (*str*) – A valid GMT data family name (e.g., `'GMT_IS_DATASET'`). Should be the same as the one you used to create your data structure.
- **geometry** (*str*) – A valid GMT data geometry name (e.g., `'GMT_IS_POINT'`). Should be the same as the one you used to create your data structure.
- **direction** (*str*) – Either `'GMT_IN'` or `'GMT_OUT'` to indicate if passing data to GMT or getting it out of GMT, respectively. By default, GMT can modify the data you pass in. Add modifier `'GMT_IS_REFERENCE'` to tell GMT the data are read-only, or `'GMT_IS_DUPLICATE'` to tell GMT to duplicate the data.
- **data** (*int*) – The ctypes void pointer to your GMT data structure.

Yields **vfname** (*str*) – The name of the virtual file that you can pass to a GMT module.

Examples

```
>>> from pygmt.helpers import GMTTempFile
>>> import os
>>> import numpy as np
>>> x = np.array([0, 1, 2, 3, 4])
>>> y = np.array([5, 6, 7, 8, 9])
>>> with Session() as lib:
...     family = "GMT_IS_DATASET|GMT_VIA_VECTOR"
...     geometry = "GMT_IS_POINT"
...     dataset = lib.create_data(
...         family=family,
...         geometry=geometry,
...         mode="GMT_CONTAINER_ONLY",
...         dim=[2, 5, 1, 0],  # columns, lines, segments, type
...     )
...     lib.put_vector(dataset, column=0, vector=x)
...     lib.put_vector(dataset, column=1, vector=y)
...     # Add the dataset to a virtual file
...     vfargs = (family, geometry, "GMT_IN|GMT_IS_REFERENCE", dataset)
...     with lib.open_virtual_file(*vfargs) as vfile:
...         # Send the output to a temp file so that we can read it
```

(continues on next page)

(continued from previous page)

```
...         with GMTTempFile() as ofile:
...             args = f"{vfile} ->{ofile.name}"
...             lib.call_module("info", args)
...             print(ofile.read().strip())
...
<vector memory>: N = 5 <0/4> <5/9>
```

pygmt.clib.Session.extract_region

Session.**extract_region**()

Extract the WESN bounding box of the currently active figure.

Retrieves the information from the PostScript file, so it works for country codes as well.

Returns * **wesn** (*1d array*) – A 1D numpy array with the west, east, south, and north dimensions of the current figure.

Examples

```
>>> import pygmt
>>> fig = pygmt.Figure()
>>> fig.coast(
...     region=[0, 10, -20, -10],
...     projection="M6i",
...     frame=True,
...     land="black",
... )
>>> with Session() as lib:
...     wesn = lib.extract_region()
...
>>> print(", ".join([f"{x:.2f}" for x in wesn]))
0.00, 10.00, -20.00, -10.00
```

Using ISO country codes for the regions (for example `'US.HI'` for Hawaii):

```
>>> fig = pygmt.Figure()
>>> fig.coast(
...     region="US.HI", projection="M6i", frame=True, land="black"
... )
>>> with Session() as lib:
...     wesn = lib.extract_region()
...
>>> print(", ".join([f"{x:.2f}" for x in wesn]))
-164.71, -154.81, 18.91, 23.58
```

The country codes can have an extra argument that rounds the region a multiple of the argument (for example, `'US.HI+r5'` will round the region to multiples of 5):

```
>>> fig = pygmt.Figure()
>>> fig.coast(
...     region="US.HI+r5", projection="M6i", frame=True, land="black"
```

(continues on next page)

(continued from previous page)

```
... )
>>> with Session() as lib:
...     wesn = lib.extract_region()
...
>>> print(", ".join([f"{x:.2f}" for x in wesn]))
-165.00, -150.00, 15.00, 25.00
```

pygmt.clib.Session.get_libgmt_func

`Session.`**`get_libgmt_func`**(*name*, *argtypes=None*, *restype=None*)

Get a ctypes function from the libgmt shared library.

Assigns the argument and return type conversions for the function.

Use this method to access a C function from libgmt.

Parameters

- **name** (*str*) – The name of the GMT API function.
- **argtypes** (*list*) – List of ctypes types used to convert the Python input arguments for the API function.
- **restype** (*ctypes type*) – The ctypes type used to convert the input returned by the function into a Python type.

Returns *function* – The GMT API function.

Examples

```
>>> from ctypes import c_void_p, c_int
>>> with Session() as lib:
...     func = lib.get_libgmt_func(
...         "GMT_Destroy_Session", argtypes=[c_void_p], restype=c_int
...     )
...
>>> type(func)
<class 'ctypes.CDLL.__init__.<locals>._FuncPtr'>
```

9.22 Changelog

9.22.1 Release v0.5.0 (2021/10/29)

Highlights

- **Fifth minor release of PyGMT**
- Wrapped 12 GMT modules
- Standardized and reorder table inputs to be ‘data, x, y, z’ across functions (#1479)
- Added a gallery example showing usage of line objects from a geopandas.GeoDataFrame (#1474)

New Features

- Wrap blockmode (#1456)
- Wrap gmtselect (#1429)
- Wrap grd2xyz (#1284)
- Wrap grdproject (#1377)
- Wrap grdsample (#1380)
- Wrap grdvolume (#1299)
- Wrap nearneighbor (#1379)
- Wrap project (#1122)
- Wrap sph2grd (#1434)
- Wrap sphdistance (#1383)
- Wrap sphinterpolate (#1418)
- Wrap xyz2grd (#636)
- Add function to import seafloor crustal age dataset (#1471)
- Add pygmt.load_dataarray function (#1439)

Enhancements

- Expand table-like input options for Figure.contour (#1531)
- Expand table-like input options for pygmt.surface (#1455)
- Raise GMTInvalidInput exception when required z is missing (#1478)
- Add support for passing pathlib.Path objects as filenames (#1382)
- Allow passing a list to the ‘incols’ parameter for blockm, grdtrack and text (#1475)
- Plot square or cube by default for OGR/GMT files with Point/MultiPoint types (#1438)
- Plot square or cube by default for geopandas Point/MultiPoint types (#1405)
- Add area_thresh to COMMON_OPTIONS (#1426)
- Add function to import Mars dataset (#1420)
- Add function to import hotspot dataset (#1386)

Deprecations

- pygmt.blockm*: Reorder input parameters to 'data, x, y, z' (#1565)
- pygmt.surface: Reorder input parameters to 'data, x, y, z' (#1562)
- Figure.contour: Reorder input parameters to 'data, x, y, z' (#1561)
- Figure.plot3d: Reorder input parameters to 'data, x, y, z' (#1560)
- Figure.plot: Reorder input parameters to "data, x, y" (#1547)
- Figure.rose: Reorder input parameters to 'data, length, azimuth' (#1546)
- Figure.wiggle: Reorder input parameter to 'data, x, y, z' (#1548)
- Figure.histogram: Deprecate parameter "table" to "data" (remove in v0.7.0) (#1540)
- pygmt.info: Deprecate parameter "table" to "data" (remove in v0.7.0) (#1538)
- Figure.wiggle: Deprecate parameter "columns" to "incols" (remove in v0.7.0) (#1504)
- pygmt.surface: Deprecate parameter "outfile" to "outgrid" (remove in v0.7.0) (#1458)
- NEP29: Set minimum required version to NumPy 1.18+ (#1430)

Bug Fixes

- Allow GMTDataArrayAccessor to work on sliced datacubes (#1581)
- Allow non-string color when input data is a matrix or a file for plot and plot3d (#1526)
- Raise RuntimeWarning instead of an exception for irregular grid spacing (#1530)
- Raise an error for zero increment grid (#1484)

Documentation

- Add CITATION.cff file for PyGMT (#1592)
- Update region and projection standard docstrings (#1510)
- Document gmtwhich -Ga option to download to appropriate cache folder (#1554)
- Add gallery example showing the usage of text symbols (#1522)
- Add gallery example for grdgradient (#1428)
- Add gallery example for grdlandmask (#1469)
- Add missing aliases to pygmt.grdgradient (#1515)
- Add missing aliases to pygmt.sphdistance (#1516)
- Add missing aliases to pygmt.blockmean and pygmt.blockmedian (#1500)
- Add missing aliases to pygmt.Figure.wiggle (#1498)
- Add missing aliases to pygmt.Figure.velo (#1497)
- Add missing aliases to pygmt.surface (#1501)
- Add missing aliases to pygmt.Figure.plot3d (#1503)
- Add missing aliases to pygmt.grdlandmask (#1423)

- Add missing aliases to pygmt.grdtrack (#1499)
- Add missing aliases to pygmt.Figure.plot (#1502)
- Add missing aliases to pygmt.Figure.text (#1448)
- Add missing aliases to pygmt.Figure.histogram (#1451)
- Add missing alias to pygmt.Figure.legend (#1453)
- Add missing aliases to pygmt.Figure.rose (#1452)
- Add missing alias to pygmt.Figure.grdview (#1450)
- Add missing aliases to pygmt.Figure.image.py (#1449)
- Add missing common options to contour (#1446)
- Add missing 'incols' alias to info (#1476)

Maintenance

- Add support for Python 3.10 (#1591)
- Make IPython partially optional on CI to increase test coverage of figure.py (#1496)
- Use mamba to install Continuous Integration dependencies (#841)
- Remove deprecated codecov dependency from CI (#1494)
- Add the use of Flake8 to check examples and fix warnings (#1477)

Contributors

- Dongdong Tian
- Michael Grund
- Will Schlitzer
- Wei Ji Leong
- Meghan Jones
- Yohai Magen
- Amanda Leaman
- @daroari
- @obaney
- @srijac
- Andrés Ignacio Torres
- Becky Salvage
- Claudio Satriano
- Jamie J Quinn
- @carocamargo

9.22.2 Release v0.4.1 (2021/08/07)

Highlights

- **Patch release with multiple gallery examples**
- Change default GitHub branch name from "master" to "main" to increase inclusivity (#1360)
- Add a "PyGMT Team" page (#1308)

Enhancements

- Add common alias "verbose" (V) to grdlandmask and savefig (#1343)

Bug Fixes

- Change invalid input conditions in grdtrack (#1376)
- Fix bug so that x2sys_cross accepts dataframes with NaN values (#1369)

Documentation

- Combine documentation and compatibility sections in README.rst (#1415)
- Add a gallery example for grdclip (#1396)
- Add a gallery example for different colormaps in subplots (#1394)
- Add a gallery example for the contour method (#1387)
- Add a gallery example showing individual custom symbols (#1348)
- Add common option aliases to COMMON_OPTIONS in decorators.py (#1407)
- Add return statement to grdclip and grdgradient docstring (#1390)
- Restructure contributing.md to separate docs/general info from contributing code section (#1339)

Maintenance

- Add tomli as a dependency in GMT Dev Tests (#1401)
- NEP29: Test PyGMT on NumPy 1.21 (#1355)

Contributors

- Meghan Jones
- Will Schlitzer
- Michael Grund
- Wei Ji Leong
- Yohai Magen
- Jiayuan Yao
- Dongdong Tian
- Kadatatlu Kishore
- @sean0921
- Soham Banerjee

9.22.3 Release v0.4.0 (2021/06/20)

Highlights

- **Fourth minor release of PyGMT**
- Add tutorials for datetime data (#1193) and plotting vectors (#1070)
- Support tab auto-completion in Jupyter (#1282)
- Minimum required GMT version is now 6.2.0 or newer (#1321)

New Features

- Wrap blockmean (#1092)
- Wrap grdclip (#1261)
- Wrap grdfill (#1276)
- Wrap grdgradient (#1269)
- Wrap grdlandmask (#1273)
- Wrap histogram (#1072)
- Wrap rose (#794)
- Wrap solar (#804)
- Wrap velo (#525)
- Wrap wiggle (#1145)
- Add new function to load fractures sample data (#1101)
- Allow load_earth_relief() to load the original land-only 01s or 03s SRTM tiles (#976)

- Handle geopandas and shapely geometries via geo_interface link (#1000)
- Support passing string type numbers, geographic coordinates and datetimes (#975)

Enhancements

- Allow passing an array as intensity for plot3d (#1109)
- Allow passing an array as intensity for plot (#1065)
- Allow passing xr.DataArray as shading to grdimage (#750)
- Allow x/y/z input for blockmedian and blockmean (#1319)
- Allow pygmt.which to accept a list of filenames as input (#1312)
- Refactor blockm* to use virtualfile_from_data and improve i/o (#1280)
- Refactor grdtrack to use virtualfile_from_data and improve i/o to pandas.DataFrame (#1189)
- Add parameters to histogram (#1249)
- Add alias 'aspatial' to methods blockmedian, info, plot, plot3d, surface (#1090)
- Add alias 'registration' to methods blockmean, info, grdfilter, surface (#1089)
- Add incols to COMMON_OPTIONS, blockmean, and blockmedian (#1300)
- Improve Figure.show for displaying previews in Jupyter notebooks and external viewers (#529)
- Let Figure.savefig recommend .eps or .pdf when .ps extension is used (#1307)

Deprecations

- Figure.contour: Deprecate parameter "columns" to "incols" (remove in v0.6.0) (#1303)
- Figure.plot: Deprecate parameter "sizes" to "size" (remove in v0.6.0) (#1254)
- Figure.plot: Deprecate parameter "columns" to "incols" (remove in v0.6.0) (#1298)
- Figure.plot3d: Deprecate parameter "sizes" to "size" (remove in v0.6.0) (#1258)
- Figure.plot3d: Deprecate parameter "columns" to "incols" (remove in v0.6.0) (#1040)
- Figure.rose: Deprecate parameter "columns" to "incols" (remove in v0.6.0) (#1306)
- NEP29: Set minimum required versions to NumPy 1.17+ and Python 3.7+ (#1074)
- Raise a warning for the use of short-form parameters when long-forms are available (#1316)

Bug Fixes

- Allow pandas.Series inputs to fig.histogram and pygmt.info (#1329)
- Explicitly use netcdf4 engine in xarray.open_dataarray to read grd files (#1264)
- Let Figure.savefig support filenames with spaces (#1116)
- Let Figure.show(method='external') work well in Python scripts (#1062)

Documentation

- Add histogram gallery example (#1272)
- Add a gallery example showing individual basic geometric symbols (#1211)
- Specify rectangle's width and height via style parameter in multi-parameter symbols example (#1325)
- Update the inset gallery example (#1287)
- Add categorical colorbars for plot, plot3d and line colors gallery examples (#1267)
- Apply NIST SI unit convention to some gallery examples (#1194)
- Use colorblind-friendly colors in the scatter plots gallery example (#1013)
- Added documentation for three oblique mercator projections (#1251)
- Add a list of external PyGMT resources (#1210)
- Complete documentation for grdtrack (#1190)
- Add projection and region to grdview docstring (#1295)
- Add common alias spacing (-I) for specifying grid increments (#1288)
- Standardize docstrings for table-like inputs (#1186)
- Clarify that the "transparency" parameter in plot/plot3d/text can be 1d array (#1265)
- Clarify that the "color" parameter in plot/plot3d can be 1d array (#1260)
- Clarify interplay of spacing and per_column in info (#1127)
- Remove the "full test" section from installation guide (#1206)
- Clarify position of deprecate_parameter decorator to be above use_alias (#1302)
- Add guidelines for managing issues to maintenance.md (#1301)
- Add alias name convention to CONTRIBUTING.md (#1256)
- Move contributing guide details to website and rename two sections (#1335)
- Update the check_figures_equal testing section in CONTRIBUTING.md (#1108)
- Revise Pull Request review process in CONTRIBUTING.md (#1119)

Maintenance

- Add a workflow to upload baseline images as a release asset (#1317)
- Add regression test for grdimage plotting an xarray.DataArray grid subset (#1314)
- Add download_test_data to download data files used in tests (#1310)
- Remove xfails and workarounds for datetime inputs into pygmt.info (#1236)
- Improve the DVC image diff workflow to support side-by-side comparison of modified images (#1219)
- Document the deprecation policy and add the deprecate_parameter decorator to deprecate parameters (#1160)
- Convert booleans arguments in build_arg_string, not in kwargs_to_strings (#1125)
- Create Github Action workflow for reporting DVC image diffs (#1104)
- Update "GMT Dev Tests" workflow to test macOS-11.0 and pre-release Python packages (#1105)
- Initialize data version control for managing test images (#1036)

- Separate workflows for running tests and building documentation (#1033)

Contributors

- Dongdong Tian
- Wei Ji Leong
- Michael Grund
- Meghan Jones
- Will Schlitzer
- Jiayuan Yao
- Abhishek Anant
- Claire Klima
- Megan Munzek
- Michael Neumann
- Nathan Loria
- Noor Buchi
- Shivani chauhan
- @alperen-kilic
- Loïc Houpert
- Emily McMullan
- Lawrence Qupty
- Matthew Tankersley
- @shahid-0
- Vitor Gratiere Torres

9.22.4 Release v0.3.1 (2021/03/14)

Highlights

- **Multiple bug fixes and an improved gallery**
- Reorganized gallery examples into new categories (#995)
- Added gallery examples for plotting vectors (#950, #890)
- Last version to support GMT 6.1.1, future PyGMT versions will require GMT 6.2.0 or newer

Enhancements

- Support passing a sequence to the spacing parameter of pygmt.info() (#1031)

Bug Fixes

- Fix issues in loading GMT's shared library (#977)
- Let pygmt.info load datetime columns into a str dtype array (#960)
- Check invalid combinations of resolution and registration in load_earth_relief() (#965)
- Open figures using the associated application on Windows (#952)
- Fix bug that stops Figure.coast from plotting with only dcw parameter (#910)

Documentation

- Add a gallery example showing different line front styles (#1022)
- Add a gallery example for a double y-axes graph (#1019)
- Add a gallery example of inset map showing a rectangle region (#1020)
- Add a gallery example to show coloring of points by categories (#1006)
- Add gallery example showing different polar projection use cases (#955)
- Add underscore guideline to CONTRIBUTING.md (#1034)
- Add instructions to upgrade installed PyGMT version (#1029)
- Improve the docstring of the pygmt package (#1016)
- Add common alias coltypes (-f) for specifying i/o data types (#994)
- Expand documentation linking in CONTRIBUTING.md (#802)
- Write changelog in markdown using MyST (#941)
- Update web font to Atkinson Hyperlegible (#938)
- Improve the gallery example for datetime inputs (#919)

Maintenance

- Refactor plot and plot3d to use virtualfile_from_data (#990)
- Explicitly exclude unnecessary files in source distributions (#999)
- Refactor grd modules to use virtualfile_from_data (#992)
- Refactor info and grdinfo to use virtualfile_from_data (#961)
- Onboarding maintainer checklist (#773)
- Add comprehensive tests for pygmt.clib.loading.clib_full_names (#872)
- Add a workflow checking links in plaintext and HTML files (#634)
- Remove nbsphinx extension (#931)
- Improve the error message for loading an old version of the GMT library (#925)

- Move requirements-dev.txt dependencies to environment.yml (#812)
- Ensure proper non-dev version string when publishing to PyPI (#900)
- Run tests in a single CI job (Ubuntu + Python 3.9) for draft PRs (#906)

Contributors

- Dongdong Tian
- Jiayuan Yao
- Wei Ji Leong
- Meghan Jones
- Michael Grund
- Will Schlitzer
- Liam Toney
- Kathryn Materna
- Alicia Ngoc Diep Ha
- Tawanda Moyo

9.22.5 Release v0.3.0 (2021/02/15)

Highlights

- **Third minor release of PyGMT**
- Wrap inset (#788) for making overview maps and subplot (#822) for multi-panel figures
- Apply standardized formatting conventions (#775) across most documentation pages
- Drop Python 3.6 support (#699) so PyGMT now requires Python 3.7 or newer

New Features

- Wrap grd2cpt (#803)
- Let Figure.text support record-by-record transparency (#716)
- Provide basic support for FreeBSD (#700, #878)

Enhancements

- Let load_earth_relief support the ‘region’ parameter for all resolutions (#873)
- Improve how PyGMT finds the GMT library (#702)
- Add common alias panel (-c) to all plotting functions (#853)
- Add aliases dcw (#765) and lakes (#781) to Figure.coast
- Add alias shading to Figure.colorbar (#752)
- Add alias annotation (A) to Figure.contour (#883)
- Wrap Figure.grdinfo aliases (#799)
- Add aliases frame and cmap to Figure.colorbar (#709)
- Add alias frame to Figure.grdview (#707)
- Improve the error message when PyGMT fails to load the GMT library (#814)
- Add GMTInvalidInput error to Figure.coast (#787)

Documentation

- Add authorship policy (#726)
- Update PyGMT development installation instructions (#865)
- Add a tutorial for adding a map title (#720)
- Add a tutorial for plotting Earth relief (#712)
- Add a tutorial for 3D perspective image (#743)
- Add a tutorial for contour maps (#705)
- Add a tutorial for plotting lines (#741)
- Add a tutorial for the region argument (#800)
- Add a gallery example for datetime inputs (#779)
- Add a gallery example for Figure.logo (#823)
- Add a gallery example for plotting multi-parameter symbols (#772)
- Add a gallery example for Figure.image (#777)
- Add a gallery example for setting line colors with a custom CPT (#774)
- Add more gallery examples for projections (#761, #721, #757, #723, #762, #742, #728, #727)
- Update the docstrings in the plotting modules (#881)
- Update the docstrings in the non-plotting modules (#882)
- Update Figure.coast docstrings (#798)
- Update the docstrings of common aliases (#862)
- Add sphinx-copybutton extension to easily copy codes (#838)
- Choose the best figures in tutorials for thumbnails (#826)
- Update axis label explanation in frames tutorial (#820)
- Add guidelines for types of tests to write (#796)

- Recommend using SI units in documentation (#795)
- Add a table for compatibility of PyGMT with Python and GMT (#763)
- Add description for the "columns" arguments (#766)
- Add a table of the available projections (#753)
- Add projection description for Lambert Azimuthal Equal-Area (#760)
- Change text when GMTInvalidInput error is raised for basemap (#729)

Bug Fixes

- Fix a bug of Figure.text when "text" is a non-string array (#724)
- Fix the error message when IPython is not available (#701)

Maintenance

- Add dependabot to keep GitHub Actions up to date (#861)
- Skip workflows in PRs if only non-source-code files are changed (#839)
- Add slash command '/test-gmt-dev' to test GMT dev version (#831)
- Check files for UNIX-style line breaks and 644 permission (#736)
- Rename vercel configuration file from now.json to vercel.json (#738)
- Add a CI job testing GMT master branch on Windows (#756)
- Migrate documentation deployment from Travis CI to GitHub Actions (#713)
- Move Figure.meca into a standalone module (#686)
- Move plotting functions to separate modules (#808)
- Move non-plotting modules to separate modules (#832)
- Add isort to sort imports alphabetically (#745)
- Convert relative imports to absolute imports (#754)
- Switch from versioneer to setuptools-scm (#695)
- Add docformatter to format plain text in docstrings (#642)
- Migrate pytest configurations to pyproject.toml (#725)
- Migrate coverage configurations to pyproject.toml (#667)
- Show test execution times in pytest (#835)
- Add tests for grdfilter (#809)
- Add tests for GMTInvalidInput of Figure.savefig and Figure.show (#810)
- Add args_in_kwargs function (#791)
- Add a Makefile target 'distclean' for deleting project metadata files (#744)
- Add a test for Figure.basemap map_scale (#739)
- Use args_in_kwargs for Figure.basemap error raising (#797)

Contributors

- Will Schlitzer
- Dongdong Tian
- Wei Ji Leong
- Michael Grund
- Liam Toney
- Meghan Jones

9.22.6 Release v0.2.1 (2020/11/14)

Highlights

- **Patch release with more tutorials and gallery examples!**
- Support Python 3.9 (#689)
- Add Liam's ROSES 2020 PyGMT talk (#643)

New Features

- Wrap plot3d (#471)
- Wrap grdfilter (#616)

Enhancements

- Allow np.object dtypes into virtualfile_from_vectors (#684)
- Let plot() accept record-by-record transparency (#626)
- Refactor info to allow datetime inputs from xarray.Dataset and pandas.DataFrame tables (#619)

Tutorials & Gallery

- Add tutorial for pygmt.config (#482)
- Add an example for different line styles (#604, #664)
- Add a gallery example for varying transparent points (#654)
- Add tutorial for pygmt.Figure.text (#480)
- Add an example for scatter plots with auto legends (#607)
- Improve colorbar gallery example (#596)

Documentation Improvements

- doc: Fix the description of grdcontour -G option (#681)
- Refresh Code of Conduct from v1.4 to v2.0 (#673)
- Add PyGMT Zenodo BibTeX entry to main README.md (#678)
- Complete most of documentation for makecpt (#676)
- Complete documentation for plot (#666)
- Add "no_clip" to plot, text, contour and meca (#661)
- Add common alias "verbose" (V) to all functions (#662)
- Improve documentation of Figure.logo() (#651)
- Add mini-galleries for methods and functions (#648)
- Complete documentation of grdimage (#620)
- Add common alias perspective (p) for plotting 3D illustrations (#627)
- Add common aliases xshift (X) and yshift (Y) (#624)
- Add common alias cores (x) for grdimage and other multi-threaded modules (#625)
- Enable switching different versions of documentation (#621)
- Add common alias transparency (-t) to all plotting functions (#614)

Bug Fixes

- Disallow passing arguments like -XNone to GMT (#639)

Maintenance

- Migrate PyPI release to GitHub Actions (#679)
- Upload artifacts showing diff images on test failure (#675)
- Add slash command "/format" to automatically format PRs (#646)
- Add instructions to run specific tests (#660)
- Add more tests for xarray grid shading (#650)
- Refactor xfail tests to avoid storing baseline images (#603)
- Add blackdoc to format Python codes in docstrings (#641)
- Check and lint sphinx configuration file doc/conf.py (#630)
- Improve Makefile to clean `__pycache__` directory recursively (#611)
- Update release process and checklist template (#602)

Contributors

- Dongdong Tian
- Wei Ji Leong
- Conor Bacon
- carocamargo

9.22.7 Release v0.2.0 (2020/09/12)

Highlights

- **Second minor release of PyGMT**
- Minimum required GMT version is now 6.1.1 or newer (#577)
- Plotting xarray grids using grdimage and grdview should not crash anymore and works for most cases (#560)
- Easier time-series plots with support for datetime-like inputs to plot (#464) and the region argument (#562)

New Features

- Wrap GMT_Put_Strings to pass str columns into GMT C API directly (#520)
- Wrap meca (#516)
- Wrap x2sys_init and x2sys_cross (#546)
- Let grdcut() accept xarray.DataArray as input (#541)
- Initialize a GMTDataArrayAccessor (#500)

Enhancements

- Allow passing in pandas dataframes to x2sys_cross (#591)
- Sensible array outputs for pygmt info (#575)
- Allow pandas.DataFrame table and 1D/2D numpy array inputs into pygmt.info (#574)
- Add auto-legend feature to grdcontour and contour (#568)
- Add common alias verbose (V) (#550)
- Let load_earth_relief() support all resolutions and optional subregion (#542)
- Allow load_earth_relief() to load pixel or gridline registered data (#509)

Documentation

- Link to try-gmt binder repository (#598)
- Improve docstring of data_kind() to include xarray grid (#588)
- Improve the documentation of Figure.shift_origin() (#536)
- Add shading to grdview gallery example (#506)

Bug Fixes

- Ensure surface and grdcut loads GMTDataArray accessor info into xarray (#539)
- Raise an error if short- and long-form arguments coexist (#537)
- Fix the grdtrack example to avoid crashes on macOS (#531)
- Properly allow for either pixel or gridline registered grids (#476)

Maintenance

- Add a test for xarray shading (#581)
- Remove expected failures on grdview tests (#589)
- Redesign check_figures_equal testing function to be more explicit (#590)
- Cut Windows CI build time in half to 15 min (#586)
- Add a test for Session.write_data() writing netCDF grids (#583)
- Add a test to make sure shift_origin does not crash (#580)
- Add testing.check_figures_equal to avoid storing baseline images (#555)
- Eliminate unnecessary jobs from Travis CI (#567) and Azure Pipelines (#513)
- Improve the workflow to test both GMT master (#485) and 6.1 branches (#554)
- Automatically cancel in-progress CI runs of old commits (#544)
- Remove the Stickler CI configuration file (#538), run style checks using GitHub Actions (#519)
- Cache GMT remote data as artifacts on GitHub (#530)
- Let pytest generate both HTML and XML coverage reports (#512)
- Run Continuous Integration tests on GitHub Actions (#475)

Contributors

- Dongdong Tian
- Wei Ji Leong
- Tyler Newton
- Liam Toney

9.22.8 Release v0.1.2 (2020/07/07)

Highlights

- Patch release in preparation for the SciPy 2020 sprint session
- Last version to support GMT 6.0, future PyGMT versions will require GMT 6.1 or newer

New Features

- Wrap grdcut (#492)
- Add show_versions() function for printing debugging information used in issue reports (#466)

Enhancements

- Change load_earth_relief()'s default resolution to 01d (#488)
- Enhance text with extra functionality and aliases (#481)

Documentation

- Add gallery example for grdview (#502)
- Turn all short aliases into long form (#474)
- Update the plotting example using the colormap generated by pygmt.makecpt (#472)
- Add instructions to view the test coverage reports locally (#468)
- Update the instructions for testing pygmt install (#459)

Bug Fixes

- Fix a bug when passing data to GMT in Session.open_virtual_file() (#490)

Maintenance

- Temporarily expect failures for some grdcontour and grdview tests (#503)
- Fix several failures due to updates of earth relief data (#498)
- Unpin pylint version and fix some lint warnings (#484)
- Separate tests of gmtinfo and grdinfo (#461)
- Fix the test for GMT_COMPATIBILITY=6 (#454)
- Update baseline images for updates of earth relief data (#452)
- Simplify PyGMT Release process (#446)

Contributors

- Dongdong Tian
- Wei Ji Leong
- Liam Toney

9.22.9 Release v0.1.1 (2020/05/22)

Highlights

- Windows users rejoice, this bugfix release is for you!
- Let PyGMT work with the conda GMT package on Windows (#434)

Enhancements

- Handle setting special parameters without default settings for config (#411)

Documentation

- Update install instructions (#430)
- Add PyGMT AGU 2019 poster to website (#425)
- Redirect www.pygmt.org to latest, instead of dev (#423)

Bug Fixes

- Set GMT_COMPATIBILITY to 6 when pygmt session starts (#432)
- Improve how PyGMT finds the GMT library (#440)

Maintenance

- Finalize fixes on Windows test suite for v0.1.1 (#441)
- Cache test data on Azure Pipelines (#438)

Contributors

- Dongdong Tian
- Wei Ji Leong
- Jason K. Moore

9.22.10 Release v0.1.0 (2020/05/03)

Highlights

- **First official release of PyGMT**
- Python 3.8 is now supported (#398)
- PyGMT now uses the stable version of GMT 6.0.0 by default (#363)
- Use sphinx-gallery to manage examples and tutorials (#268)

New Features

- Wrap blockmedian (#349)
- Add pygmt.config() to change gmt defaults locally and globally (#293)
- Wrap grdview (#330)
- Wrap grdtrack (#308)
- Wrap colorbar (#332)
- Wrap text (#321)
- Wrap legend (#333)
- Wrap makecpt (#329)
- Add a new method to shift plot origins (#289)

Enhancements

- Allow text accepting “frame” as an argument (#385)
- Allow for grids with negative lat/lon increments (#369)
- Allow passing in list to ‘region’ argument in surface (#378)
- Allow passing in scalar number to x and y in plot (#376)
- Implement default position/box for legend (#359)
- Add sequence_space converter in kwargs_to_string (#325)

Documentation

- Update PyPI install instructions and API disclaimer message (#421)
- Fix the link to GMT documentation (#419)
- Use napoleon instead of numpydoc with sphinx (#383)
- Document using a list for repeated arguments (#361)
- Add legend gallery entry (#358)
- Update instructions to set GMT_LIBRARY_PATH (#324)
- Fix the link to the GMT homepage (#331)
- Split projections gallery by projection types (#318)
- Fix the link to GMT/Matlab API in the README (#297)
- Use shinx extlinks for linking GMT docs (#294)
- Comment about country code in projection examples (#290)
- Add an overview page listing presentations (#286)

Bug Fixes

- Let surface return xr.DataArray instead of xr.Dataset (#408)
- Update GMT constant GMT_STR16 to GMT_VF_LEN for GMT API change in 6.1.0 (#397)
- Properly trigger pytest matplotlib image comparison (#352)
- Use uuid.uuid4 to generate unique names (#274)

Maintenance

- Quickfix Zeit Now miniconda installer link to anaconda.com (#413)
- Fix GitHub Pages deployment from Travis (#410)
- Update and clean TravisCI configuration (#404)
- Quickfix min elevation for new SRTM15+V2.1 earth relief grids (#401)
- Wrap docstrings to 79 chars and check with flake8 (#384)
- Update continuous integration scripts to 1.2.0 (#355)
- Use Zeit Now to deploy doc builds from PRs (#344)
- Move gmt from requirements.txt to CI scripts instead (#343)
- Change py.test to pytest (#338)
- Add Google Analytics to measure site visitors (#314)
- Register mpl_image_compare marker to remove PytestUnknownMarkWarning (#323)
- Disable Windows CI builds before PR #313 is merged (#320)
- Enable Mac and Windows CI on Azure Pipelines (#312)
- Fixes for using GMT 6.0.0rc1 (#311)

- Assign authorship to "The PyGMT Developers" (#284)

Deprecations

- Remove mention of gitter.im (#405)
- Remove portrait (-P) from common options (#339)
- Remove require.js since WorldWind was dropped (#278)
- Remove Web WorldWind support (#275)

Contributors

- Dongdong Tian
- Wei Ji Leong
- Leonardo Uieda
- Liam Toney
- Brook Tozer
- Claudio Satriano
- Cody Woodson
- Mark Wieczorek
- Philipp Loose
- Kathryn Materna

9.23 Team Gallery

We are an international team dedicated to building a Pythonic API for the Generic Mapping Tools (GMT). Our goal is to improve GMT's accessibility for new and experienced users by creating user-friendly interfaces with the GMT C API, supporting rich display in Jupyter notebooks, and integrating with the PyData ecosystem.

All are welcome to become involved with the PyGMT project! For more information about how to get involved, see the *Contributors Guide*.

9.23.1 Distinguished Contributors

PyGMT Distinguished Contributors are recognized for their substantial contributions to PyGMT, which may include code, documentation, pull request review, triaging, forum responses, community building and engagement, outreach, and inclusion and diversity. New Distinguished Contributors are selected twice per year by those listed below.

Distinguished Contributors is not meant as a means of conveying responsibilities. Distinguished Contributors who are also active maintainers of the PyGMT project and have responsibilities detailed in the *Maintainers Guide* have 'Maintainer' listed below their names.

9.24 Contributors Guide

This is a community driven project and everyone is welcome to contribute.

The project is hosted at the PyGMT GitHub repository.

The goal is to maintain a diverse community that's pleasant for everyone. **Please be considerate and respectful of others**. Everyone must abide by our Code of Conduct and we encourage all to read it carefully.

9.24.1 Ways to Contribute

Ways to Contribute Documentation and/or Code

- Tackle any issue that you wish! Some issues are labeled as **"good first issues"** to indicate that they are beginner friendly, meaning that they don't require extensive knowledge of the project.
- Make a tutorial or gallery example of how to do something.
- Improve the API documentation.
- Contribute code! This can be code that you already have and it doesn't need to be perfect! We will help you clean things up, test it, etc.

Ways to Contribute Feedback

- Provide feedback about how we can improve the project or about your particular use case. Open an issue with feature requests or bug fixes, or post general comments/questions on the forum.
- Help triage issues, or give a "thumbs up" on issues that others reported which are relevant to you.

Ways to Contribute to Community Building

- Participate and answer questions on the PyGMT forum Q&A.
- Participate in discussions at the quarterly PyGMT Community Meetings, which are announced on the forum governance page.
- Cite PyGMT when using the project.
- Spread the word about PyGMT or star the project!

9.24.2 Providing Feedback

Reporting a Bug

- Find the *Issues* tab on the top of the GitHub repository and click *New Issue*.
- Click on *Get started* next to *Bug report*.
- **Please try to fill out the template with as much detail as you can**.
- After submitting your bug report, try to answer any follow up questions about the bug as best as you can.

Submitting a Feature Request

- Find the *Issues* tab on the top of the GitHub repository and click *New Issue*.
- Click on *Get started* next to *Feature request*.
- **Please try to fill out the template with as much detail as you can**.
- After submitting your feature request, try to answer any follow up questions as best as you can.

Submitting General Comments/Questions

There are several pages on the Community Forum where you can submit general comments and/or questions:

- For questions about using PyGMT, select *New Topic* from the PyGMT Q&A Page.
- For general comments, select *New Topic* from the Lounge Page.
- To share your work, select *New Topic* from the Showcase Page.

9.24.3 General Guidelines

Resources for New Contributors

Please take a look at these resources to learn about Git and pull requests (don't hesitate to *ask questions*):

- How to Contribute to Open Source.
- Git Workflow Tutorial by Aaron Meurer.
- How to Contribute to an Open Source Project on GitHub.

Getting Help

Discussion often happens on GitHub issues and pull requests. In addition, there is a Discourse forum for the project where you can ask questions.

Pull Request Workflow

We follow the git pull request workflow to make changes to our codebase. Every change made goes through a pull request, even our own, so that our continuous integration services have a chance to check that the code is up to standards and passes all our tests. This way, the *main* branch is always stable.

General Guidelines for Making a Pull Request (PR):

- What should be included in a PR
  - Have a quick look at the titles of all the existing issues first. If there is already an issue that matches your PR, leave a comment there to let us know what you plan to do. Otherwise, **open an issue** describing what you want to do.
  - Each pull request should consist of a **small** and logical collection of changes; larger changes should be broken down into smaller parts and integrated separately.
  - Bug fixes should be submitted in separate PRs.

- How to write and submit a PR
  - Use underscores for all Python (*.py) files as per PEP8, not hyphens. Directory names should also use underscores instead of hyphens.
  - Describe what your PR changes and *why* this is a good thing. Be as specific as you can. The PR description is how we keep track of the changes made to the project over time.
  - Do not commit changes to files that are irrelevant to your feature or bugfix (e.g.: `.gitignore`, IDE project files, etc).
  - Write descriptive commit messages. Chris Beams has written a guide on how to write good commit messages.
- PR review
  - Be willing to accept criticism and work on improving your code; we don't want to break other users' code, so care must be taken not to introduce bugs.
  - Be aware that the pull request review process is not immediate, and is generally proportional to the size of the pull request.

General Process for Pull Request Review:

After you've submitted a pull request, you should expect to hear at least a comment within a couple of days. We may suggest some changes, improvements or alternative implementation details.

To increase the chances of getting your pull request accepted quickly, try to:

- Submit a friendly PR
  - Write a good and detailed description of what the PR does.
  - Write some documentation for your code (docstrings) and leave comments explaining the *reason* behind non-obvious things.
  - Write tests for the code you wrote/modified if needed. Please refer to *Testing your code* or *Testing plots*.
  - Include an example of new features in the gallery or tutorials. Please refer to *Gallery plots* or *Tutorials*.
- Have a good coding style
  - Use readable code, as it is better than clever code (even with comments).
  - Follow the PEP8 style guide for code and the numpy style guide for docstrings. Please refer to *Code style*.

Pull requests will automatically have tests run by GitHub Actions. This includes running both the unit tests as well as code linters. GitHub will show the status of these checks on the pull request. Try to get them all passing (green). If you have any trouble, leave a comment in the PR or *get in touch*.

9.24.4 Setting up your Environment

These steps for setting up your environment are necessary for *editing the documentation locally* and *contributing code*. A local PyGMT development environment is not needed for *editing the documentation on GitHub*.

We highly recommend using Anaconda and the `conda` package manager to install and manage your Python packages. It will make your life a lot easier!

The repository includes a conda environment file `environment.yml` with the specification for all development requirements to build and test the project. See the `environment.yml` file for the list of dependencies and the environment name (`pygmt`). Once you have forked and cloned the repository to your local machine, you can use this file to create an isolated environment on which you can work. Run the following on the base of the repository to create a new conda environment from the `environment.yml` file:

```
conda env create
```

Before building and testing the project, you have to activate the environment (you'll need to do this every time you start a new terminal):

```
conda activate pygmt
```

We have a `Makefile` that provides commands for installing, running the tests and coverage analysis, running linters, etc. If you don't want to use `make`, open the `Makefile` and copy the commands you want to run.

To install the current source code into your testing environment, run:

```
make install  # on Linux/macOS
pip install --no-deps -e .  # on Windows
```

This installs your project in *editable* mode, meaning that changes made to the source code will be available when you import the package (even if you're on a different directory).

9.24.5 Contributing Documentation

PyGMT Documentation Overview

There are four main components to PyGMT's documentation:

- Gallery examples, with source code in Python `*.py` files under the `examples/gallery/` folder.
- Tutorial examples, with source code in Python `*.py` files under the `examples/tutorials/` folder.
- API documentation, with source code in the docstrings in Python `*.py` files under the `pygmt/src/` and `pygmt/datasets/` folders.
- Getting started/developer documentation, with source text in ReST `*.rst` and markdown `*.md` files under the `doc/` folder.

The documentation are written primarily in reStructuredText and built by Sphinx. Please refer to reStructuredText Cheatsheet if you are new to reStructuredText. When contributing documentation, be sure to follow the general guidelines in the *pull request workflow* section.

There are two primary ways to edit the PyGMT documentation:

- For simple documentation changes, you can easily *edit the documentation on GitHub*. This only requires you to have a GitHub account.
- For more complicated changes, you can *edit the documentation locally*. In order to build the documentation locally, you first need to *set up your environment*.

Editing the Documentation on GitHub

If you're browsing the documentation and notice a typo or something that could be improved, please consider letting us know by *creating an issue* or (even better) submitting a fix.

You can submit fixes to the documentation pages completely online without having to download and install anything:

1. On each documentation page, there should be an "Improve This Page" link at the very top.
2. Click on that link to open the respective source file (usually an `.rst` file in the `doc/` folder or a `.py` file in the `examples/` folder) on GitHub for editing online (you'll need a GitHub account).
3. Make your desired changes.
4. When you're done, scroll to the bottom of the page.
5. Fill out the two fields under "Commit changes": the first is a short title describing your fixes; the second is a more detailed description of the changes. Try to be as detailed as possible and describe *why* you changed something.
6. Choose "Create a new branch for this commit and start a pull request" and click on the "Propose changes" button to open a pull request.
7. The pull request will run the GMT automated tests and make a preview deployment. You can see how your change looks in the PyGMT documentation by clicking the "View deployment" button after the Vercel bot has finished (usually 5-10 minutes after the pull request was created).
8. We'll review your pull request, recommend changes if necessary, and then merge them in if everything is OK.
9. Done!

Alternatively, you can make the changes offline to the files in the `doc` folder or the example scripts. See *editing the documentation locally* for instructions.

Editing the Documentation Locally

For more extensive changes, you can edit the documentation in your cloned repository and build the documentation to preview changes before submitting a pull request. First, follow the *setting up your environment* instructions. After making your changes, you can build the HTML files from sources using:

```
cd doc
make all
```

This will build the HTML files in `doc/_build/html`. Open `doc/_build/html/index.html` in your browser to view the pages. Follow the *pull request workflow* to submit your changes for review.

Contributing Gallery Plots

The gallery and tutorials are managed by sphinx-gallery. The source files for the example gallery are `.py` scripts in `examples/gallery/` that generate one or more figures. They are executed automatically by sphinx-gallery when the *documentation is built*. The output is gathered and assembled into the gallery.

You can **add a new** plot by placing a new `.py` file in one of the folders inside the `examples/gallery` folder of the repository. See the other examples to get an idea for the format.

General guidelines for making a good gallery plot:

- Examples should highlight a single feature/command. Good: *how to add a label to a colorbar*. Bad: *how to add a label to the colorbar and use two different CPTs and use subplots*.

- Try to make the example as simple as possible. Good: *use only commands that are required to show the feature you want to highlight*. Bad: *use advanced/complex Python features to make the code smaller*.
- Use a sample dataset from `pygmt.datasets` if you need to plot data. If a suitable dataset isn't available, open an issue requesting one and we'll work together to add it.
- Add comments to explain things are aren't obvious from reading the code. Good: *Use a Mercator projection and make the plot 15 centimeters wide*. Bad: *Draw coastlines and plot the data*.
- Describe the feature that you're showcasing and link to other relevant parts of the documentation.
- SI units should be used in the example code for gallery plots.

Contributing Tutorials

The tutorials (the User Guide in the docs) are also built by sphinx-gallery from the `.py` files in the `examples/tutorials` folder of the repository. To add a new tutorial:

- Include a `.py` file in the `examples/tutorials` folder on the base of the repository.
- Write the tutorial in "notebook" style with code mixed with paragraphs explaining what is being done. See the other tutorials for the format.
- Include the tutorial in the table of contents of the documentation (side bar). Do this by adding a line to the User Guide `toc` directive in `doc/index.rst`. Notice that the file included is the `.rst` generated by sphinx-gallery.
- Choose the most representative figure as the thumbnail figure by adding a comment line `# sphinx_gallery_thumbnail_number = <fig_number>` to any place (usually at the top) in the tutorial. The *fig_number* starts from 1.

Guidelines for a good tutorial:

- Each tutorial should focus on a particular set of tasks that a user might want to accomplish: plotting grids, interpolation, configuring the frame, projections, etc.
- The tutorial code should be as simple as possible. Avoid using advanced/complex Python features or abbreviations.
- Explain the options and features in as much detail as possible. The gallery has concise examples while the tutorials are detailed and full of text.
- SI units should be used in the example code for tutorial plots.

Note that the `Figure.show()` function needs to be called for a plot to be inserted into the documentation.

Editing the API Documentation

The API documentation is built from the docstrings in the Python `*.py` files under the `pygmt/src/` and `/pygmt/datasets/` folders. **All docstrings** should follow the numpy style guide. All functions/classes/methods should have docstrings with a full description of all arguments and return values.

While the maximum line length for code is automatically set by Black, docstrings must be formatted manually. To play nicely with Jupyter and IPython, **keep docstrings limited to 79 characters** per line.

Standards for Example Code

When editing documentation, use the following standards to demonstrate the example code:

1. Python arguments, such as import statements, Boolean expressions, and function arguments should be wrapped as `code` by using `` on both sides of the code. Examples: ``import pygmt`` results in `import pygmt`, ``True`` results in `True`, ``style="v"`` results in `style="v"`.
2. Literal GMT arguments should be **bold** by wrapping the arguments with ** (two asterisks) on both sides. The argument description should be in *italicized* with * (single asterisk) on both sides. Examples: `**+l**\ *label*` results in **+l***label*, `**05m**` results in **05m**.
3. Optional arguments are wrapped with [ ] (square brackets).
4. Arguments that are mutually exclusive are separated with a | (bar) to denote "or".
5. Default arguments for parameters and configuration settings are wrapped with [ ] (square brackers) with the prefix "Default is". Example: [Default is **p**].

Cross-referencing with Sphinx

The API reference is manually assembled in `doc/api/index.rst`. The *autodoc* sphinx extension will automatically create pages for each function/class/module listed there.

You can reference functions, classes, methods, and modules from anywhere (including docstrings) using:

- :func:`package.module.function`
- :class:`package.module.class`
- :meth:`package.module.method`
- :mod:`package.module`

An example would be to use :meth:`pygmt.Figure.grdview` to link to https://www.pygmt.org/latest/api/generated/pygmt.Figure.grdview.html. PyGMT documentation that is not a class, method, or module can be linked with :doc:`Any Link Text </path/to/the/file>`. For example, :doc:`Install instructions </install>` links to https://www.pygmt.org/latest/install.html.

Linking to the GMT documentation and GMT configuration parameters can be done using:

- :gmt-docs:`page_name.html`
- :gmt-term:`GMT_PARAMETER`

An example would be using :gmt-docs:`makecpt.html` to link to https://docs.generic-mapping-tools.org/latest/makecpt.html. For GMT configuration parameters, an example is :gmt-term:`COLOR_FOREGROUND` to link to https://docs.generic-mapping-tools.org/latest/gmt.conf.html#term-COLOR_FOREGROUND.

Sphinx will create a link to the automatically generated page for that function/class/module.

9.24.6 Contributing Code

PyGMT Code Overview

The source code for PyGMT is located in the `pygmt/` directory. When contributing code, be sure to follow the general guidelines in the *pull request workflow* section.

Code Style

We use some tools to to format the code so we don't have to think about it:

- Black
- blackdoc
- docformatter
- isort

Black and blackdoc loosely follows the PEP8 guide but with a few differences. Regardless, you won't have to worry about formatting the code yourself. Before committing, run it to automatically format your code:

```
make format
```

For consistency, we also use UNIX-style line endings (`\n`) and file permission 644 (`-rw-r--r--`) throughout the whole project. Don't worry if you forget to do it. Our continuous integration systems will warn us and you can make a new commit with the formatted code. Even better, you can just write `/format` in the first line of any comment in a Pull Request to lint the code automatically.

When wrapping a new alias, use an underscore to separate words bridged by vowels (aeiou), such as `no_skip` and `z_only`. Do not use an underscore to separate words bridged only by consonants, such as `distcalc`, and `crossprofile`. This convention is not applied by the code checking tools, but the PyGMT maintainers will comment on any pull requests as needed.

We also use flake8 and pylint to check the quality of the code and quickly catch common errors. The `Makefile` contains rules for running both checks:

```
make check   # Runs black, blackdoc, docformatter, flake8 and isort (in check mode)
make lint    # Runs pylint, which is a bit slower
```

Testing your Code

Automated testing helps ensure that our code is as free of bugs as it can be. It also lets us know immediately if a change we make breaks any other part of the code.

All of our test code and data are stored in the `tests` subpackage. We use the pytest framework to run the test suite.

Please write tests for your code so that we can be sure that it won't break any of the existing functionality. Tests also help us be confident that we won't break your code in the future.

When writing tests, don't test everything that the GMT function already tests, such as the every unique combination arguments. An exception to this would be the most popular modules, such as `plot` and `basemap`. The highest priority for tests should be the Python-specific code, such as numpy, pandas, and xarray objects and the virtualfile mechanism.

If you're **new to testing**, see existing test files for examples of things to do. **Don't let the tests keep you from submitting your contribution!** If you're not sure how to do this or are having trouble, submit your pull request anyway. We will help you create the tests and sort out any kind of problem during code review.

Pull the baseline images, run the tests, and calculate test coverage using:

```
dvc status  # should report any files 'not_in_cache'
dvc pull  # pull down files from DVC remote cache (fetch + checkout)
make test
```

The coverage report will let you know which lines of code are touched by the tests. If all the tests pass, you can view the coverage reports by opening `htmlcov/index.html` in your browser. **Strive to get 100% coverage for the lines you changed.** It's OK if you can't or don't know how to test something. Leave a comment in the PR and we'll help you out.

You can also run tests in just one test script using:

```
pytest pygmt/tests/NAME_OF_TEST_FILE.py
```

or run tests which contain names that match a specific keyword expression:

```
pytest -k KEYWORD pygmt/tests
```

Testing Plots

Writing an image-based test is only slightly more difficult than a simple test. The main consideration is that you must specify the "baseline" or reference image, and compare it with a "generated" or test image. This is handled using the *decorator* functions `@pytest.mark.mpl_image_compare` and `@check_figures_equal` whose usage are further described below.

Using mpl_image_compare

This is the preferred way to test plots whenever possible.

This method uses the pytest-mpl plug-in to test plot generating code. Every time the tests are run, `pytest-mpl` compares the generated plots with known correct ones stored in `pygmt/tests/baseline`. If your test created a `pygmt.Figure` object, you can test it by adding a *decorator* and returning the `pygmt.Figure` object:

```python
@pytest.mark.mpl_image_compare
def test_my_plotting_case():
    "Test that my plotting function works"
    fig = Figure()
    fig.basemap(region=[0, 360, -90, 90], projection='W7i', frame=True)
    return fig
```

Your test function **must** return the `pygmt.Figure` object and you can only test one figure per function.

Before you can run your test, you'll need to generate a *baseline* (a correct version) of your plot. Run the following from the repository root:

```
pytest --mpl-generate-path=baseline pygmt/tests/NAME_OF_TEST_FILE.py
```

This will create a `baseline` folder with all the plots generated in your test file. Visually inspect the one corresponding to your test function. If it's correct, copy it (and only it) to `pygmt/tests/baseline`. When you run `make test` the next time, your test should be executed and passing.

Don't forget to commit the baseline image as well! The images should be pushed up into a remote repository using `dvc` (instead of `git`) as will be explained in the next section.

Using Data Version Control (dvc) to Manage Test Images

As the baseline images are quite large blob files that can change often (e.g. with new GMT versions), it is not ideal to store them in `git` (which is meant for tracking plain text files). Instead, we will use dvc which is like `git` but for data. What `dvc` does is to store the hash (md5sum) of a file. For example, given an image file like `test_logo.png`, `dvc` will generate a `test_logo.png.dvc` plain text file containing the hash of the image. This `test_logo.png.dvc` file can be stored as usual on GitHub, while the `test_logo.png` file can be stored separately on our `dvc` remote at https://dagshub.com/GenericMappingTools/pygmt.

To **pull** or sync files from the `dvc` remote to your local repository, use the commands below. Note how `dvc` commands are very similar to `git`.

```
dvc status  # should report any files 'not_in_cache'
dvc pull  # pull down files from DVC remote cache (fetch + checkout)
```

Once the sync/download is complete, you should notice two things. There will be images stored in the `pygmt/tests/baseline` folder (e.g. `test_logo.png`) and these images are technically reflinks/symlinks/copies of the files under the `.dvc/cache` folder. You can now run the image comparison test suite as per usual.

```
pytest pygmt/tests/test_logo.py  # run only one test
make test  # run the entire test suite
```

To **push** or sync changes from your local repository up to the `dvc` remote at DAGsHub, you will first need to set up authentication using the commands below. This only needs to be done once, i.e. the first time you contribute a test image to the PyGMT project.

```
dvc remote modify upstream --local auth basic
dvc remote modify upstream --local user "$DAGSHUB_USER"
dvc remote modify upstream --local password "$DAGSHUB_PASS"
```

The configuration will be stored inside your `.dvc/config.local` file. Note that the $DAGSHUB_PASS token can be generated at https://dagshub.com/user/settings/tokens after creating a DAGsHub account (can be linked to your GitHub account). Once you have an account set up, please ask one of the PyGMT maintainers to add you as a collaborator at https://dagshub.com/GenericMappingTools/pygmt/settings/collaboration before proceeding with the next steps.

The entire workflow for generating or modifying baseline test images can be summarized as follows:

```
# Sync with both git and dvc remotes
git pull
dvc pull

# Generate new baseline images
pytest --mpl-generate-path=baseline pygmt/tests/test_logo.py
mv baseline/*.png pygmt/tests/baseline/

# Generate hash for baseline image and stage the *.dvc file in git
git rm -r --cached 'pygmt/tests/baseline/test_logo.png'  # only run if migrating␣
↪existing image from git to dvc
dvc status  # check which files need to be added to dvc
dvc add pygmt/tests/baseline/test_logo.png
git add pygmt/tests/baseline/test_logo.png.dvc

# Commit changes and push to both the git and dvc remotes
git commit -m "Add test_logo.png into DVC"
```

(continues on next page)

(continued from previous page)

```
git push
dvc push
```

Using check_figures_equal

This approach draws the same figure using two different methods (the reference method and the tested method), and checks that both of them are the same. It takes two `pygmt.Figure` objects ('fig_ref' and 'fig_test'), generates a png image, and checks for the Root Mean Square (RMS) error between the two. Here's an example:

```
@check_figures_equal()
def test_my_plotting_case():
  "Test that my plotting function works"
  fig_ref, fig_test = Figure(), Figure()
  fig_ref.grdimage("@earth_relief_01d_g", projection="W120/15c", cmap="geo")
  fig_test.grdimage(grid, projection="W120/15c", cmap="geo")
  return fig_ref, fig_test
```

9.25 Maintainers Guide

This page contains instructions for project maintainers about how our setup works, making releases, creating packages, etc.

If you want to make a contribution to the project, see the Contributing Guide instead.

9.25.1 Onboarding Access Checklist

- Added to python-maintainers team in the GenericMappingTools organization on GitHub (gives 'maintain' permissions)
- Added as collaborator on DAGsHub (gives 'write' permission to dvc remote storage)
- Added as moderator on GMT forum (to see mod-only discussions)
- Added as member on the PyGMT devs Slack channel (for casual conversations)
- Added as maintainer on PyPI and Test PyPI [optional]
- Added as member on HackMD [optional]

9.25.2 Branches

- *main*: Always tested and ready to become a new version. Don't push directly to this branch. Make a new branch and submit a pull request instead.
- *gh-pages*: Holds the HTML documentation and is served by GitHub. Pages for the main branch are in the `dev` folder. Pages for each release are in their own folders. **Automatically updated by GitHub Actions** so you shouldn't have to make commits here.

9.25.3 Managing GitHub issues

A few guidelines for managing GitHub issues:

- Assign labels and the expected milestone to issues as appropriate.
- When people request to work on an open issue, either assign the issue to that person and post a comment about the assignment or explain why you are not assigning the issue to them and, if possible, recommend other issues for them to work on.
- People with write access should self-assign issues and/or comment on the issues that they will address.
- For upstream bugs, close the issue after an upstream release fixes the bug. If possible, post a comment when an upstream PR is merged that fixes the problem, and consider adding a regression test for serious bugs.

9.25.4 Reviewing and merging pull requests

A few guidelines for reviewing:

- Always **be polite** and give constructive feedback.
- Welcome new users and thank them for their time, even if we don't plan on merging the PR.
- Don't be harsh with code style or performance. If the code is bad, either (1) merge the pull request and open a new one fixing the code and pinging the original submitter (2) comment on the PR detailing how the code could be improved. Both ways are focused on showing the contributor **how to write good code**, not shaming them.

Pull requests should be **squash merged**. This means that all commits will be collapsed into one. The main advantages of this are:

- Eliminates experimental commits or commits to undo previous changes.
- Makes sure every commit on the main branch passes the tests and has a defined purpose.
- The maintainer writes the final commit message, so we can make sure it's good and descriptive.

9.25.5 Continuous Integration

We use GitHub Actions continuous integration (CI) services to build and test the project on Linux, macOS and Windows. They rely on the `environment.yml` file to install required dependencies using conda and the `Makefile` to run the tests and checks.

There are 11 configuration files located in `.github/workflows`:

1. `style_checks.yaml` (Code lint and style checks)

   This is run on every commit to the *main* and Pull Request branches. It is also scheduled to run daily on the *main* branch.

2. `ci_tests.yaml` (Tests on Linux/macOS/Windows)

   This is run on every commit to the *main* and Pull Request branches. It is also scheduled to run daily on the *main* branch. In draft Pull Requests, only two jobs on Linux are triggered to save on Continuous Integration resources:

   - Minimum NEP29 Python/NumPy versions
   - Latest Python/NumPy versions + optional packages (e.g. GeoPandas)

   This workflow is also responsible for uploading test coverage reports stored in `.coverage.xml` to https://app.codecov.io/gh/GenericMappingTools/pygmt via the Codecov GitHub Action. More codecov related configurations are stored in `.github/codecov.yml`.

3. `ci_docs.yml` (Build documentation on Linux/macOS/Windows)

   This is run on every commit to the *main* and Pull Request branches. In draft Pull Requests, only the job on Linux is triggered to save on Continuous Integration resources.

   On the *main* branch, the workflow also handles the documentation deployment:

   - Updating the development documentation by pushing the built HTML pages from the *main* branch onto the `dev` folder of the *gh-pages* branch.
   - Updating the `latest` documentation link to the new release.

4. `ci_tests_dev.yaml` (GMT Dev Tests on Linux/macOS/Windows).

   This is triggered when a PR is marked as "ready for review", or using the slash command `/test-gmt-dev`. It is also scheduled to run daily on the *main* branch.

5. `cache_data.yaml` (Caches GMT remote data files needed for GitHub Actions CI)

   This is scheduled to run every Sunday at 12:00 (UTC). If new remote files are needed urgently, maintainers can manually uncomment the 'pull_request:' line in that `cache_data.yaml` file to refresh the cache.

6. `publish-to-pypi.yml` (Publish wheels to PyPI and TestPyPI)

   This workflow is run to publish wheels to PyPI and TestPyPI (for testing only). Archives will be pushed to TestPyPI on every commit to the *main* branch and tagged releases, and to PyPI for tagged releases only.

7. `release-drafter.yml` (Drafts the next release notes)

   This workflow is run to update the next releases notes as pull requests are merged into the main branch.

8. `check-links.yml` (Check links in the repository and website)

   This workflow is run weekly to check all external links in plaintext and HTML files. It will create an issue if broken links are found.

9. `format-command.yml` (Format the codes using slash command)

   This workflow is triggered in a PR if the slash command `/format` is used.

10. `dvc-diff.yml` (Report changes to test images on dvc remote)

    This workflow is triggered in a PR when any *.png.dvc files have been added, modified, or deleted. A GitHub comment will be published that contains a summary table of the images that have changed along with a visual report.

11. `release-baseline-images.yml` (Upload the ZIP archive of baseline images as a release asset)

    This workflow is run to upload the ZIP archive of baseline images as a release asset when a release is published.

9.25.6 Continuous Documentation

We use the Vercel for GitHub App to preview changes made to our documentation website every time we make a commit in a pull request. The service has a configuration file `vercel.json`, with a list of options to change the default behaviour at https://vercel.com/docs/configuration. The actual script `package.json` is used by Vercel to install the necessary packages, build the documentation, copy the files to a 'public' folder and deploy that to the web, see https://vercel.com/docs/build-step.

9.25.7 Dependencies Policy

PyGMT has adopted NEP29 alongside the rest of the Scientific Python ecosystem, and therefore supports:

- All minor versions of Python released 42 months prior to the project, and at minimum the two latest minor versions.
- All minor versions of NumPy released in the 24 months prior to the project, and at minimum the last three minor versions.

In `setup.py`, the `python_requires` variable should be set to the minimum supported version of Python. Minimum Python and NumPy version support should be adjusted upward on every major and minor release, but never on a patch release.

9.25.8 Backwards compatibility and deprecation policy

PyGMT is still undergoing rapid development. All of the API is subject to change until the v1.0.0 release. Versioning in PyGMT is based on the semantic versioning specification (e.g. vMAJOR.MINOR.PATCH). Basic policy for backwards compatibility:

- Any incompatible changes should go through the deprecation process below.
- Incompatible changes are only allowed in major and minor releases, not in patch releases.
- Incompatible changes should be documented in the release notes.

When making incompatible changes, we should follow the process:

- Discuss whether the incompatible changes are necessary on GitHub.
- Make the changes in a backwards compatible way, and raise a `FutureWarning` warning for old usage. At least one test using the old usage should be added.
- The warning message should clearly explain the changes and include the versions in which the old usage is deprecated and is expected to be removed.
- The `FutureWarning` warning should appear for 2-4 minor versions, depending on the impact of the changes. It means the deprecation period usually lasts 3-12 months.
- Remove the old usage and warning when reaching the declared version.

To rename a function parameter, add the `@deprecate_parameter` decorator near the top after the `@fmt_docstring` decorator but before the `@use_alias` decorator (if those two exists). Here is an example:

```python
@fmt_docstring
@deprecate_parameter("columns", "incols", "v0.4.0", remove_version="v0.6.0")
@use_alias(J="projection", R="region", V="verbose", i="incols")
@kwargs_to_strings(R="sequence", i='sequence_comma')
def plot(self, x=None, y=None, data=None, size=None, direction=None, **kwargs):
    pass
```

In this case, the old parameter name `columns` is deprecated since v0.4.0, and will be fully removed in v0.6.0. The new parameter name is `incols`.

9.25.9 Making a Release

We try to automate the release process as much as possible. GitHub Actions workflow handles publishing new releases to PyPI and updating the documentation. The version number is set automatically using setuptools_scm based information obtained from git. There are a few steps that still must be done manually, though.

Updating the changelog

The Release Drafter GitHub Action will automatically keep a draft changelog at https://github.com/GenericMappingTools/pygmt/releases, adding a new entry every time a Pull Request (with a proper label) is merged into the main branch. This release drafter tool has two configuration files, one for the GitHub Action at .github/workflows/release-drafter.yml, and one for the changelog template at .github/release-drafter.yml. Configuration settings can be found at https://github.com/release-drafter/release-drafter.

The drafted release notes are not perfect, so we will need to tidy it prior to publishing the actual release notes at https://www.pygmt.org/latest/changes.html.

1. Go to https://github.com/GenericMappingTools/pygmt/releases and click on the 'Edit' button next to the current draft release note. Copy the text of the automatically drafted release notes under the 'Write' tab to `doc/changes.md`. Add a section separator `---` between the new and old changelog sections.
2. Update the DOI badge in the changelog. Remember to replace the DOI number inside the badge url.

   ```
   [![Digital Object Identifier for PyGMT vX.Y.Z](https://zenodo.org/badge/DOI/10.5281/
   ↪zenodo.<INSERT-DOI-HERE>.svg)](https://doi.org/10.5281/zenodo.<INSERT-DOI-HERE>)
   ```

3. Open a new Pull Request using the title 'Changelog entry for vX.Y.Z' with the updated release notes, so that other people can help to review and collaborate on the changelog curation process described next.
4. Edit the change list to remove any trivial changes (updates to the README, typo fixes, CI configuration, test updates due to GMT releases, etc).
5. Sort the items within each section (i.e., New Features, Enhancements, etc.) such that similar items are located near each other (e.g., new wrapped modules, gallery examples, API docs changes) and entries within each group are alphabetical.
6. Move a few important items from the main sections to the highlights section.
7. Edit the list of people who contributed to the release, linking to their GitHub account. Sort their names by the number of commits made since the last release (e.g., use `git shortlog HEAD...v0.4.0 -sne`).
8. Update `README.rst` with new information on the new release version, including a vX.Y.Z documentation link, and compatibility with GMT/Python/NumPy versions. Follow NEP 29 for compatibility updates.
9. Refresh citation information. Specifically, the BibTeX in `README.rst` and `CITATION.cff` needs to be updated with any metadata changes. Please follow guidelines in `AUTHORSHIP.md` for updating the author list in the BibTeX. More information about the `CITATION.cff` specification can be found at https://github.com/citation-file-format/citation-file-format/blob/main/schema-guide.md

Check the README syntax

GitHub is a bit forgiving when it comes to the RST syntax in the README but PyPI is not. So slightly broken RST can cause the PyPI page to not render the correct content. Check using the `rst2html.py` script that comes with docutils:

```
python setup.py --long-description | rst2html.py --no-raw > index.html
```

Open `index.html` and check for any flaws or error messages.

Pushing to PyPI and updating the documentation

After the changelog is updated, making a release can be done by going to https://github.com/GenericMappingTools/pygmt/releases, editing the draft release, and clicking on publish. A git tag will also be created, make sure that this tag is a proper version number (following Semantic Versioning) with a leading `v`. E.g. `v0.2.1`.

Once the release/tag is created, this should trigger GitHub Actions to do all the work for us. A new source distribution will be uploaded to PyPI, a new folder with the documentation HTML will be pushed to *gh-pages*, and the `latest` link will be updated to point to this new folder.

Archiving on Zenodo

Grab both the source code and baseline images zip files from the GitHub release page and upload them to Zenodo using the previously reserved DOI.

Updating the conda package

When a new version is released on PyPI, conda-forge's bot automatically creates version updates for the feedstock. In most cases, the maintainers can simply merge that PR.

If changes need to be done manually, you can:

1. Fork the pygmt feedstock repository if you haven't already. If you have a fork, update it.
2. Update the version number and sha256 hash on `recipe/meta.yaml`. You can get the hash from the PyPI "Download files" section.
3. Add or remove any new dependencies (most are probably only `run` dependencies).
4. Make a new branch, commit, and push the changes **to your personal fork**.
5. Create a PR against the original feedstock main.
6. Once the CI tests pass, merge the PR or ask a maintainer to do so.

PYTHON MODULE INDEX

INDEX