The PyGMT Documentation

Release v0.16.0.dev39+g7741d064

The PyGMT Developers

Apr 29, 2025

GETTING STARTED

1 Overview **1**
- 1.1 About . . . 1
- 1.2 Presentations . . . 1

2 Installing **5**
- 2.1 Quickstart . . . 5
- 2.2 Which Python? . . . 6
- 2.3 Which GMT? . . . 6
- 2.4 Dependencies . . . 6
- 2.5 Installing GMT and other dependencies . . . 7
- 2.6 Installing PyGMT . . . 8
- 2.7 Testing your install . . . 9
- 2.8 Common installation issues . . . 10

3 Intro to PyGMT **13**
- 3.1 1. Making your first figure . . . 13
- 3.2 2. Create a contour map . . . 19
- 3.3 3. Figure elements . . . 24
- 3.4 4. PyGMT I/O: Table inputs . . . 26

4 Tutorials **31**
- 4.1 Basics . . . 31
- 4.2 Advanced . . . 73

5 Gallery **159**
- 5.1 Maps and map elements . . . 159
- 5.2 Lines and vectors . . . 168
- 5.3 Symbols and markers . . . 196
- 5.4 Images, contours, and fields . . . 214
- 5.5 3D Plots . . . 230
- 5.6 Seismology and geodesy . . . 234
- 5.7 Base maps . . . 237
- 5.8 Histograms . . . 241
- 5.9 Plot embellishments . . . 247

6 Projections **263**
- 6.1 Azimuthal Projections . . . 263
- 6.2 Conic Projections . . . 270
- 6.3 Cylindric Projections . . . 273
- 6.4 Miscellaneous Projections . . . 284
- 6.5 Non-geographic Projections . . . 290

7 External Resources **299**
7.1 Tutorials . . . 299
7.2 Examples from Publications and Posters . . . 308

8 API Reference **311**
8.1 Plotting . . . 311
8.2 Data Processing . . . 385
8.3 Input/output . . . 470
8.4 GMT Defaults . . . 471
8.5 Metadata . . . 473
8.6 Xarray Integration . . . 476
8.7 Enums . . . 480
8.8 Miscellaneous . . . 481
8.9 Datasets . . . 482
8.10 Exceptions . . . 514
8.11 GMT C API . . . 515

9 Technical Reference **539**
9.1 Common Parameters . . . 539
9.2 GMT Map Projections . . . 539
9.3 Supported Fonts . . . 540
9.4 Bit and Hachure Patterns . . . 542
9.5 Supported Encodings and Non-ASCII Characters . . . 544
9.6 Environment Variables . . . 547

10 Changelog **549**
10.1 Release v0.15.0 (2025/03/31) . . . 549
10.2 Release v0.14.2 (2025/02/15) . . . 550
10.3 Release v0.14.1 (2025/02/01) . . . 551
10.4 Release v0.14.0 (2024/12/31) . . . 551
10.5 Release v0.13.0 (2024/09/05) . . . 554
10.6 Release v0.12.0 (2024/05/01) . . . 556
10.7 Release v0.11.0 (2024/02/01) . . . 559
10.8 Release v0.10.0 (2023/09/02) . . . 561
10.9 Release v0.9.0 (2023/03/31) . . . 564
10.10 Release v0.8.0 (2022/12/30) . . . 566
10.11 Release v0.7.0 (2022/07/01) . . . 568
10.12 Release v0.6.1 (2022/04/11) . . . 570
10.13 Release v0.6.0 (2022/03/14) . . . 571
10.14 Release v0.5.0 (2021/10/29) . . . 573
10.15 Release v0.4.1 (2021/08/07) . . . 576
10.16 Release v0.4.0 (2021/06/20) . . . 577
10.17 Release v0.3.1 (2021/03/14) . . . 581
10.18 Release v0.3.0 (2021/02/15) . . . 582
10.19 Release v0.2.1 (2020/11/14) . . . 585
10.20 Release v0.2.0 (2020/09/12) . . . 587
10.21 Release v0.1.2 (2020/07/07) . . . 588
10.22 Release v0.1.1 (2020/05/22) . . . 589
10.23 Release v0.1.0 (2020/05/03) . . . 590

11 Minimum Supported Versions **593**

12 Ecosystem **595**
12.1 PyGMT dependencies . . . 595
12.2 PyGMT ecosystem . . . 597

13 PyGMT Team **599**
13.1 Founders . . . 600
13.2 Active Maintainers . . . 602
13.3 Distinguished Contributors . . . 606

14 Contributors Guide **611**
14.1 Ways to Contribute . . . 611
14.2 Providing Feedback . . . 611
14.3 General Guidelines . . . 612
14.4 Setting up your Environment . . . 614
14.5 Contributing Documentation . . . 614
14.6 Contributing Code . . . 618

15 Maintainers Guide **623**
15.1 Onboarding/Offboarding Access Checklist . . . 623
15.2 Branches . . . 624
15.3 Managing GitHub Issues . . . 624
15.4 Reviewing and Merging Pull Requests . . . 624
15.5 Continuous Integration . . . 624
15.6 Continuous Documentation . . . 625
15.7 Continuous Benchmarking . . . 625
15.8 Dependencies Policy . . . 625
15.9 Backwards Compatibility and Deprecation Policy . . . 625
15.10 Making a Release . . . 627

CHAPTER

ONE

OVERVIEW

1.1 About

PyGMT is a Python wrapper for the Generic Mapping Tools (GMT), a command-line program widely used across the Earth, Ocean, and Planetary sciences and beyond. 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 Hawai'i at Mānoa. The development of PyGMT has been supported by NSF grants OCE-1558403 and EAR-1948602.

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

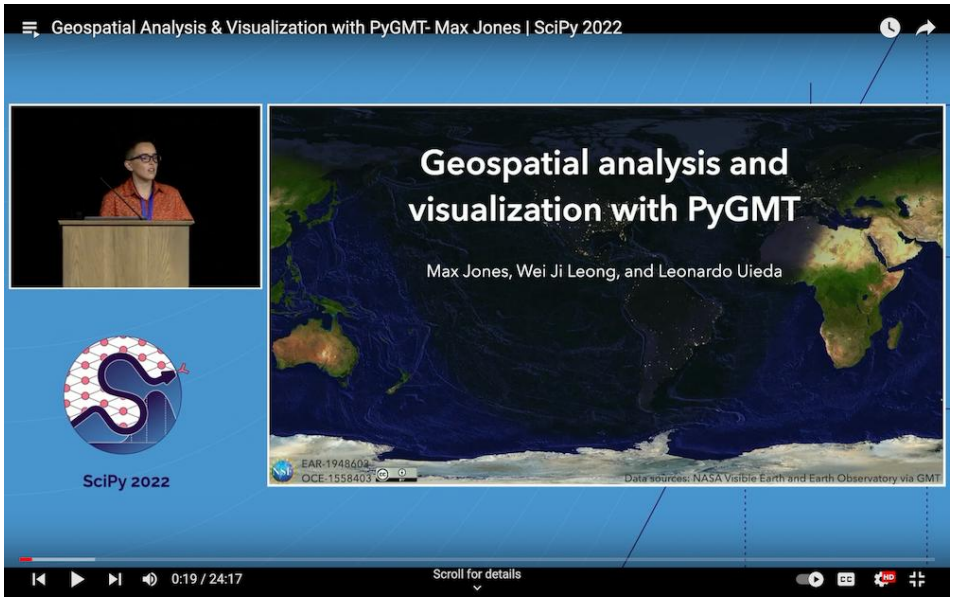
1.2 Presentations

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

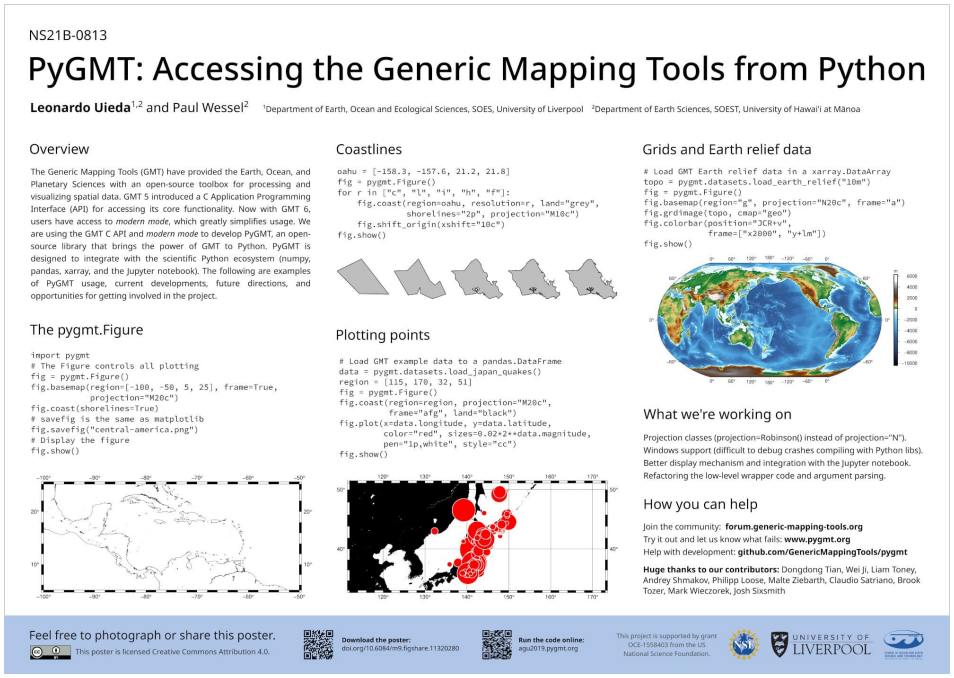
- "Accessing and Integrating GMT with Python and the Scientific Python Ecosystem". 2024. Yvonne Fröhlich, Dongdong Tian, Wei Ji Leong, Max Jones, and Michael Grund. Presented at *AGU 2024*. doi:10.6084/m9.figshare.28049495



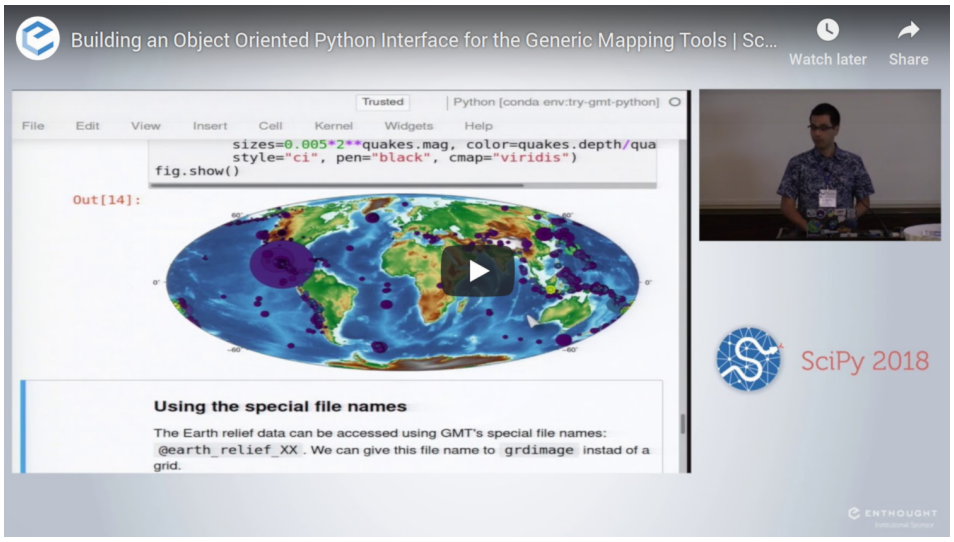
- "Geospatial Analysis & Visualization with PyGMT". 2022. Max Jones, Wei Ji Leong, and Leonardo Uieda. Presented at *SciPy 2022*. doi:10.6084/m9.figshare.20483793



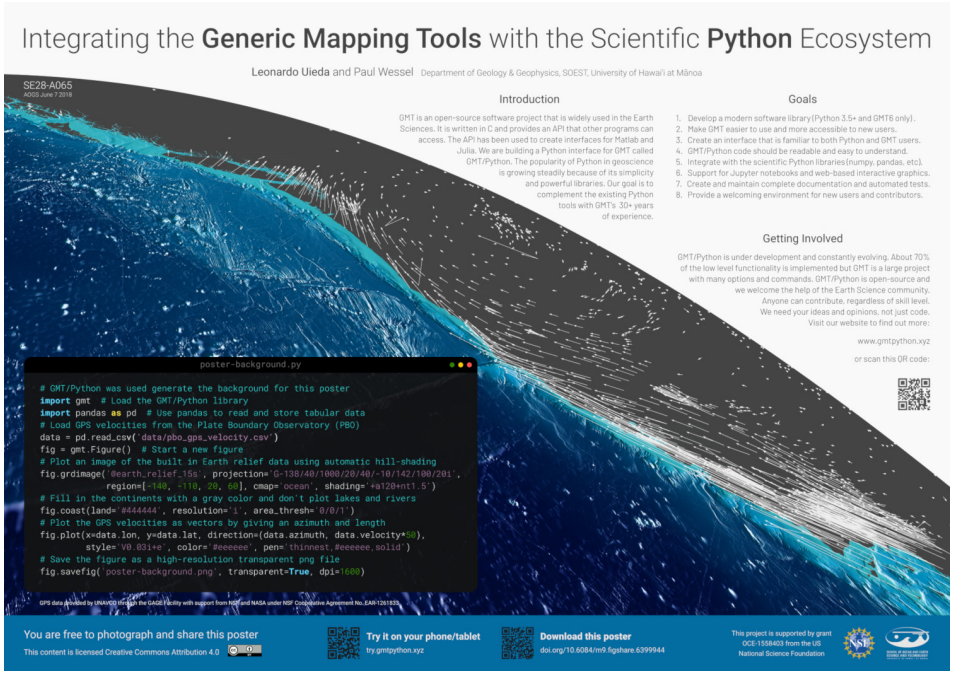
- "PyGMT: Accessing the Generic Mapping Tools from Python". 2019. Leonardo Uieda and Paul Wessel. Presented at *AGU 2019*. doi:10.6084/m9.figshare.11320280



- "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



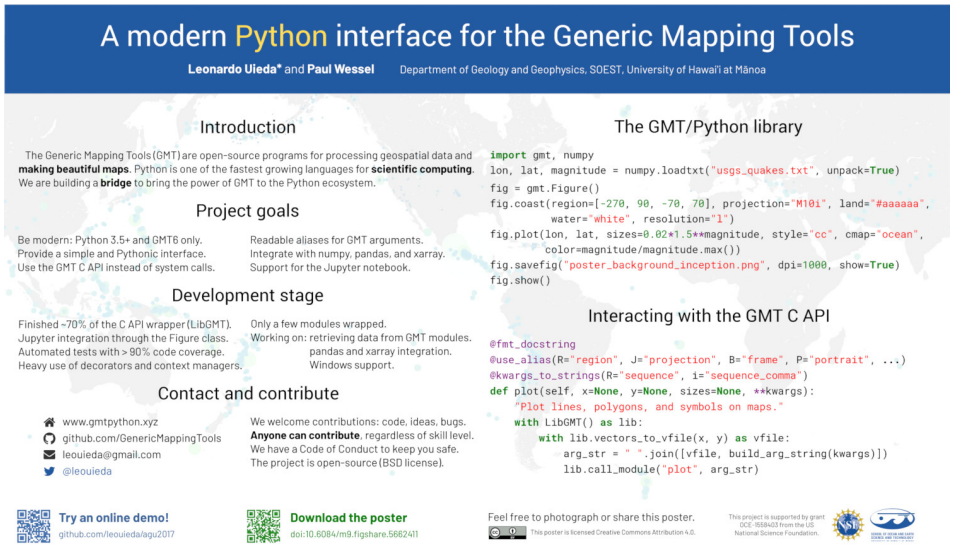
- "Integrating the Generic Mapping Tools with the Scientific Python Ecosystem". 2018. Leonardo Uieda and Paul Wessel. Presented at *AOGS Annual Meeting 2018*. doi:10.6084/m9.figshare.6399944



- "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



CHAPTER

TWO

INSTALLING

2.1 Quickstart

The fastest way to install PyGMT is with the mamba or 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:

mamba

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

conda

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

To activate the virtual environment, you can do:

mamba

```
mamba activate pygmt
```

conda

```
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()
```

```
PyGMT information:
  version: v0.16.0.dev39+g7741d064
System information:
  python: 3.13.3 | packaged by conda-forge | (main, Apr 14 2025, 20:44:03) [GCC 13.3.
↪0]
  executable: /home/runner/micromamba/envs/pygmt/bin/python
  machine: Linux-6.11.0-1012-azure-x86_64-with-glibc2.39
Dependency information:
  numpy: 2.2.5
  pandas: 2.2.3
```

(continues on next page)

(continued from previous page)

```
  xarray: 2025.3.1
  packaging: 25.0
  contextily: 1.6.2
  geopandas: 1.0.1
  IPython: 9.2.0
  pyarrow: 19.0.1
  rioxarray: 0.19.0
  gdal: None
  ghostscript: 10.04.0
GMT library information:
  version: 6.5.0
  padding: 2
  share dir: /home/runner/micromamba/envs/pygmt/share/gmt
  plugin dir: /home/runner/micromamba/envs/pygmt/lib/gmt/plugins
  library path: /home/runner/micromamba/envs/pygmt/lib/libgmt.so
  cores: 4
  grid layout: rows
  image layout:
  binary version: 6.5.0
```

You are now ready to make your first figure! Start by looking at our *Intro*, *Tutorials*, and *Gallery*. 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.

2.2 Which Python?

PyGMT is tested to run on Python >=3.11.

We recommend using the Miniforge Python distribution to ensure you have all dependencies installed and the mamba package manager in the base environment. Installing Miniforge does not require administrative rights to your computer and doesn't interfere with any other Python installations on your system.

2.3 Which GMT?

PyGMT requires Generic Mapping Tools (GMT) >=6.4.0 since there are many changes being made to GMT itself in response to the development of PyGMT.

Compiled conda packages of GMT for Linux, macOS and Windows are provided through conda-forge. Advanced users can also build GMT from source instead.

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

2.4 Dependencies

PyGMT requires the following packages to be installed:

- NumPy
- pandas

- Xarray
- packaging

> **Note**
>
> For the minimum supported versions of the dependencies, please see *Minimum Supported Versions*.

> **Note**
>
> Some optional dependencies (e.g., IPython, GeoPandas) add more functionality to PyGMT. For a complete list of the optional dependencies, refer to *Ecosystem*.

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 `mamba` or `conda` package manager. We recommend working in an isolated virtual 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 virtual environment with Python and all our dependencies installed (we'll call it `pygmt` but feel free to change it to whatever you want):

mamba

```
mamba create --name pygmt python=3.13 numpy pandas xarray packaging gmt
```

conda

```
conda create --name pygmt python=3.13 numpy pandas xarray packaging gmt
```

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

mamba

```
mamba activate pygmt
```

conda

```
conda activate pygmt
```

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

> **Tip**
>
> You can also enable more PyGMT functionalities by installing PyGMT's optional dependencies in the environment.

mamba

```
mamba install contextily geopandas ipython pyarrow-core rioxarray
```

conda

```
conda install contextily geopandas ipython pyarrow-core rioxarray
```

2.6 Installing PyGMT

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

2.6.1 Using mamba/conda (recommended)

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

mamba

```
mamba install pygmt
```

conda

```
conda install pygmt
```

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

mamba

```
mamba update pygmt
```

conda

```
conda update pygmt
```

2.6.2 Using pip

This installs the latest stable release from PyPI:

```
python -m pip install pygmt
```

Tip

You can also run `python -m pip install pygmt[all]` to install PyGMT with all of its optional dependencies.

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

```
python -m pip install --pre --extra-index-url https://test.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 (mamba/conda/pip) should allow you to use the PyGMT package from Python.

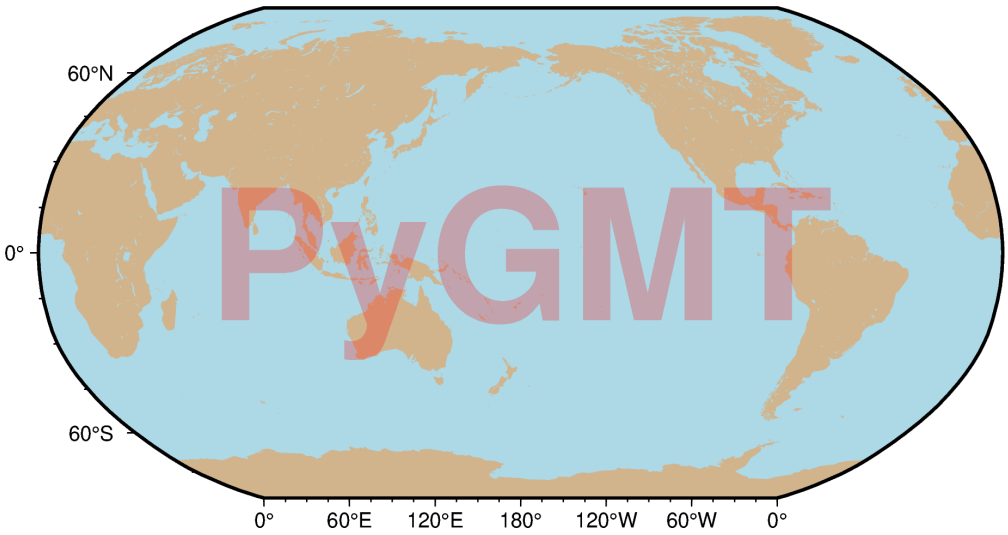
2.7 Testing your install

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

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

```
PyGMT information:
  version: v0.16.0.dev39+g7741d064
System information:
  python: 3.13.3 | packaged by conda-forge | (main, Apr 14 2025, 20:44:03) [GCC 13.3.
↪0]
  executable: /home/runner/micromamba/envs/pygmt/bin/python
  machine: Linux-6.11.0-1012-azure-x86_64-with-glibc2.39
Dependency information:
  numpy: 2.2.5
  pandas: 2.2.3
  xarray: 2025.3.1
  packaging: 25.0
  contextily: 1.6.2
  geopandas: 1.0.1
  IPython: 9.2.0
  pyarrow: 19.0.1
  rioxarray: 0.19.0
  gdal: None
  ghostscript: 10.04.0
GMT library information:
  version: 6.5.0
  padding: 2
  share dir: /home/runner/micromamba/envs/pygmt/share/gmt
  plugin dir: /home/runner/micromamba/envs/pygmt/lib/gmt/plugins
  library path: /home/runner/micromamba/envs/pygmt/lib/libgmt.so
  cores: 4
  grid layout: rows
  image layout:
  binary version: 6.5.0
```

```
fig = pygmt.Figure()
fig.coast(projection="N15c", region="g", frame=True, land="tan", water="lightblue")
fig.text(position="MC", text="PyGMT", font="80p,Helvetica-Bold,red@75")
fig.show()
```



You should see a global map with land and water masses colored in tan and lightblue respectively. On top, there should be the semi-transparent text "PyGMT". If the semi-transparency does not show up, there is probably an incompatibility between your GMT and Ghostscript versions. For details, please run `pygmt.show_versions()` and see *Not working transparency*.

2.8 Common installation issues

If you have any issues with the installation, please check out the following common problems and solutions.

2.8.1 "Error loading GMT shared library at ..."

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 environment variable *GMT_LIBRARY_PATH* to the directory where `libgmt.so`, `libgmt.dylib` or `gmt.dll` can be found on Linux, macOS or Windows, respectively.

For Linux/macOS, add the following line to your shell configuration file (usually `~/.bashrc` for Bash on Linux and `~/.zshrc` for Zsh on macOS):

```
export GMT_LIBRARY_PATH=$HOME/miniforge3/envs/pygmt/lib
```

For Windows, add the environment variable *GMT_LIBRARY_PATH* following these instructions and set its value to a path like:

```
C:\Users\USERNAME\Miniforge3\envs\pygmt\Library\bin\
```

2.8.2 `ModuleNotFoundError` in Jupyter notebook environment

If you can successfully import PyGMT in a Python interpreter or IPython, but get a `ModuleNotFoundError` when importing PyGMT in Jupyter, you may need to activate your `pygmt` virtual environment (using `mamba activate pygmt` or `conda activate pygmt`) and install a `pygmt` kernel following the commands below:

```
python -m ipykernel install --user --name pygmt  # install virtual environment␣
↪properly
jupyter kernelspec list --json
```

After that, you need to restart Jupyter, open your notebook, select the `pygmt` kernel and then import pygmt.

2.8.3 Not working transparency

It is known that some combinations of GMT and Ghostscript versions cause issues, especially regarding transparency. If the transparency doesn't work in your figures, please check your GMT and Ghostscript versions (you can run `pygmt.show_versions()`). We recommend:

- Ghostscript 9.53-9.56 for GMT 6.4.0 (or below)
- Ghostscript 10.03 or later for GMT 6.5.0

CHAPTER

THREE

INTRO TO PYGMT

Welcome to PyGMT! The tutorials in this intro are designed to teach basic concepts to create maps in PyGMT.

About this intro

It is assumed that PyGMT has been successfully *installed* on your system. To test this, run `import pygmt` in a Python IDE or Jupyter notebook.

This intro will progressively cover PyGMT data manipulation and plotting concepts, and later tutorials will use concepts explained in previous ones. It will not cover all PyGMT functions and methods.

3.1 1. Making your first figure

This tutorial covers the basics of creating a figure using PyGMT - a Python wrapper for the Generic Mapping Tools (GMT). It will only use the *pygmt.Figure.coast* method for plotting. Later tutorials will address other PyGMT methods.

3.1.1 Loading the library

The first step is to import `pygmt`. All methods and figure generation are accessible from the `pygmt` top level package.

```
import pygmt
```

3.1.2 Creating a figure

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()
```

To add elements to the figure instance or object (`fig` in this example) different methods can be called on it. This example will use the *pygmt.Figure.coast* method, which can be used to create a map without any other methods or external data. The *pygmt.Figure.coast* method plots the coastlines, borders, and bodies of water using a database that is included in GMT.

First, a region for the figure must be selected. This example will plot some of the coast of Maine in the northeastern US. A Python list can be passed to the `region` parameter with the minimum and maximum X-values (longitude) and the minimum and maximum Y-values (latitude). For this example, the minimum (bottom left) coordinates are (N43.75, W69) and the maximum (top right) coordinates are (N44.75, W68). Negative values can be passed for latitudes in the southern hemisphere or longitudes in the western hemisphere.

In addition to the region, an argument needs to be passed to *pygmt.Figure.coast* to tell it what to plot. In this example, *pygmt.Figure.coast* will be told to plot the shorelines by passing the Boolean value `True` to the `shorelines` parameter. The `shorelines` parameter has other options for finer control, but setting it to `True` uses the default values.

```
fig.coast(region=[-69, -68, 43.75, 44.75], shorelines=True)
```

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

```
fig.show()
```



3.1.3 Color the land and water

This figure plots all of the coastlines in the given region, but it does not indicate where the land and water are. Color values can be passed to `land` and `water` to set the colors on the figure.

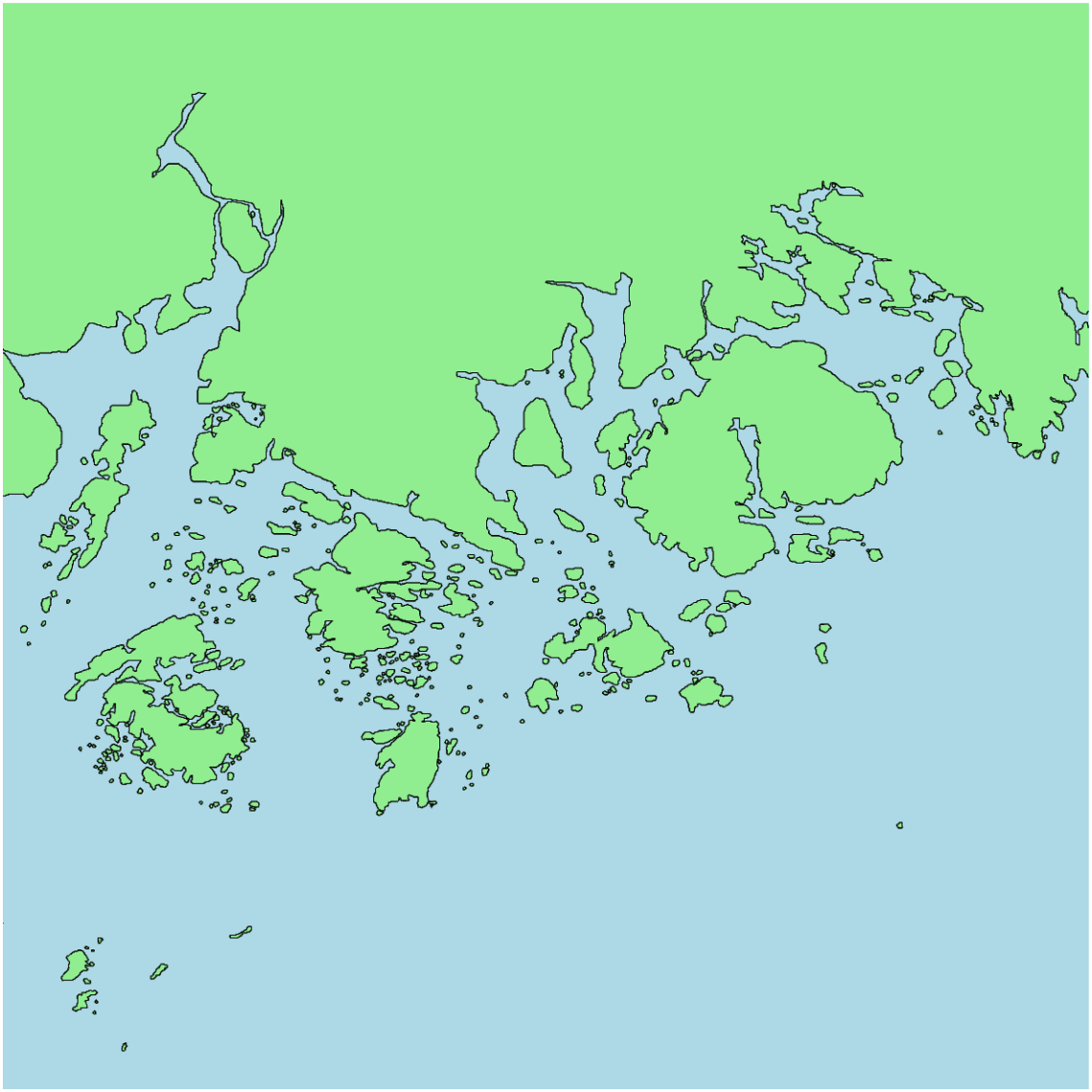
When plotting colors in PyGMT, there are multiple color codes, that can be used. This includes standard GMT color names (like `"skyblue"`), R/G/B levels (like `"0/0/255"`), a hex value (like `"#333333"`), or a gray level (like `"gray50"`). For this example, GMT color names are used.

```
fig = pygmt.Figure()
fig.coast(
    region=[-69, -68, 43.75, 44.75],
    shorelines=True,
    land="lightgreen",
    water="lightblue",
```

(continues on next page)

(continued from previous page)

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



3.1.4 Set the projection

This figure now has its colors set. By default the projection and size of the map is set to `"Q15c+du"`. Here, `"Q"` defines a cylindrical equidistant map projection, `"15c+du"` means setting the maximum (upper) map dimension to 15 cm. However, both of these values can be customized according to the requirements using the `projection` parameter.

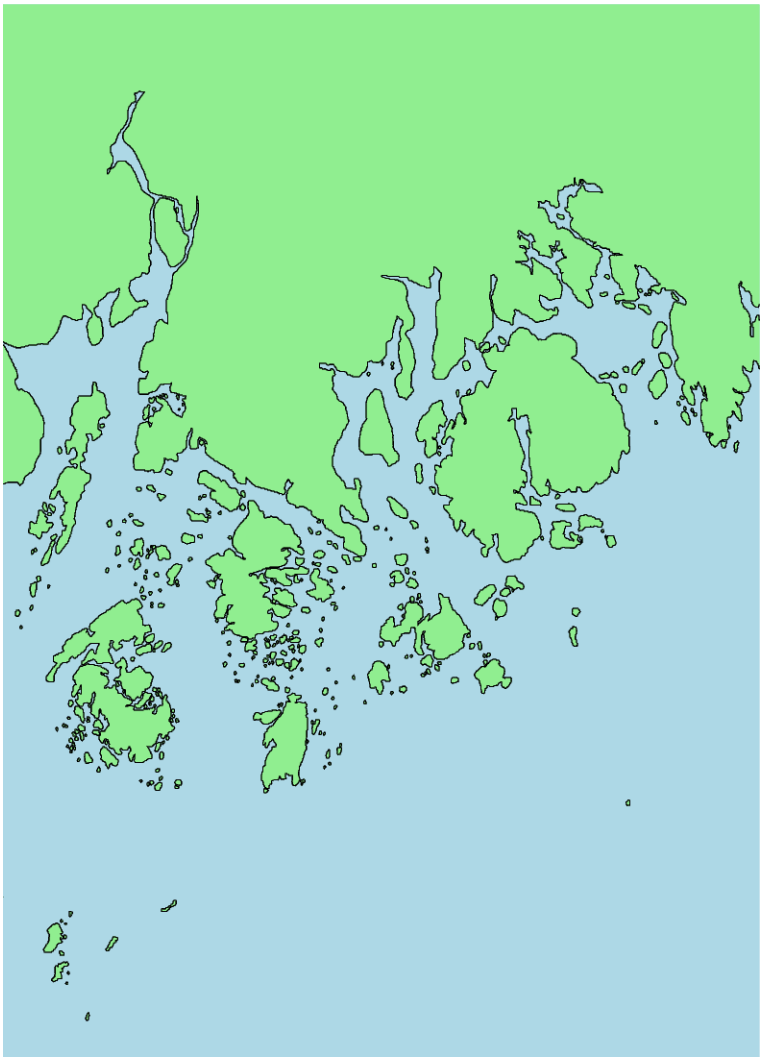
The appropriate projection varies for the type of map. The available projections are explained in the *projection* gallery. For this example, the Mercator projection is set using `"M"`. The width of the figure will be 10 centimeters, as set by `"10c"`. The map size can also be set in inches using "i" (e.g. a 5-inch wide Mercator projection would use `"M5i"`).

```
fig = pygmt.Figure()
fig.coast(
    region=[-69, -68, 43.75, 44.75],
```

(continues on next page)

(continued from previous page)

```
    shorelines=True,
    land="lightgreen",
    water="lightblue",
    projection="M10c",
)
fig.show()
```



3.1.5 Add a frame

While the map's colors, projection, and size have been set, the region that is being displayed is not apparent. A frame can be added to annotate the latitude and longitude of the region.

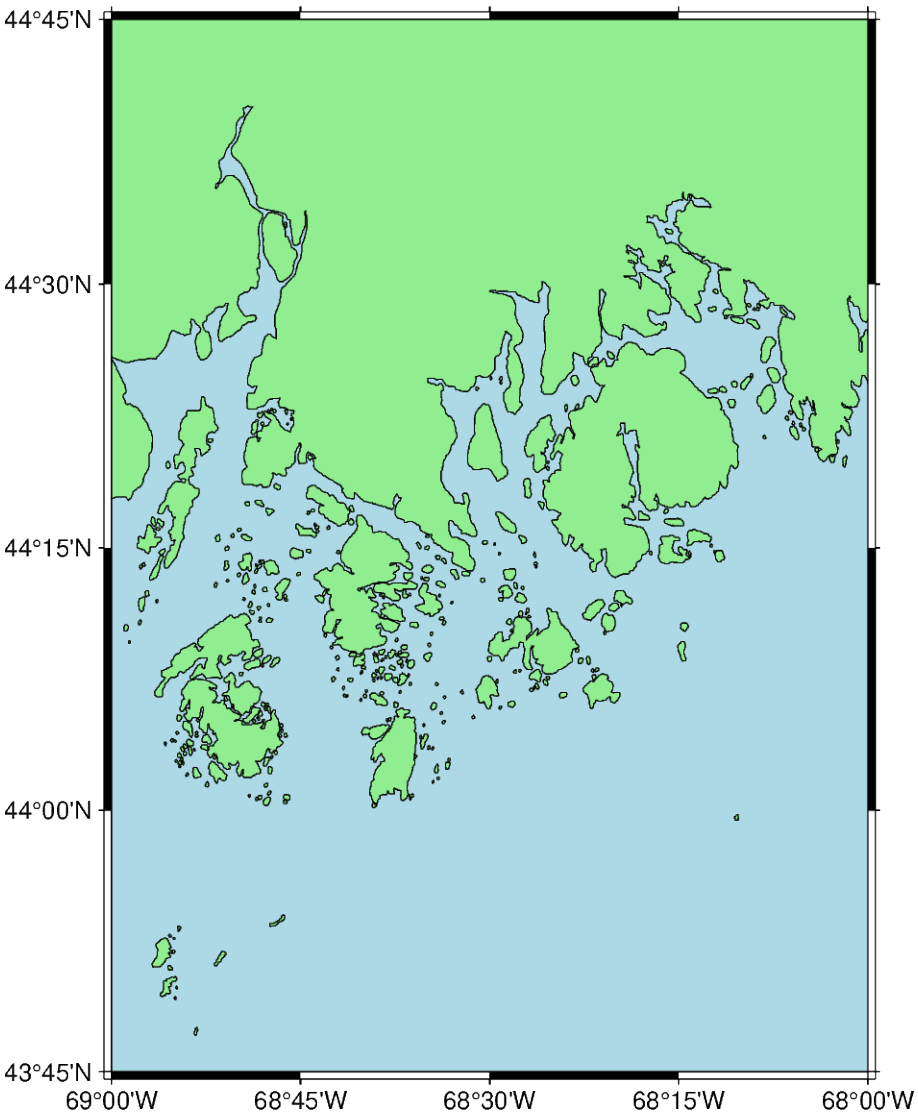
The `frame` parameter is used to add a frame to the figure. For now, it will be set to `"a"` to **a**nnotate the axes automatically.

```
fig = pygmt.Figure()
fig.coast(
    region=[-69, -68, 43.75, 44.75],
    shorelines=True,
    land="lightgreen",
```

(continues on next page)

(continued from previous page)

```
    water="lightblue",
    projection="M10c",
    frame="a",
)
fig.show()
```



3.1.6 Add a title

The `frame` parameter can be used to add a title to the figure. The title is set by passing `"+t"` followed by the title (e.g. setting the map title to "Title" would be `"+tTitle"`).

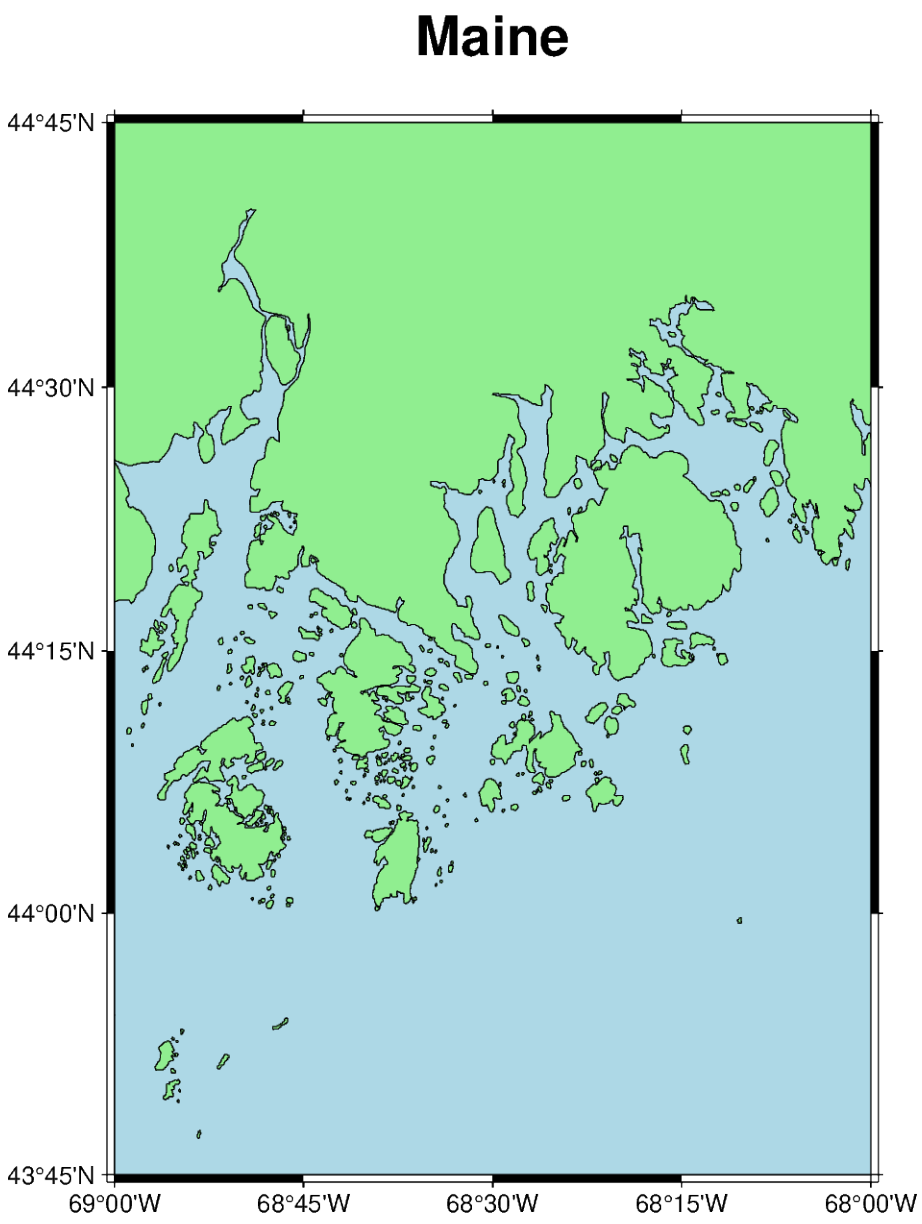
To pass multiple arguments to `frame`, a list can be used, as shown in the example below. This format uses `frame` to set both the axes annotations and the figure title.

```
fig = pygmt.Figure()
fig.coast(
    region=[-69, -68, 43.75, 44.75],
```

(continues on next page)

(continued from previous page)

```
    shorelines=True,
    land="lightgreen",
    water="lightblue",
    projection="M10c",
    frame=["a", "+tMaine"],
)
fig.show()
```



3.1.7 Additional exercises

This is the end of the first tutorial. Here are some additional exercises for the concepts that were discussed:

1. Make a map of Germany using its ISO country code ("DE"). Pass the ISO code as a Python string to the `region` parameter.

2. Change the color of the landmass to "khaki" and the water to "azure".
3. Change the color of the lakes (using the `lakes` parameter) to "red".
4. Create a global map. Set the region to "d" to center the map at the Prime Meridian or "g" to center the map at the International Date Line. When the region is set without using a list full of integers or floating numbers, the argument needs to be passed as a Python string. Create a map with a width of 15 centimeters using the Mollweide ("W") projection.

Total running time of the script: (0 minutes 1.029 seconds)

3.2 2. Create a contour map

This tutorial page covers the basics of creating a figure of the Earth relief, using a remote dataset hosted by GMT, using the method *pygmt.datasets.load_earth_relief*. It will use the *pygmt.Figure.grdimage*, *pygmt.Figure.grdcontour*, *pygmt.Figure.colorbar*, and *pygmt.Figure.coast* methods for plotting.

```
import pygmt
```

3.2.1 Loading the Earth relief dataset

The first step is to use *pygmt.datasets.load_earth_relief*. The `resolution` parameter sets the resolution of the remote grid file, which will affect the resolution of the plot made later in the tutorial. The `registration` parameter determines the grid registration.

This grid region covers the islands of Guam and Rota in the western Pacific Ocean.

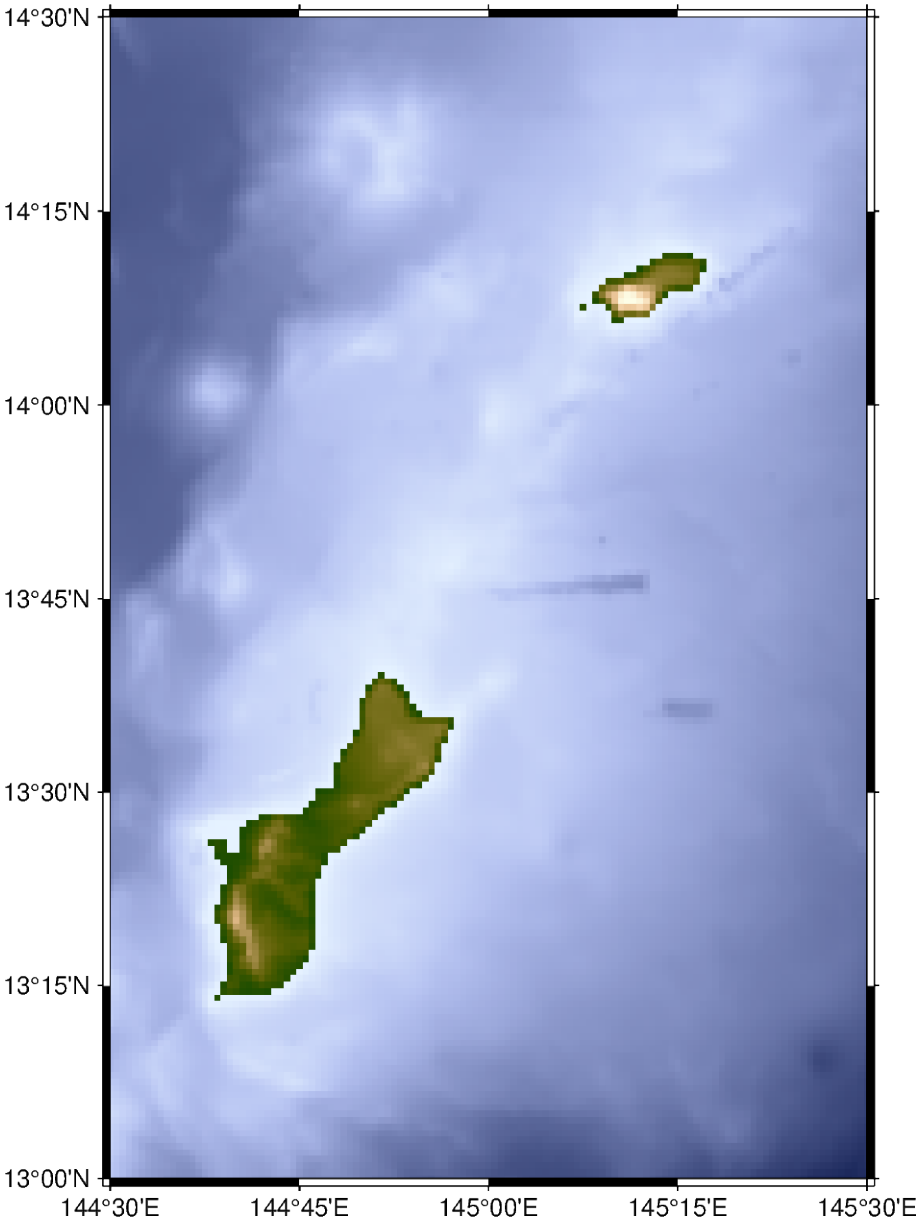
```
grid = pygmt.datasets.load_earth_relief(
    resolution="30s", region=[144.5, 145.5, 13, 14.5], registration="gridline"
)
```

3.2.2 Plotting Earth relief

To plot Earth relief data, the method *pygmt.Figure.grdimage* can be used to plot a color-coded figure to display the topography and bathymetry in the grid file. The `grid` parameter accepts the input grid, which in this case is the remote file downloaded in the previous step. If the `region` parameter is not set, the region boundaries of the input grid are used.

The `cmap` parameter sets the color palette table (CPT) used for portraying the Earth relief. The *pygmt.Figure.grdimage* method uses the input grid to relate the Earth relief values to a specific color within the CPT. In this case, the CPT "oleron" is used; a full list of CPTs can be found at https://docs.generic-mapping-tools.org/6.5/reference/cpts.html.

```
fig = pygmt.Figure()
fig.grdimage(grid=grid, frame="a", projection="M10c", cmap="oleron")
fig.show()
```



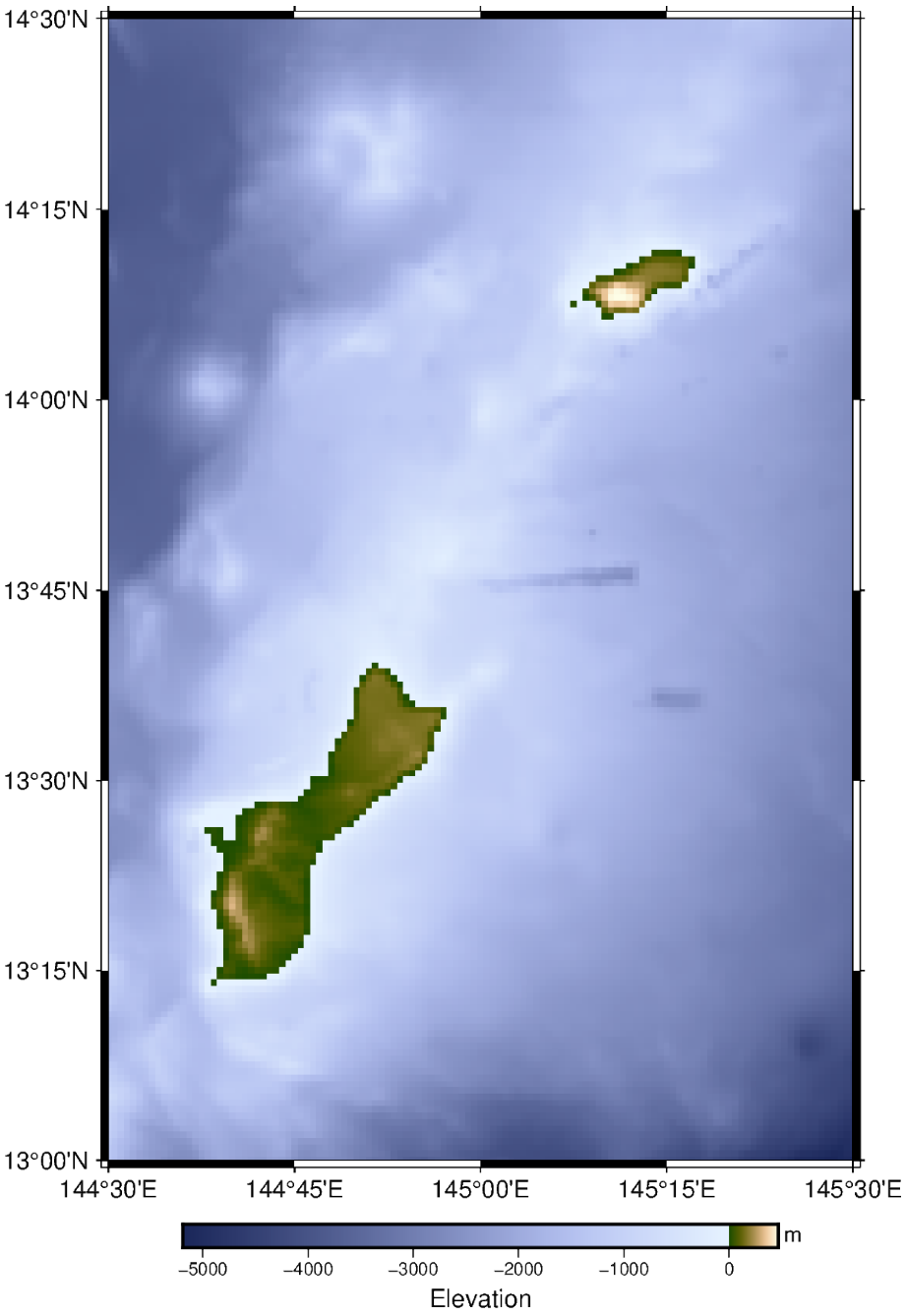
3.2.3 Adding a colorbar

To show how the plotted colors relate to the Earth relief, a colorbar can be added using the *pygmt.Figure.colorbar* method.

To control the annotation and labels on the colorbar, a list is passed to the `frame` parameter. The value beginning with `"a"` sets the interval for the annotation on the colorbar, in this case every 1,000 meters. To set the label for an axis on the colorbar, the argument begins with either `"x+l"` (x-axis) or `"y+l"` (y-axis), followed by the intended label.

By default, the CPT for the colorbar is the same as the one set in *pygmt.Figure.grdimage*.

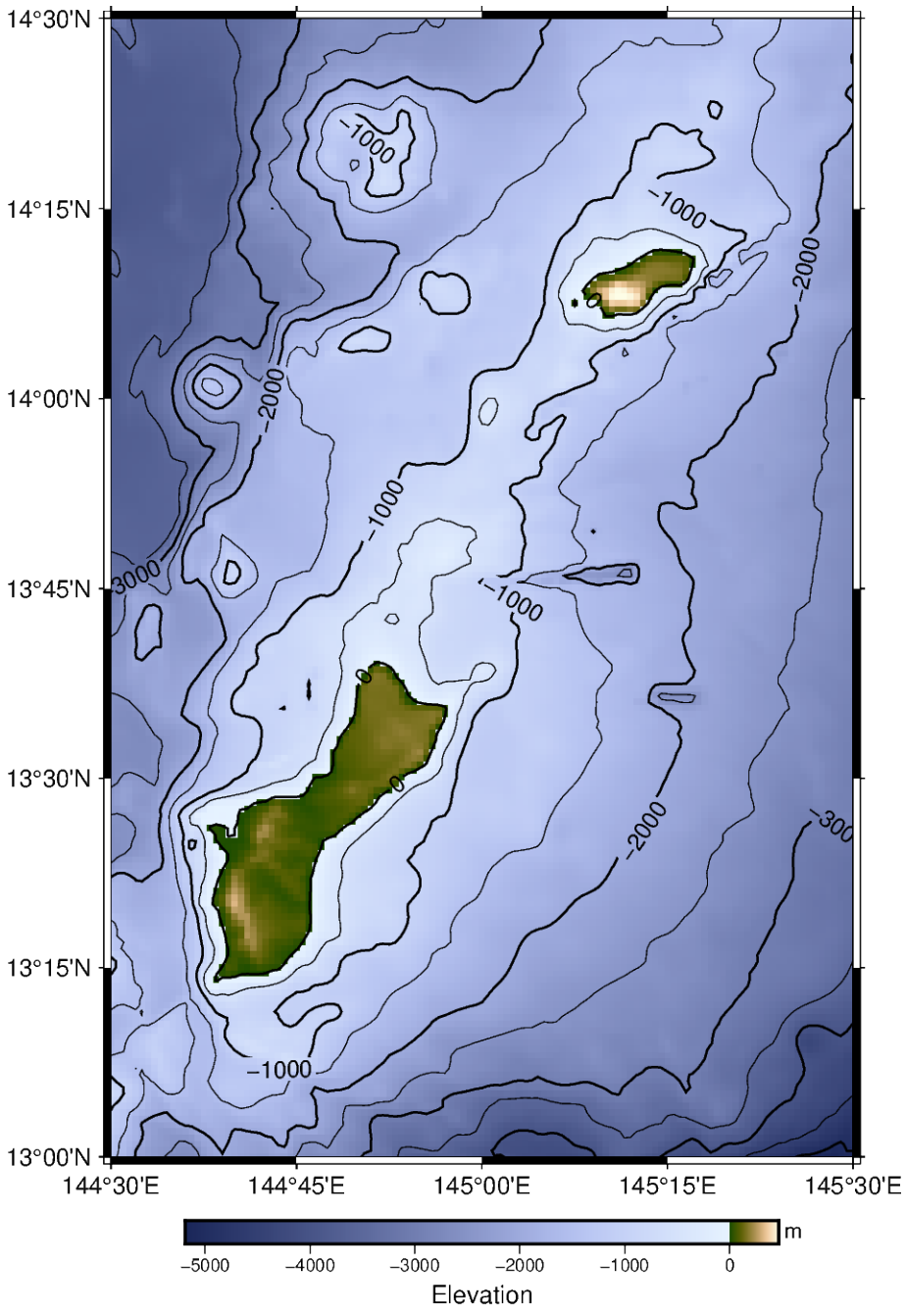
```
fig = pygmt.Figure()
fig.grdimage(grid=grid, frame="a", projection="M10c", cmap="oleron")
fig.colorbar(frame=["a1000", "x+lElevation", "y+lm"])
fig.show()
```



3.2.4 Adding contour lines

To add contour lines to the color-coded figure, the *pygmt.Figure.grdcontour* method is used. The `frame` and `projection` are already set using *pygmt.Figure.grdimage* and are not needed again. However, the same input for `grid` (in this case, the variable named "grid") must be input again. The `levels` parameter sets the spacing between adjacent contour lines (in this case, 500 meters). The `annotation` parameter annotates the contour lines corresponding to the given interval (in this case, 1,000 meters) with the related values, here elevation or bathymetry. By default, these contour lines are drawn thicker. Optionally, the appearance (thickness, color, style) of the annotated and the not-annotated contour lines can be adjusted (separately) by specifying the desired `pen`.

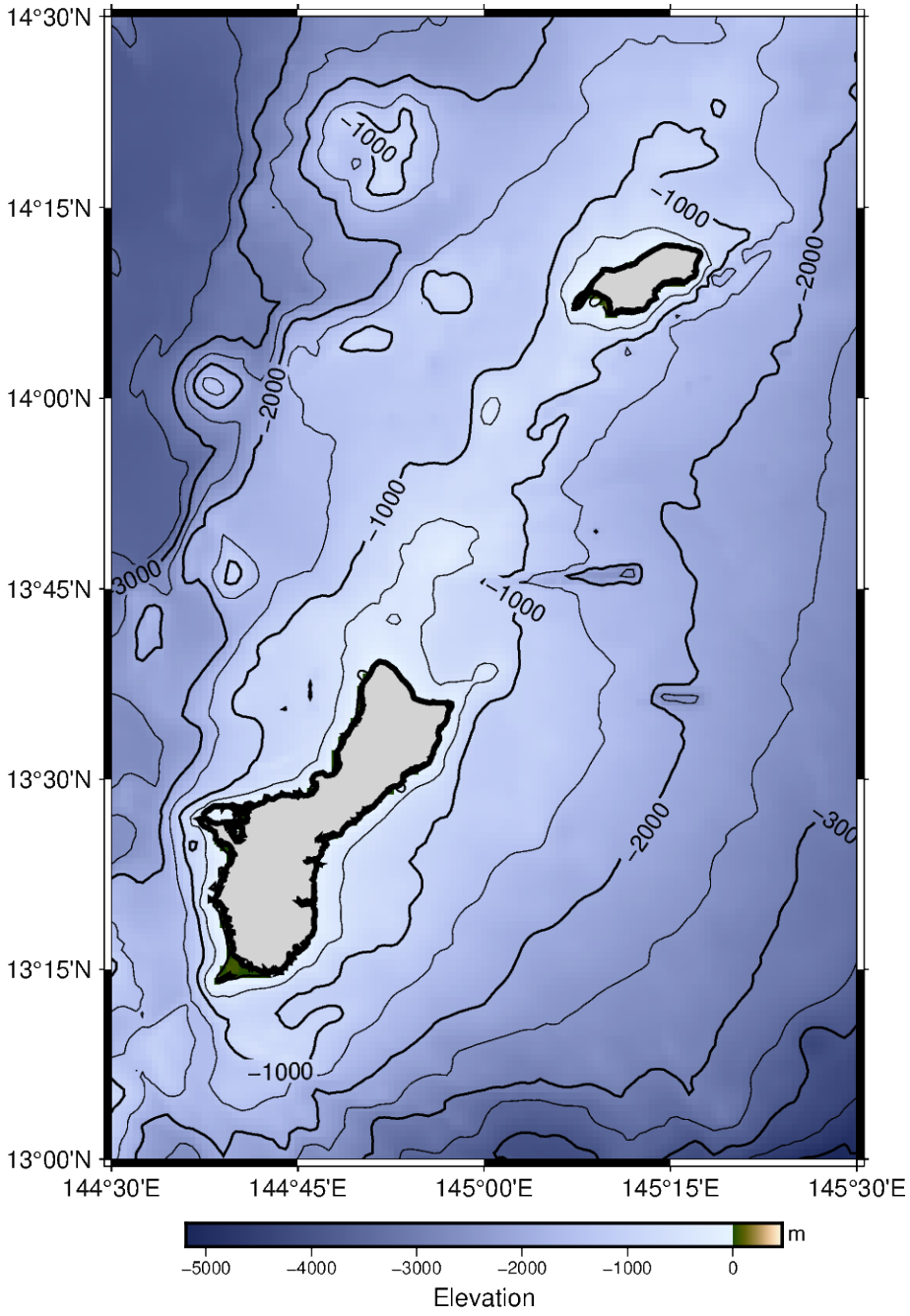
```
fig = pygmt.Figure()
fig.grdimage(grid=grid, frame="a", projection="M10c", cmap="oleron")
fig.grdcontour(grid=grid, levels=500, annotation=1000)
fig.colorbar(frame=["a1000", "x+lElevation", "y+lm"])
fig.show()
```



3.2.5 Color in land

To make it clear where the islands are located, the *pygmt.Figure.coast* method can be used to color in the landmasses. The `land` is colored in as "lightgray", and the `shorelines` parameter draws a border around the islands.

```
fig = pygmt.Figure()
fig.grdimage(grid=grid, frame="a", projection="M10c", cmap="oleron")
fig.grdcontour(grid=grid, levels=500, annotation=1000)
fig.coast(shorelines="2p", land="lightgray")
fig.colorbar(frame=["a1000", "x+lElevation", "y+lm"])
fig.show()
```



3.2.6 Additional exercises

This is the end of the second tutorial. Here are some additional exercises for the concepts that were discussed:

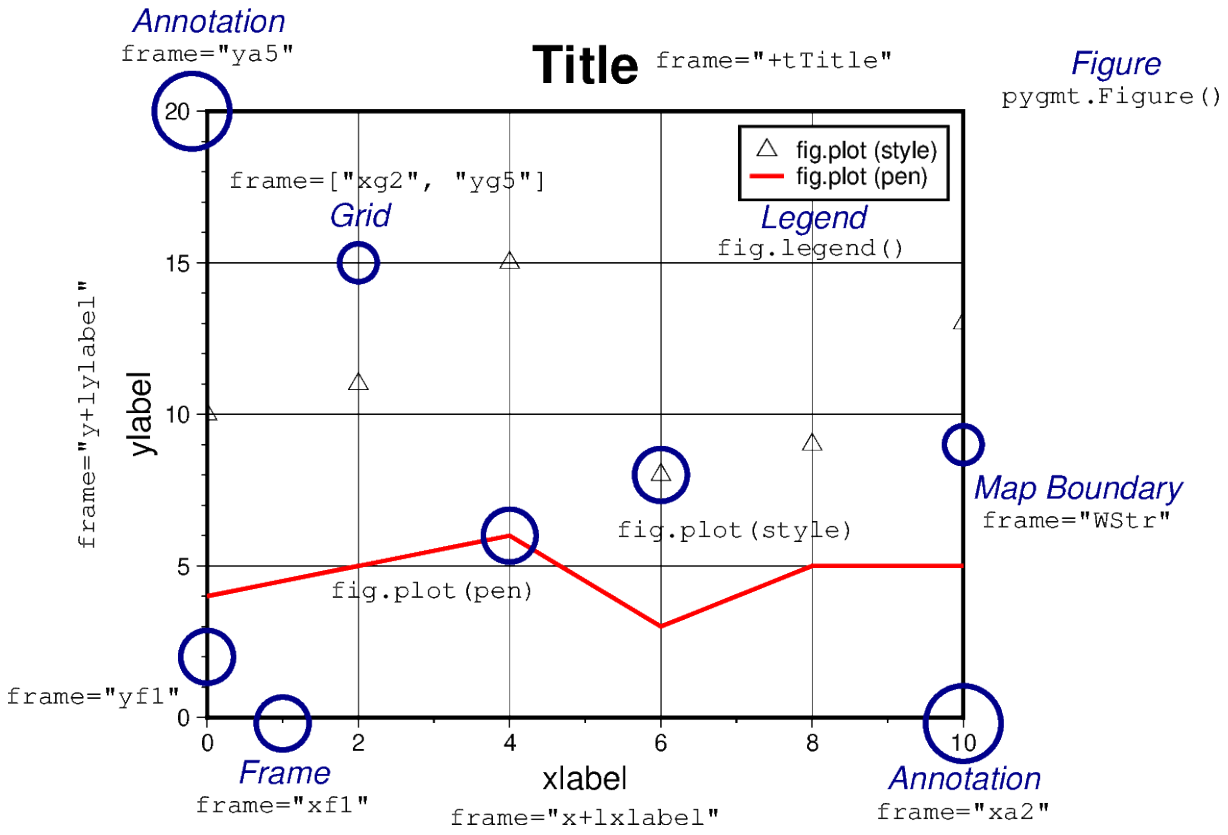
1. Change the resolution of the grid file to either `"01m"` (1 arc-minute, a lower resolution) or `"15s"` (15 arc-seconds, a higher resolution). Note that higher resolution grids will have larger file sizes. Available resolutions can be found at *pygmt.datasets.load_earth_relief*.
2. Create a contour map of the area around Mt. Rainier. A suggestion for the `region` would be `[-122, -121, 46.5, 47.5]`. Adjust the *pygmt.Figure.grdcontour* and *pygmt.Figure.colorbar* settings as needed to make the figure look good.
3. Create a contour map of São Miguel Island in the Azores; a suggested `region` is `[-26, -25, 37.5, 38]`. Instead of coloring in `land`, set `water` to "lightblue" to only display Earth relief information for the land.
4. Try other CPTs, such as "SCM/fes" or "geo".

Total running time of the script: (0 minutes 0.989 seconds)

3.3 3. Figure elements

The figure below shows the naming of figure elements in PyGMT.

- *pygmt.Figure*: having a number of plotting methods. Every new figure must start with the creation of a *pygmt.Figure* instance
- `frame`: setting plot or map boundaries (a combination of the single letters of **WSNE**, **wsne**, or **lbtr**), adding annotations, ticks, gridlines (**afg**), axis labels (**+l**), and title (**+t**), e.g., in *pygmt.Figure.basemap*. Detailed examples can be found at *frame and axes attributes*
- *pygmt.Figure.plot*: plotting lines or symbols based on `pen` or `style` parameters, respectively
- *pygmt.Figure.text*: plotting text strings whereby the `font` parameter adjusts fontsize, fontstyle, and color
- *pygmt.Figure.legend*: showing the naming of lines or symbols while the `label` is given in *pygmt.Figure.plot*
- *pygmt.Figure.show*: previewing the content added to the current figure instance




```
import pygmt

fig = pygmt.Figure()

x = range(0, 11, 2)
y_1 = [10, 11, 15, 8, 9, 13]
y_2 = [4, 5, 6, 3, 5, 5]

fig.basemap(
    region=[0, 10, 0, 20],
    projection="X10c/8c",
    frame=["WStr+tTitle", "xa2f1g2+lxlabel", "ya5f1g5+lylabel"],
)
fig.plot(x=x, y=y_1, style="t0.3c", label="fig.plot (style)")
fig.plot(x=x, y=y_2, pen="1.5p,red", label="fig.plot (pen)")

mainexplain = {"font": "12p,2,darkblue", "justify": "TC", "no_clip": True}
minorexplain = {"font": "10p,8", "justify": "TC", "no_clip": True}
# ============ Figure
fig.text(x=12, y=22, text="Figure", **mainexplain)
fig.text(x=12, y=20.8, text="pygmt.Figure()", **minorexplain)
# ============ Title
fig.text(x=7.5, y=22, text='frame="+tTitle"', **minorexplain)
# ============ xlabel
fig.text(x=5, y=-3, text='frame="x+lxlabel"', **minorexplain)
# ============ ylabel
fig.text(x=-1.7, y=10, text='frame="y+lylabel"', angle=90, **minorexplain)
```

(continues on next page)

(continued from previous page)

```
# ============ x-majorticks
fig.plot(x=10, y=-0.2, style="c1c", pen="2p,darkblue", no_clip=True)
fig.text(x=10, y=-1.6, text="Annotation", **mainexplain)
fig.text(x=10, y=-2.8, text='frame="xa2"', **minorexplain)
# ============ y-majorticks
fig.plot(x=-0.2, y=20, style="c1c", pen="2p,darkblue", no_clip=True)
fig.text(x=0, y=23.4, text="Annotation", **mainexplain)
fig.text(x=0, y=22.2, text='frame="ya5"', **minorexplain)
# ============ x-minorticks
fig.plot(x=1, y=-0.2, style="c0.7c", pen="2p,darkblue", no_clip=True)
fig.text(x=1, y=-1.4, text="Frame", **mainexplain)
fig.text(x=1, y=-2.6, text='frame="xf1"', **minorexplain)
# ============ y-minorticks
fig.plot(x=0, y=2, style="c0.7c", pen="2p,darkblue", no_clip=True)
fig.text(x=-1.5, y=1, text='frame="yf1"', **minorexplain)
# ============ Grid
fig.plot(x=2, y=15, style="c0.5c", pen="2p,darkblue")
fig.text(x=2, y=17, text="Grid", **mainexplain)
fig.text(x=2.4, y=18, text='frame=["xg2", "yg5"]', **minorexplain)
# ============ Map Boundaries
fig.plot(x=10, y=9, style="c0.5c", pen="2p,darkblue", no_clip=True)
fig.text(x=11.5, y=8, text="Map Boundary", **mainexplain)
fig.text(x=11.5, y=6.8, text='frame="WStr"', **minorexplain)
# ============ fig.plot (style)
fig.plot(x=6, y=8, style="c0.7c", pen="2p,darkblue")
fig.text(x=7, y=6.5, text="fig.plot(style)", **minorexplain)
# ============ fig.plot (pen)
fig.plot(x=4, y=6, style="c0.7c", pen="2p,darkblue")
fig.text(x=3, y=4.5, text="fig.plot(pen)", **minorexplain)
# ============ Legend
fig.legend()
fig.text(x=8, y=16.9, text="Legend", **mainexplain)
fig.text(x=8, y=15.8, text="fig.legend()", **minorexplain)

fig.show()
```

Total running time of the script: (0 minutes 0.310 seconds)

3.4 4. PyGMT I/O: Table inputs

Generally, PyGMT accepts two different types of data inputs: tables and grids.

- A table is a 2-D array with rows and columns. Each column represents a different variable (e.g., x, y and z) and each row represents a different record.
- A grid is a 2-D array of data that is regularly spaced in the x and y directions (or longitude and latitude).

In this tutorial, we'll focus on working with table inputs, and cover grid inputs in a separate tutorial.

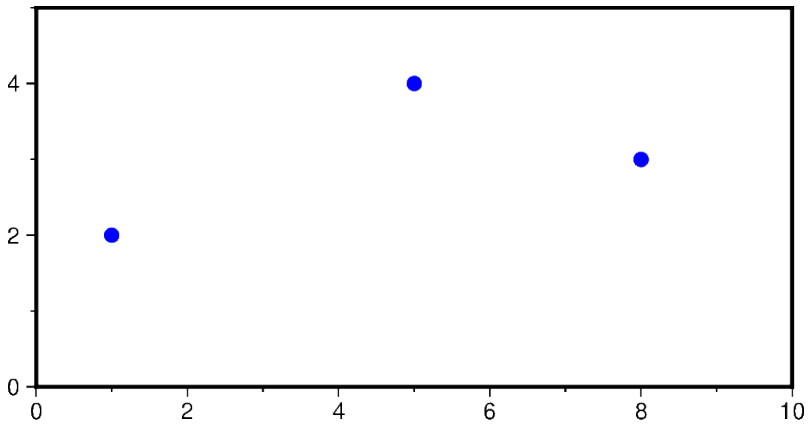
PyGMT supports a variety of table input types that allow you to work with data in a format that suits your needs. In this tutorial, we'll explore the different table input types available in PyGMT and provide examples for each. By understanding the different table input types, you can choose the one that best fits your data and analysis needs, and work more efficiently with PyGMT.

```
from pathlib import Path

import geopandas as gpd
import numpy as np
import pandas as pd
import pygmt
```

3.4.1 ASCII table file

Most PyGMT functions/methods that accept table input data have a `data` parameter. The easiest way to provide table input data to PyGMT is by specifying the file name of an ASCII table (e.g., `data="input_data.dat"`). This is useful when your data is stored in a separate text file.

```
# Create an example file with 3 rows and 2 columns
data = np.array([[1.0, 2.0], [5.0, 4.0], [8.0, 3.0]])
np.savetxt("input_data.dat", data, fmt="%f")

# Pass the file name to the data parameter
fig = pygmt.Figure()
fig.basemap(region=[0, 10, 0, 5], projection="X10c/5c", frame=True)
fig.plot(data="input_data.dat", style="p0.2c", fill="blue")
fig.show()

# Now let's delete the example file
Path("input_data.dat").unlink()
```



Besides a plain string to a table file, the following variants are also accepted:

- A `pathlib.Path` object.
- A full URL. PyGMT will download the file to the current directory first.
- A file name prefixed with `@` (e.g., `data="@input_data.dat"`), which is a special syntax in GMT to indicate that the file is a remote file hosted on the GMT data server.

Additionally, PyGMT also supports a list of file names, `pathlib.Path` objects, URLs, or remote files, to provide more flexibility in specifying input files.

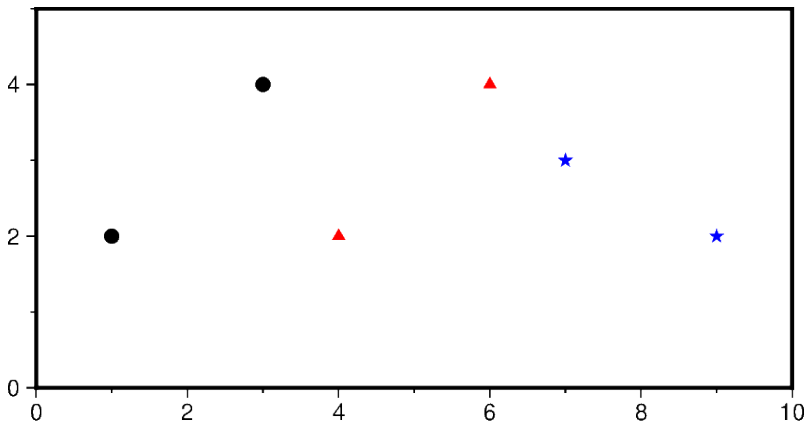
3.4.2 2-D array: *list*, *numpy.ndarray*, and *pandas.DataFrame*

The `data` parameter also accepts a 2-D array, e.g.,

- A 2-D `list` (i.e., a list of lists)
- A `numpy.ndarray` object with with a dimension of 2
- A `pandas.DataFrame` object

This is useful when you want to plot data that is already in memory.

```
fig = pygmt.Figure()
fig.basemap(region=[0, 10, 0, 5], projection="X10c/5c", frame=True)

# Pass a 2-D list to the 'data' parameter
fig.plot(data=[[1.0, 2.0], [3.0, 4.0]], style="c0.2c", fill="black")

# Pass a 2-D numpy array to the 'data' parameter
fig.plot(data=np.array([[4.0, 2.0], [6.0, 4.0]]), style="t0.2c", fill="red")

# Pass a pandas.DataFrame to the 'data' parameter
df = pd.DataFrame(np.array([[7.0, 3.0], [9.0, 2.0]]), columns=["x", "y"])
fig.plot(data=df, style="a0.2c", fill="blue")

fig.show()
```



3.4.3 `geopandas.GeoDataFrame`

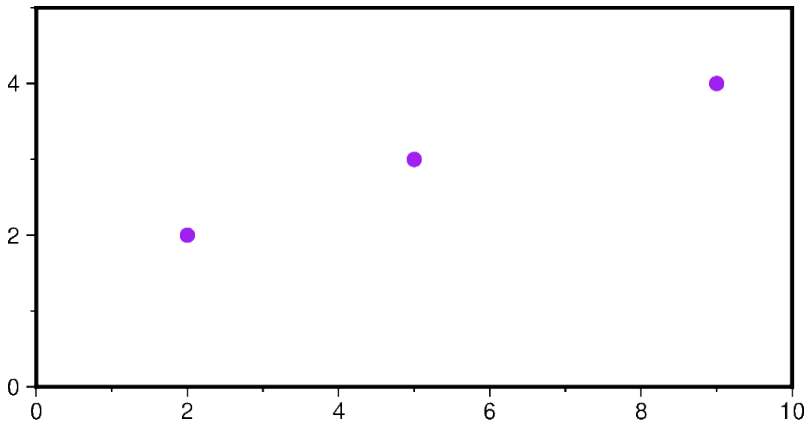
If you're working with geospatial data, you can read your data as a `geopandas.GeoDataFrame` object and pass it to the `data` parameter. This is useful if your data is stored in a geospatial data format (e.g., GeoJSON, etc.) that GMT and PyGMT do not support natively.

```
# Example GeoDataFrame
gdf = gpd.GeoDataFrame(
    {
        "geometry": gpd.points_from_xy([2, 5, 9], [2, 3, 4]),
        "value": [10, 20, 30],
    }
)

# Use the GeoDataFrame to specify the 'data' parameter
```

(continues on next page)

(continued from previous page)

```
fig = pygmt.Figure()
fig.basemap(region=[0, 10, 0, 5], projection="X10c/5c", frame=True)
fig.plot(data=gdf, style="c0.2c", fill="purple")
fig.show()
```

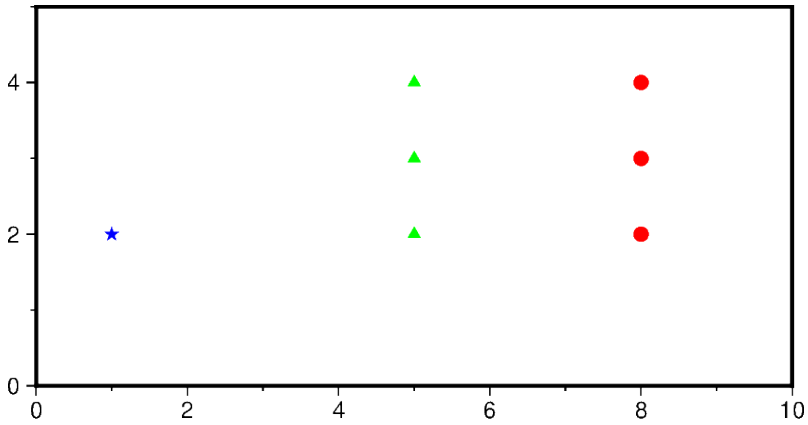



```
/home/runner/micromamba/envs/pygmt/lib/python3.13/site-packages/pyogrio/geopandas.
↪py:662: UserWarning: 'crs' was not provided.  The output dataset will not have␣
↪projection information defined and may not be usable in other systems.
  write(
```

3.4.4 Scalar values or 1-D arrays

In addition to the `data` parameter, some PyGMT functions/methods also provide individual parameters (e.g., `x` and `y` for data coordinates) which allow you to specify the data. These parameters accept individual scalar values or 1-D arrays (lists or 1-D numpy arrays).

```
fig = pygmt.Figure()
fig.basemap(region=[0, 10, 0, 5], projection="X10c/5c", frame=True)

# Pass scalar values to plot a single data point
fig.plot(x=1.0, y=2.0, style="a0.2c", fill="blue")

# Pass 1-D lists to plot multiple data points
fig.plot(x=[5.0, 5.0, 5.0], y=[2.0, 3.0, 4.0], style="t0.2c", fill="green")

# Pass 1-D numpy arrays to plot multiple data points
fig.plot(
    x=np.array([8.0, 8.0, 8.0]), y=np.array([2.0, 3.0, 4.0]), style="c0.2c", fill="red
↪"
)

fig.show()
```



3.4.5 Conclusion

In PyGMT, you have the flexibility to provide data in various table input types, including file names, 2-D arrays (2-D `list`, `numpy.ndarray`, `pandas.DataFrames`), scalar values or a series of 1-D arrays, and `geopandas.GeoDataFrame`. Choose the input type that best suits your data source and analysis requirements.

Total running time of the script: (0 minutes 0.449 seconds)

CHAPTER

FOUR

TUTORIALS

These examples teach us how to complete various tasks using PyGMT!

4.1 Basics

4.1.1 Coastlines and borders

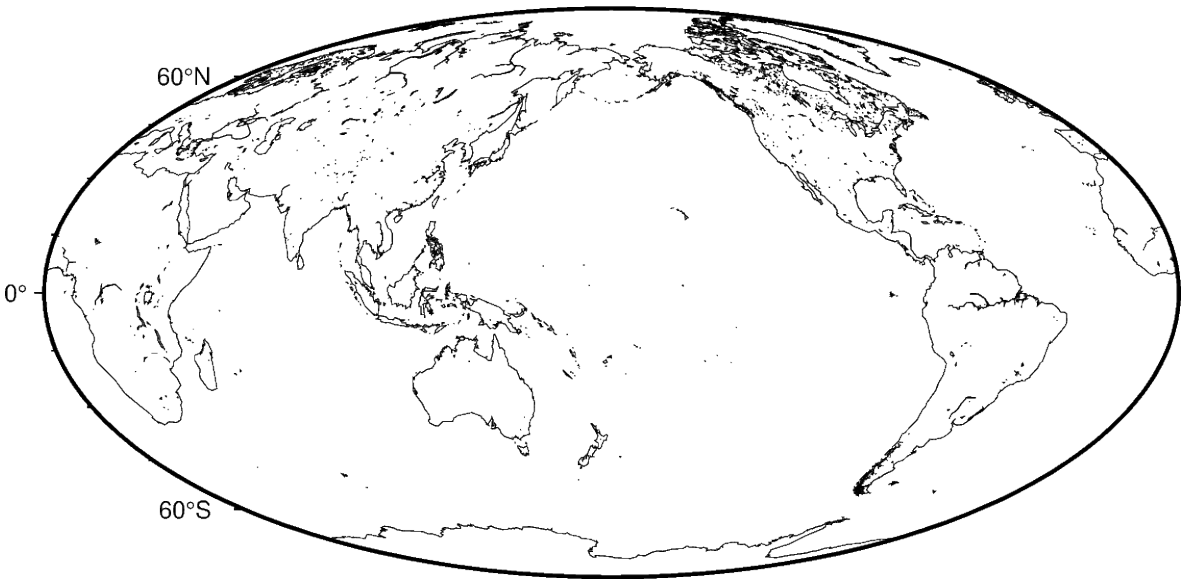
Plotting coastlines and borders is handled by *pygmt.Figure.coast*.

```
import pygmt
```

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()
```



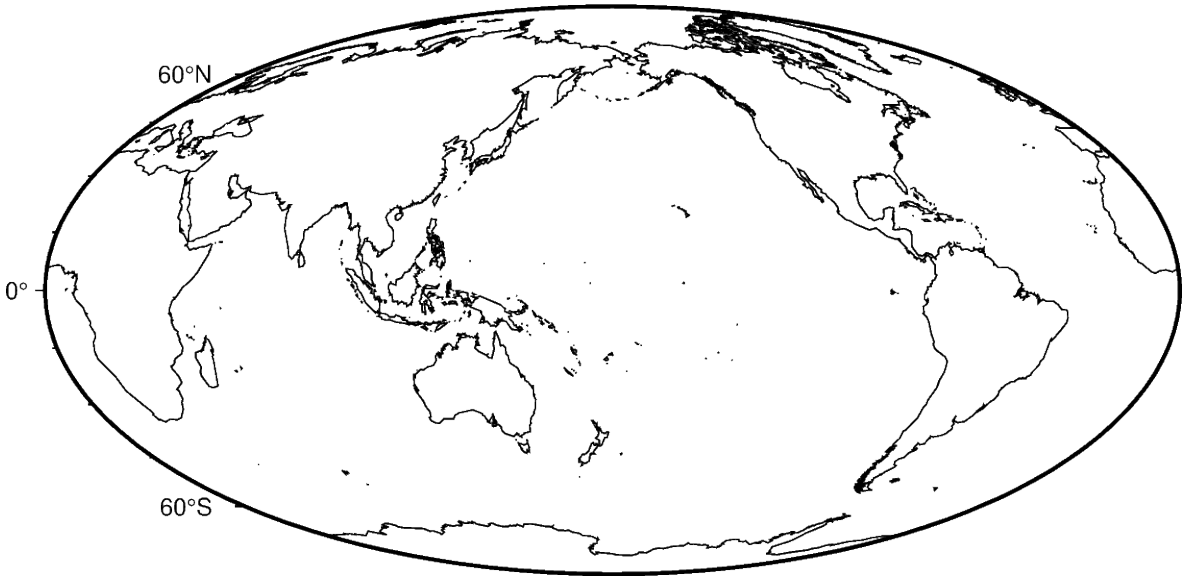
The shorelines are divided in 4 levels:

1. coastline

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

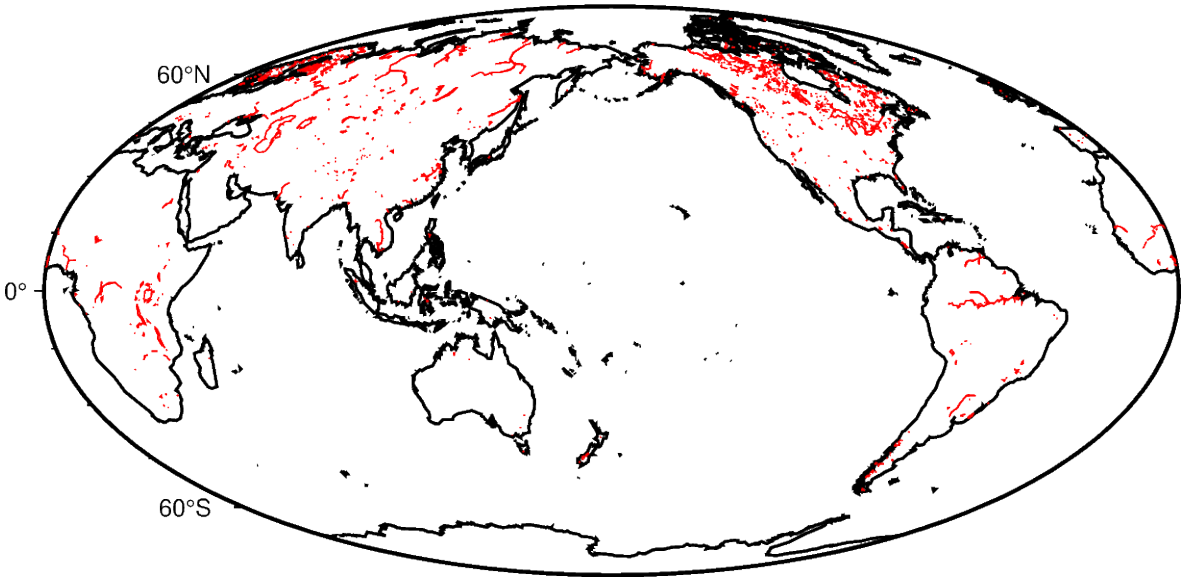
You can specify which level you want to plot by passing the level number and a GMT pen configuration. For example, to plot just the coastlines with 0.5p thickness and black lines:

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



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()
```

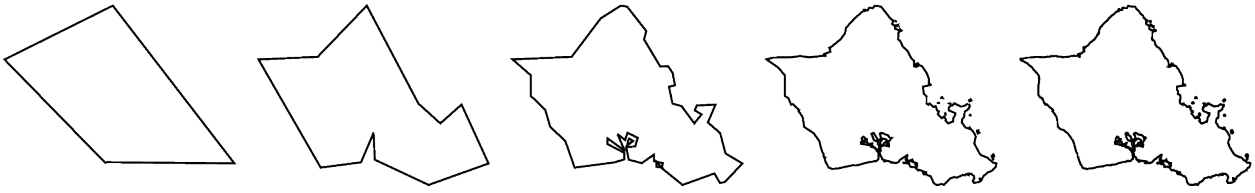


Resolutions

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

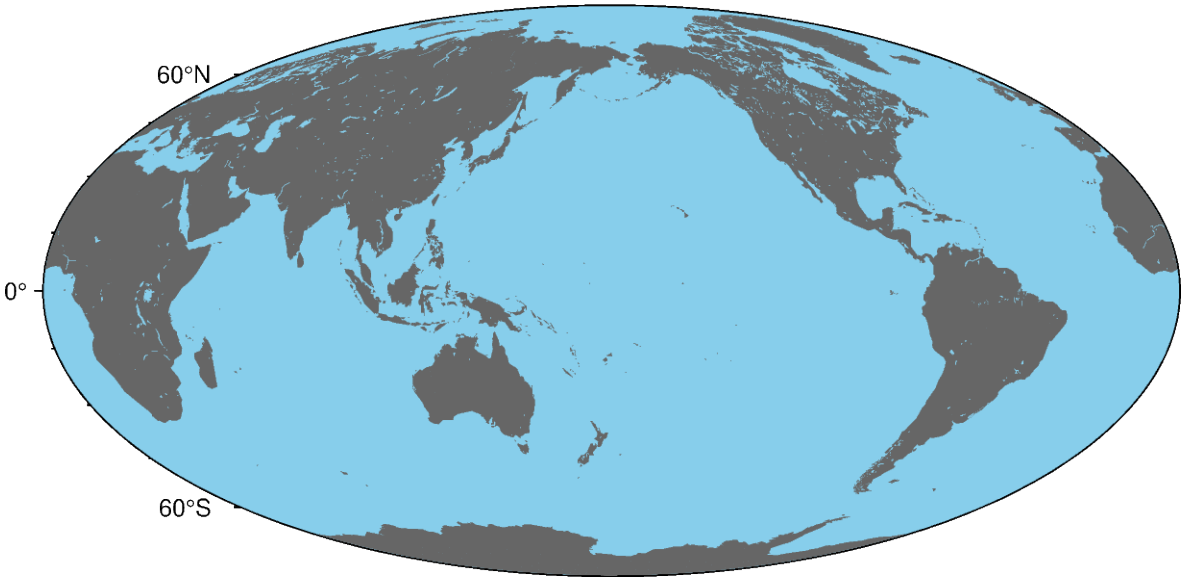
```
oahu = [-158.3, -157.6, 21.2, 21.8]
fig = pygmt.Figure()
for res in ["c", "l", "i", "h", "f"]:
    fig.coast(resolution=res, shorelines="1p", region=oahu, projection="M5c")
    fig.shift_origin(xshift="5c")
fig.show()
```



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()
```



Total running time of the script: (0 minutes 1.090 seconds)

4.1.2 Frames, ticks, titles, and labels

Setting frame, ticks, title, etc., of the plot is handled by the `frame` parameter that most plotting methods of the *pygmt.Figure* class contain.

```
import pygmt
```

Plot frame

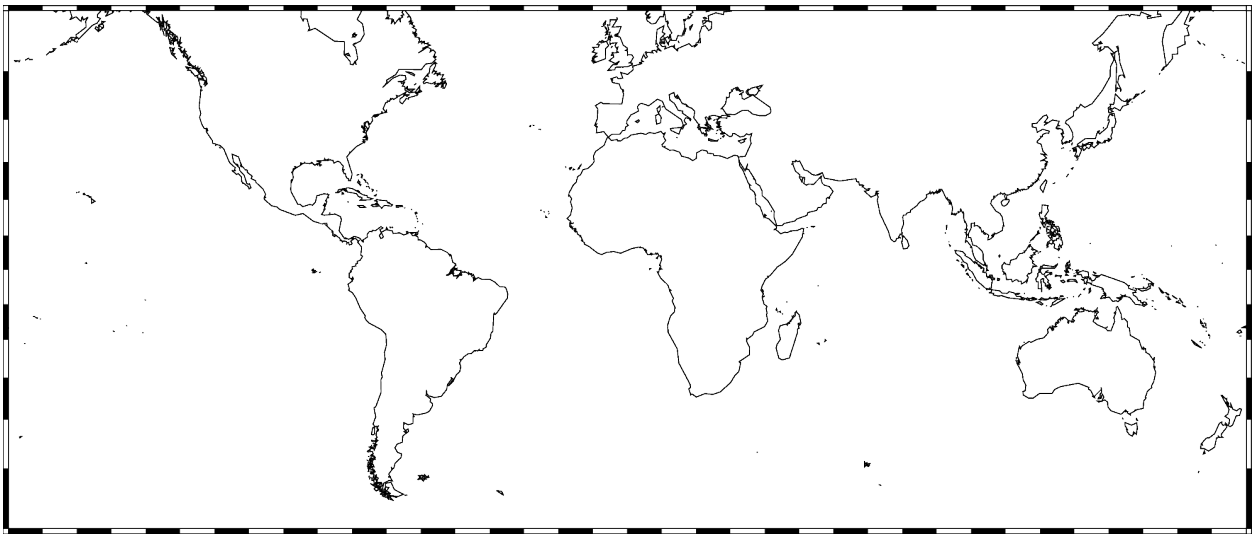
By default, PyGMT does not add a frame to your plot. For example, we can plot the coastlines of the world with a Mercator projection:

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



To add the default GMT frame style to the plot, use `frame="f"` in *pygmt.Figure.basemap* or another plotting method (which has the `frame` parameter, with the exception of *pygmt.Figure.colorbar*):

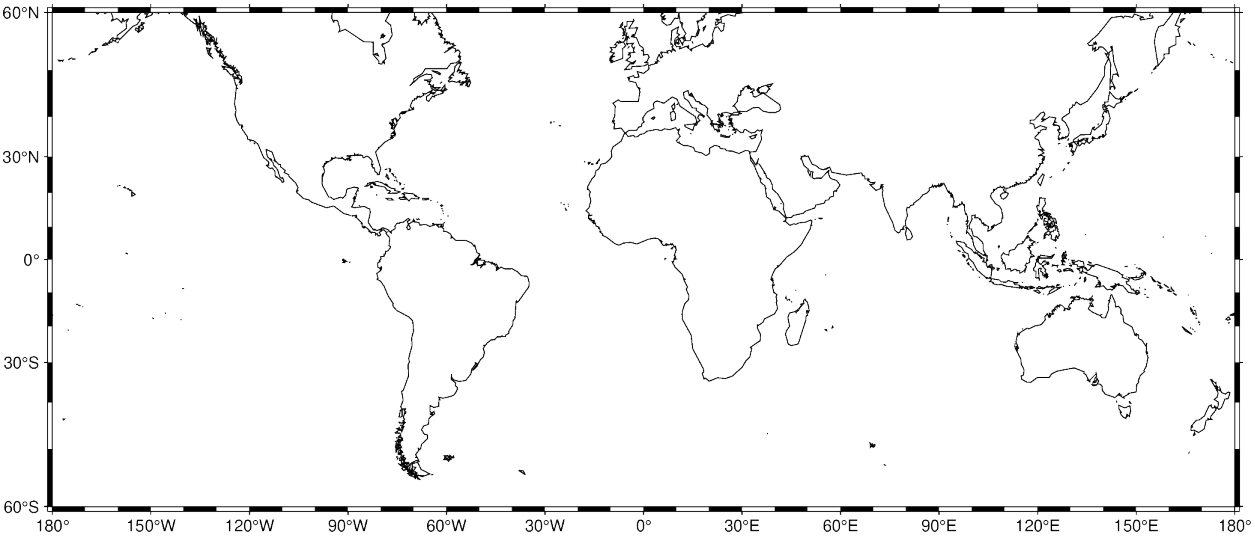
```
fig = pygmt.Figure()
fig.coast(shorelines="1/0.5p", region=[-180, 180, -60, 60], projection="M25c")
fig.basemap(frame="f")
fig.show()
```



Ticks and grid lines

The automatic frame (`frame=True` or `frame="af"`) adds the default GMT frame style and automatically determines tick labels from the plot region. In GMT the tick labels are called **a**nnotations.

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



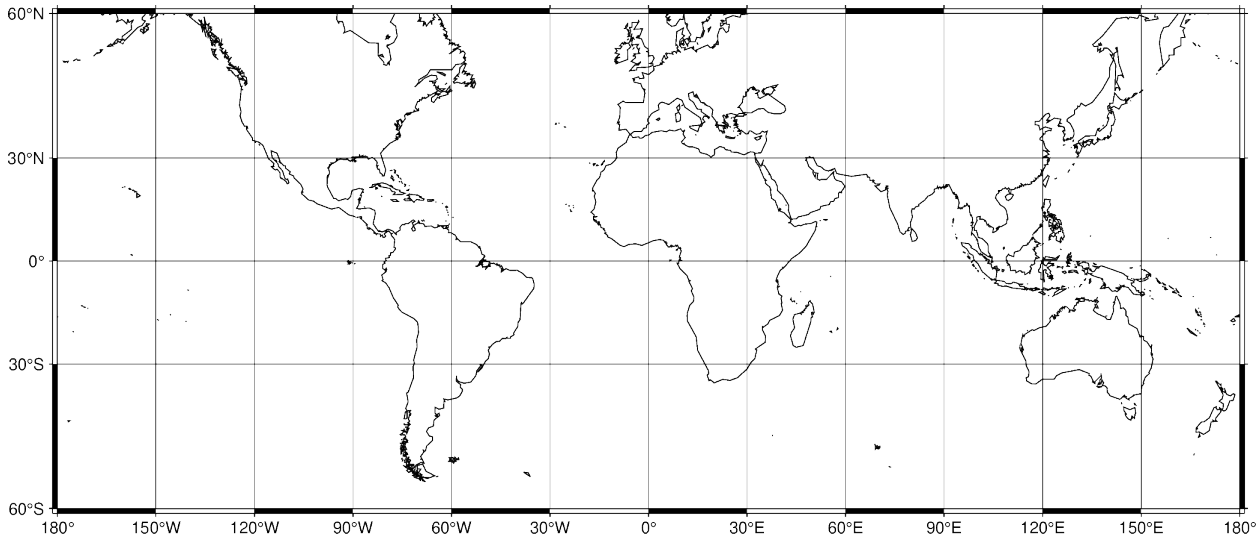
Add automatic grid lines to the plot by passing `g` through the `frame` parameter:

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

(continues on next page)

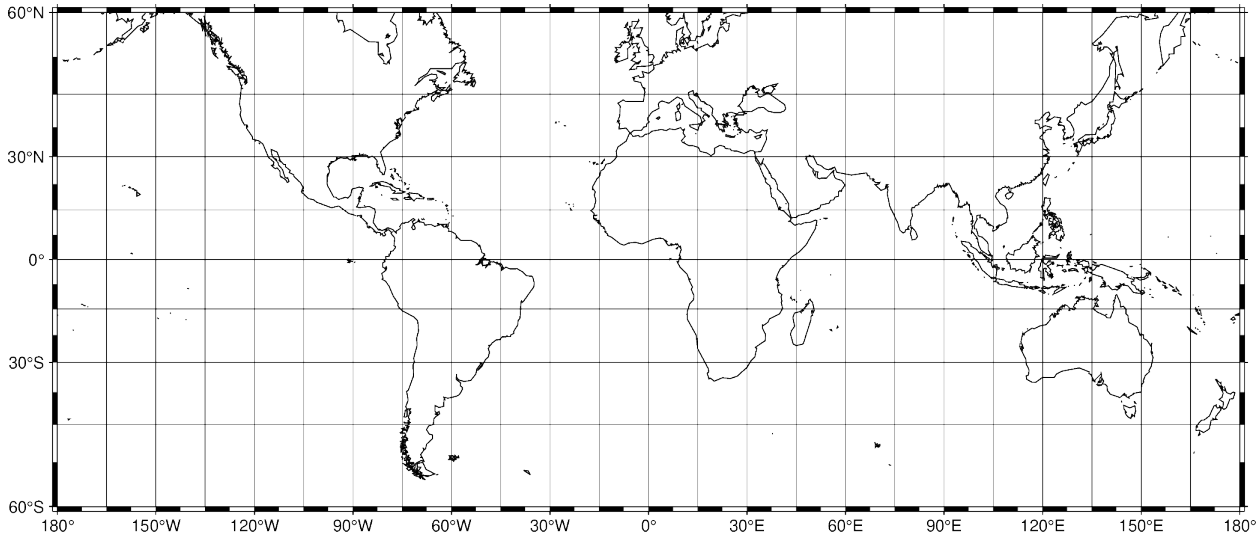
(continued from previous page)

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



To adjust the step widths of annotations, frame, and grid lines we can add the desired step widths after `a`, `f`, or `g`. In the example below, the step widths are set to 30°, 7.5°, and 15°, respectively.

```
fig = pygmt.Figure()
fig.coast(shorelines="1/0.5p", region=[-180, 180, -60, 60], projection="M25c")
fig.basemap(frame="a30f7.5g15")
fig.show()
```



Title

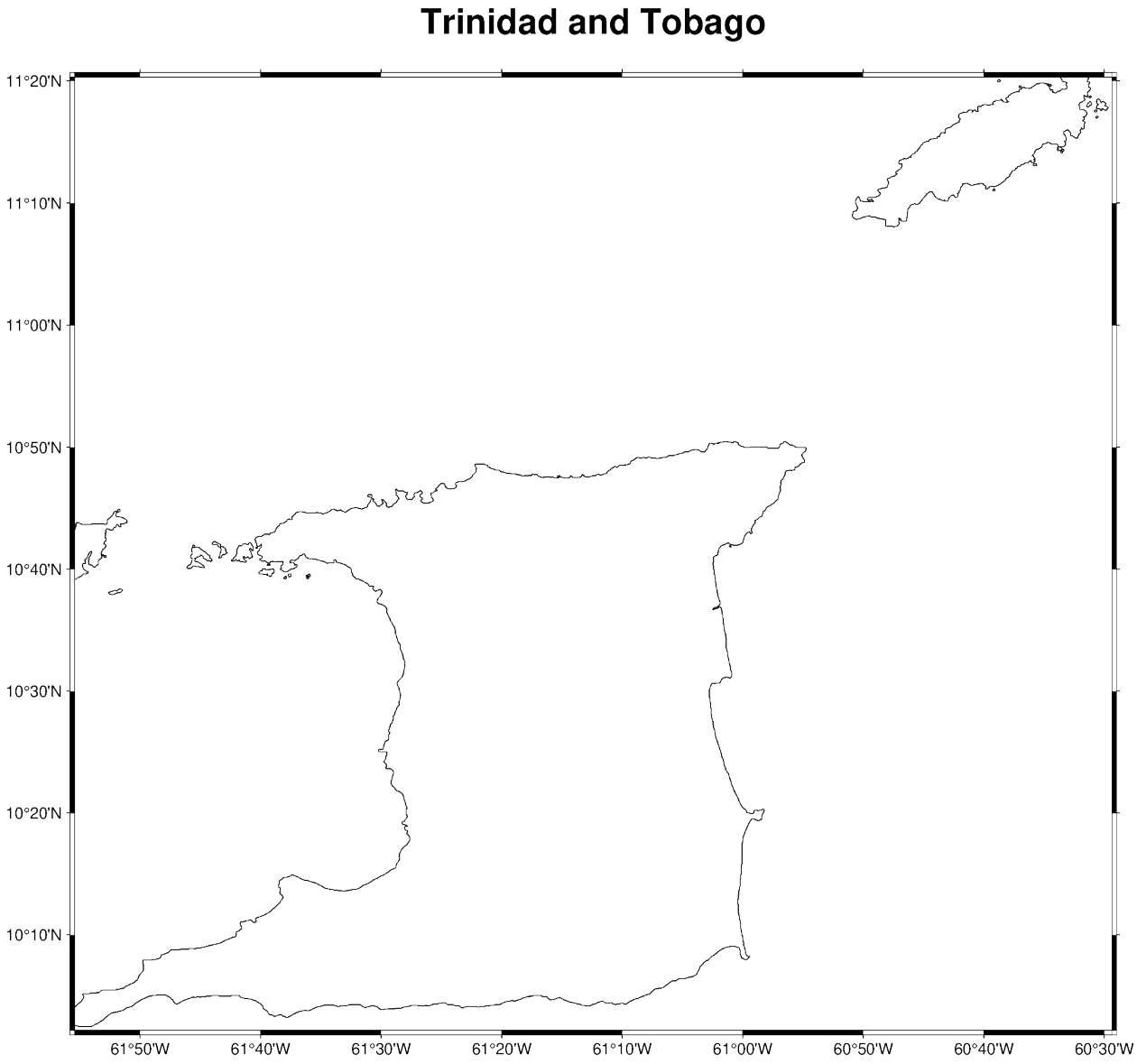
The figure title can be set by passing **+t***title* to the `frame` parameter of *`pygmt.Figure.basemap`*. Passing multiple arguments to `frame` can be done by using a list, as shown in the example below.

```
fig = pygmt.Figure()
# region="TT" specifies Trinidad and Tobago using the ISO country code
```

(continues on next page)

(continued from previous page)

```
fig.coast(shorelines="1/0.5p", region="TT", projection="M25c")
fig.basemap(frame=["a", "+tTrinidad and Tobago"])
fig.show()
```

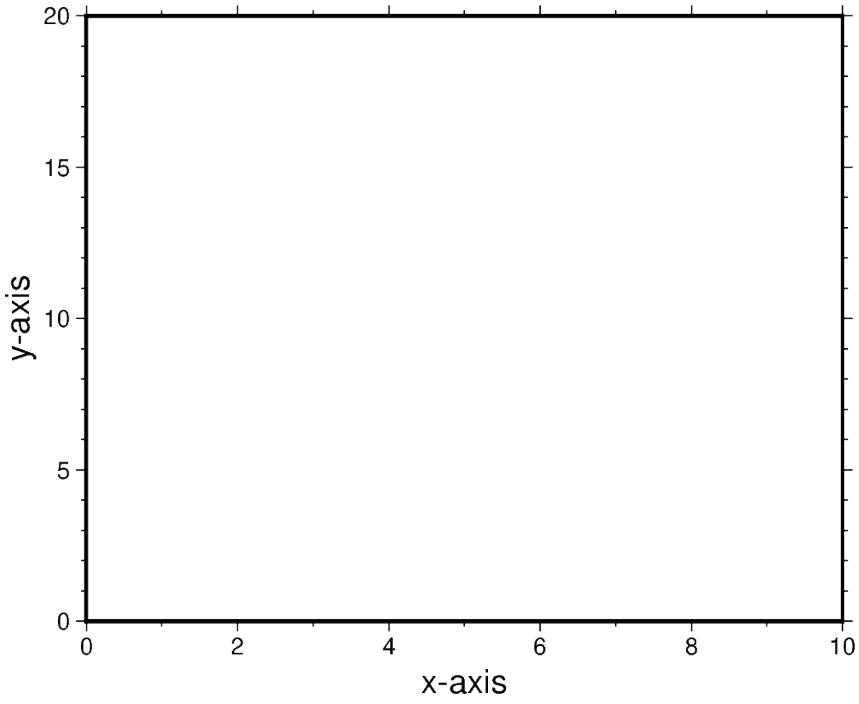


Axis labels

Axis labels, in GMT simply called 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`*. The map boundaries (or plot axes) are named as West/west/left (**W**, **w**, **l**), South/south/bottom (**S**, **s**, **b**), North/north/top (**N**, **n**, **t**), and East/east/right (**E**, **e**, **r**) sides of a figure. If an uppercase letter (**W**, **S**, **N**, **E**) is passed, the axis is plotted with tick marks and annotations. The lowercase version (**w**, **s**, **n**, **e**) plots the axis only with tick marks. To only plot the axis pass **l**, **b**, **t**, **r**. By default (`frame=True` or `frame="af"`), the West and the South axes are plotted with both tick marks and annotations.

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

```
fig = pygmt.Figure()
fig.basemap(
    region=[0, 10, 0, 20],
    projection="X10c/8c",
    # Plot axis with tick marks, annotations, and labels on the
    # West and South axes
    # Plot axis with tick marks on the north and east axes
    frame=["WSne", "xaf+lx-axis", "yaf+ly-axis"],
)
fig.show()
```



Total running time of the script: (0 minutes 1.880 seconds)

4.1.3 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 using *`pygmt.datasets.load_sample_data`*. The data are loaded as a `pandas.DataFrame`.

```
data = pygmt.datasets.load_sample_data(name="japan_quakes")
data.head()
```

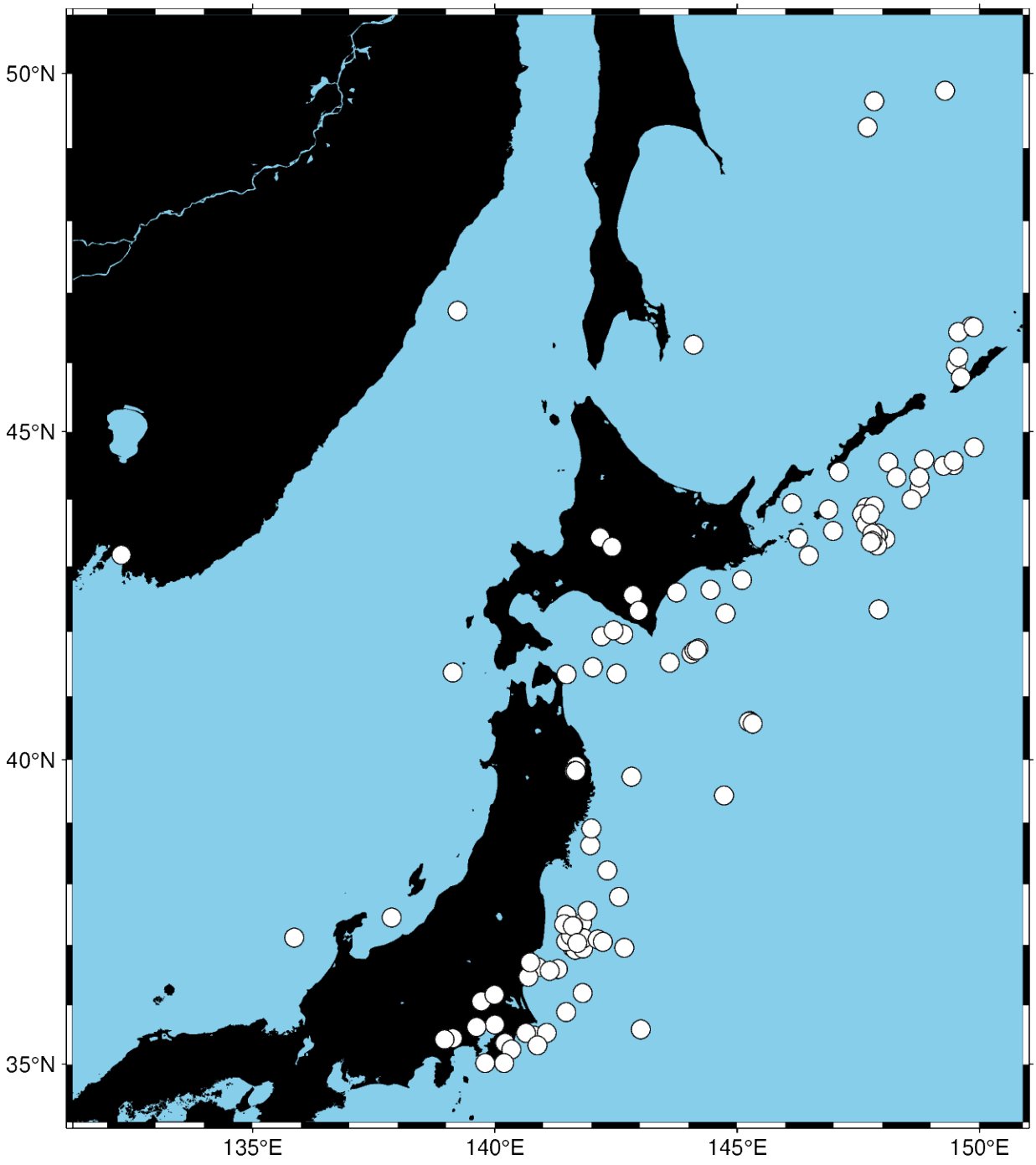
Set the region for the plot to be slightly larger than the data bounds.

```
region = [
    data.longitude.min() - 1,
    data.longitude.max() + 1,
    data.latitude.min() - 1,
    data.latitude.max() + 1,
]
region
```

```
[np.float64(131.29), np.float64(150.89), np.float64(34.02), np.float64(50.77)]
```

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", fill="white", pen="black")
fig.show()
```



We used the style `c0.3c` which means "circles with a diameter of 0.3 centimeters". The `pen` parameter controls the outline of the symbols and the `fill` parameter controls the fill.

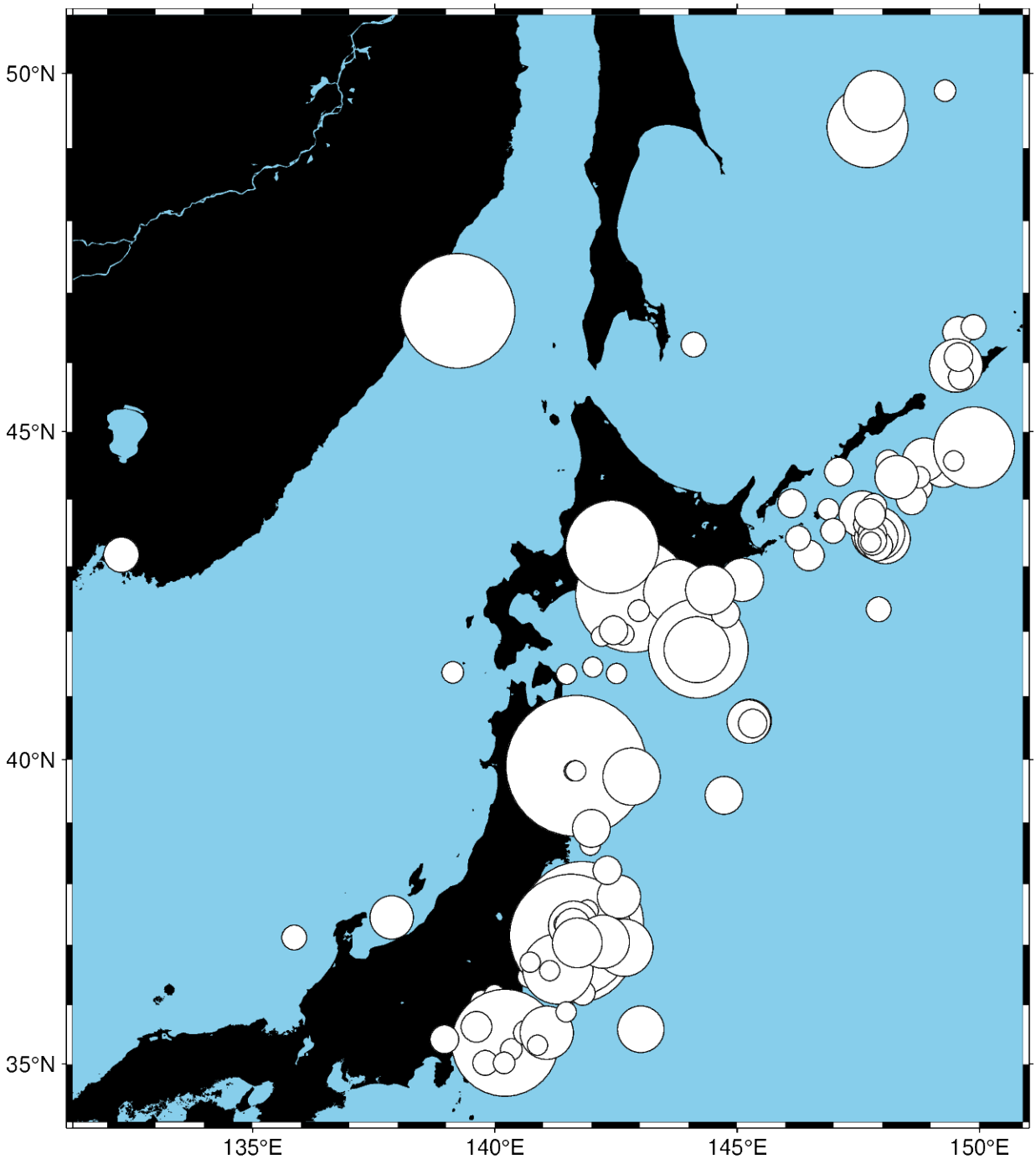
We can map the size of the circles to the earthquake magnitude by passing an array to the `size` parameter. Because the magnitude is on a logarithmic scale, it helps to show the differences by scaling the values using a power law.

```
fig = pygmt.Figure()
fig.basemap(region=region, projection="M15c", frame=True)
fig.coast(land="black", water="skyblue")
fig.plot(
```

(continues on next page)

(continued from previous page)

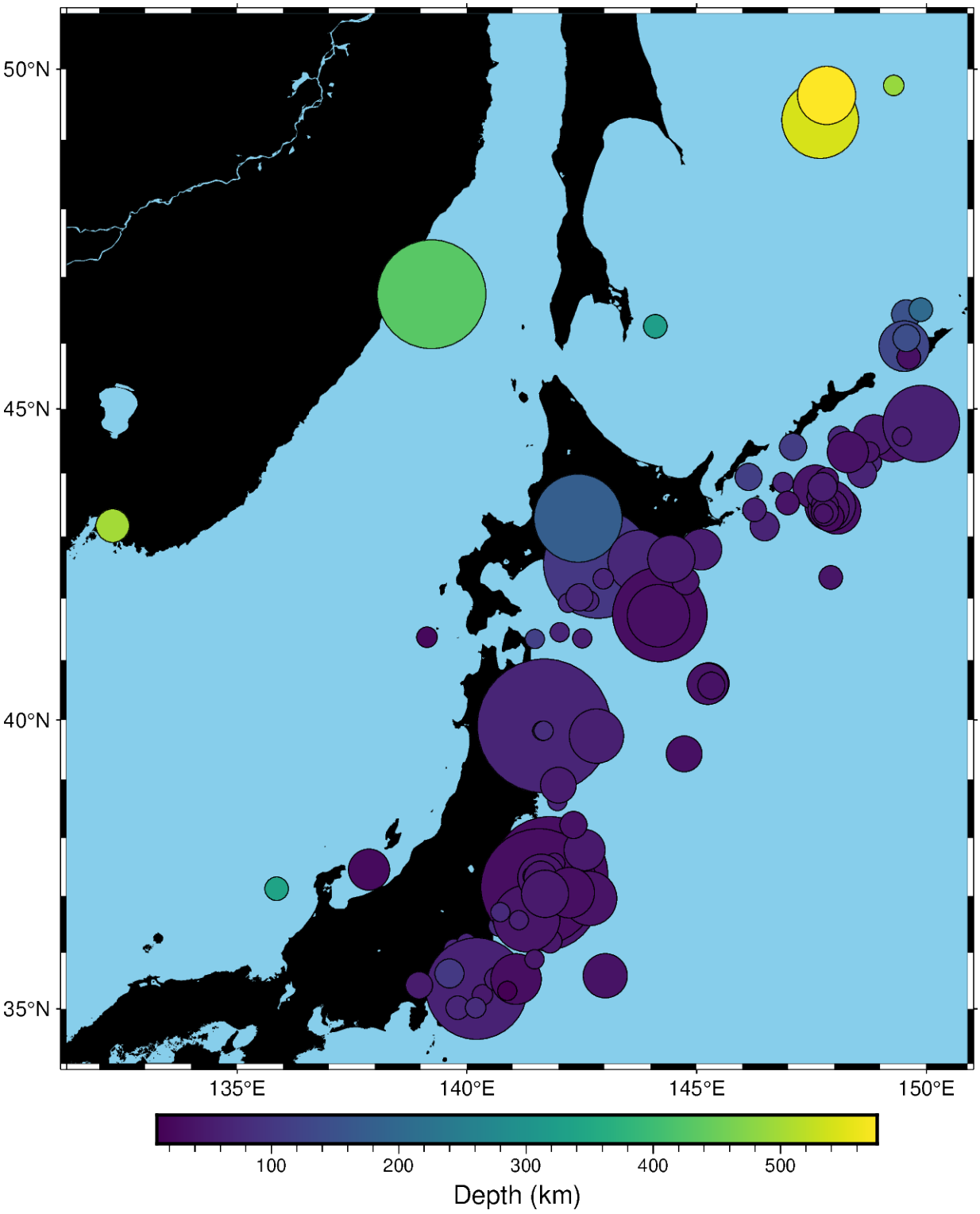
```
    x=data.longitude,
    y=data.latitude,
    size=0.02 * (2**data.magnitude),
    style="cc",
    fill="white",
    pen="black",
)
fig.show()
```



Notice that we didn't include the size in the `style` parameter this time, just the symbol `c` (circles) and the unit `c` (centimeters). 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 `fill` 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,
    fill=data.depth_km,
    cmap=True,
    style="cc",
    pen="black",
)
fig.colorbar(frame="xaf+lDepth (km)")
fig.show()
```



Total running time of the script: (0 minutes 0.745 seconds)

4.1.4 Plotting lines

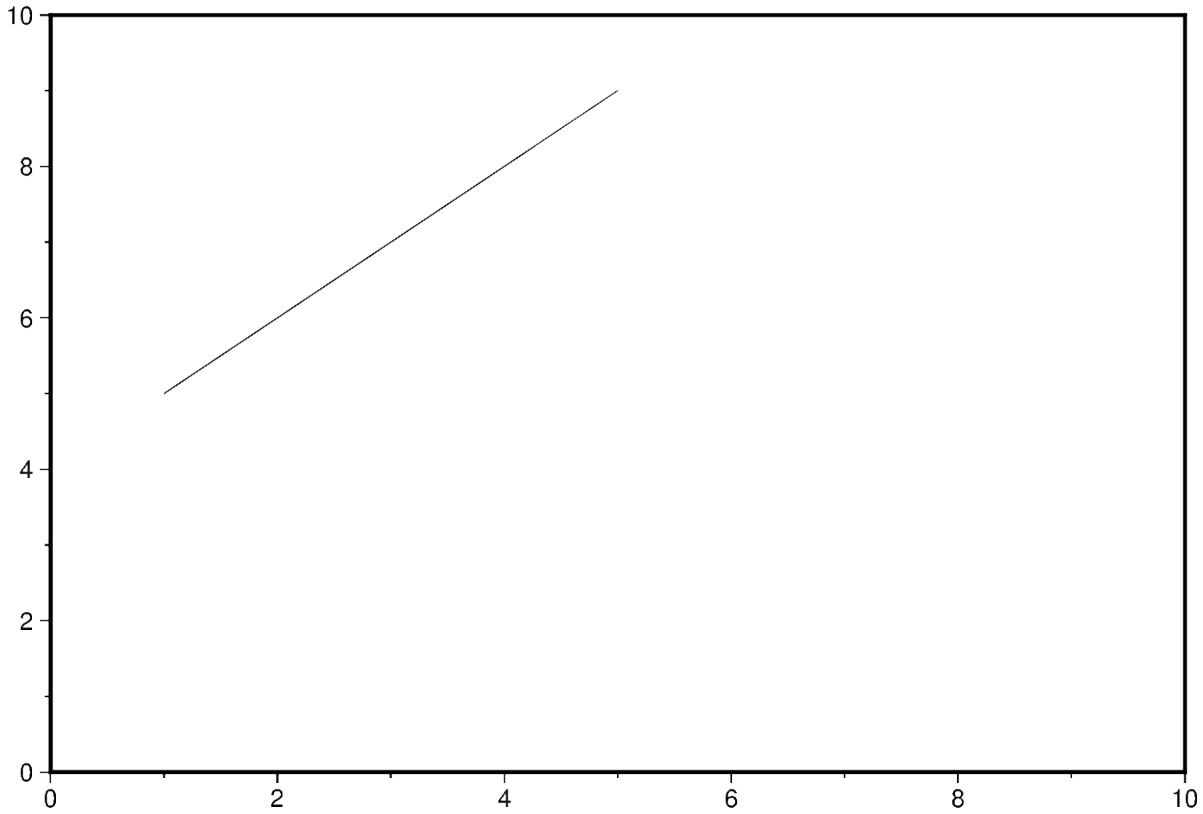
Plotting lines is handled by the *pygmt.Figure.plot* method.

```
import pygmt
```

Plot lines

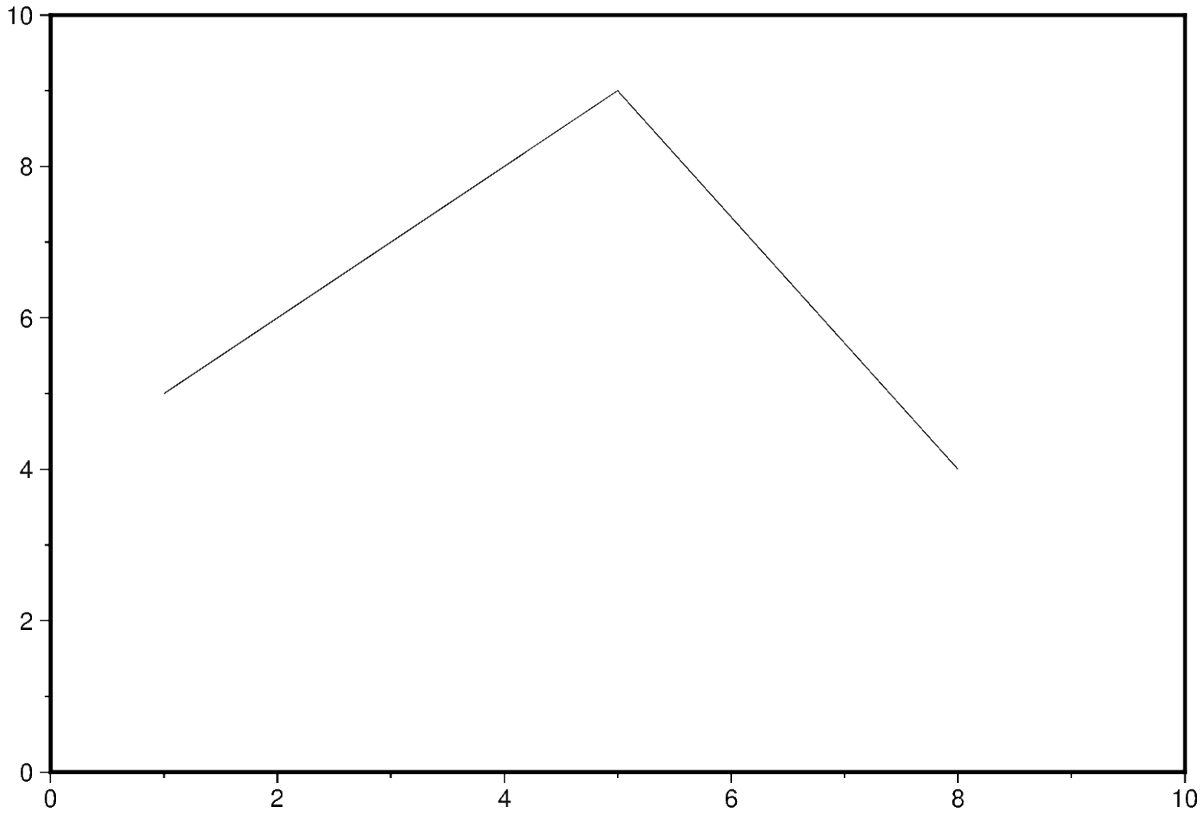
Create a Cartesian figure using the *pygmt.Figure.basemap* method. Pass lists containing two values to the `x` and `y` parameters of the *pygmt.Figure.plot* method. By default, a 0.25-points thick, black, solid line is drawn between these two data points.

```
fig = pygmt.Figure()
fig.basemap(region=[0, 10, 0, 10], projection="X15c/10c", frame=True)

fig.plot(x=[1, 5], y=[5, 9])

fig.show()
```



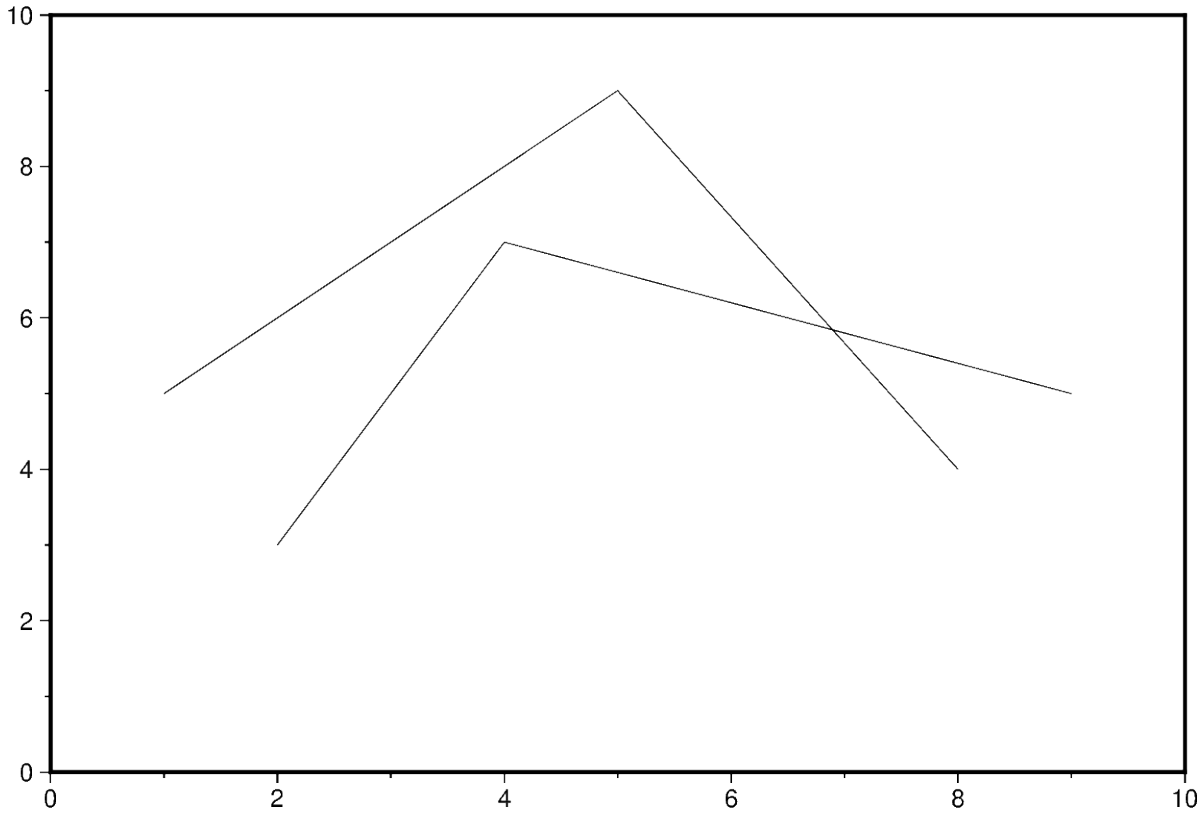
Additional line segments can be added by including more data points in the lists passed to `x` and `y`.

```
fig = pygmt.Figure()
fig.basemap(region=[0, 10, 0, 10], projection="X15c/10c", frame=True)

fig.plot(x=[1, 5, 8], y=[5, 9, 4])

fig.show()
```



To plot multiple lines, *pygmt.Figure.plot* needs to be used for each line separately.

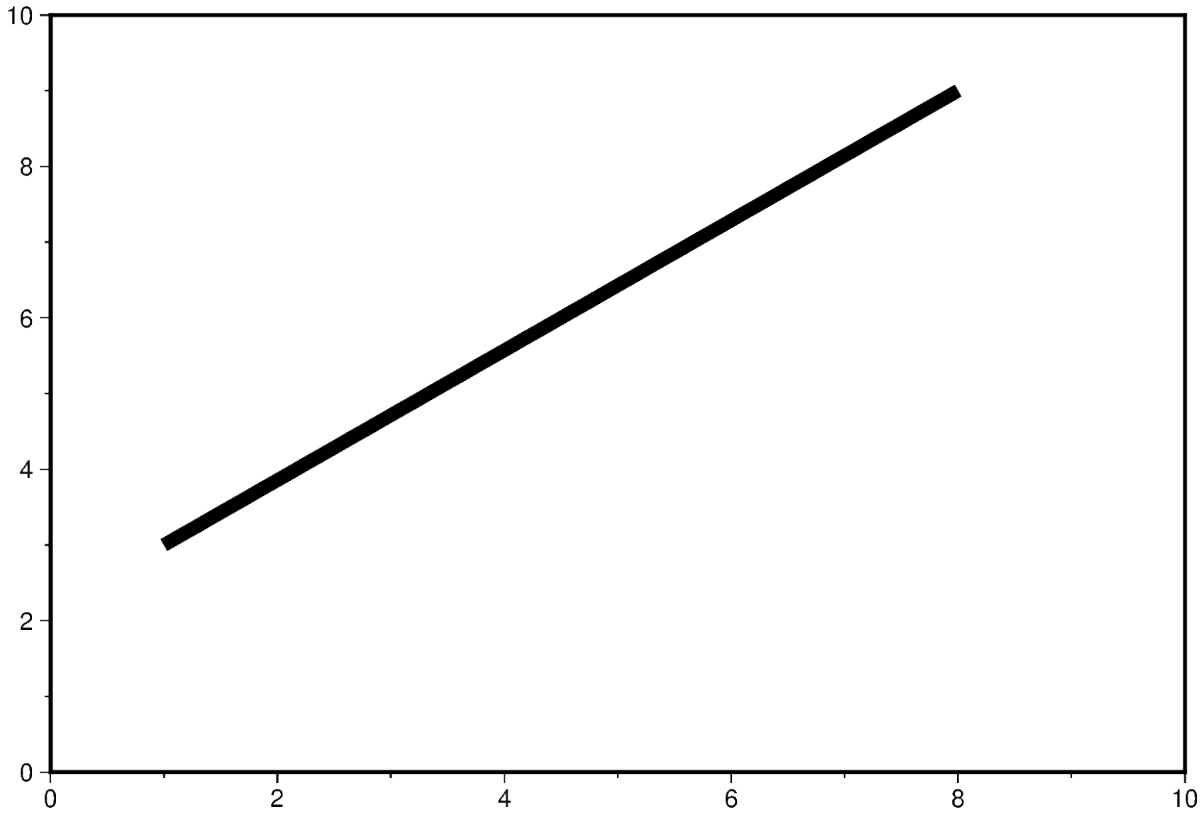
```
fig = pygmt.Figure()
fig.basemap(region=[0, 10, 0, 10], projection="X15c/10c", frame=True)

fig.plot(x=[1, 5, 8], y=[5, 9, 4])
fig.plot(x=[2, 4, 9], y=[3, 7, 5])

fig.show()
```



Change line attributes

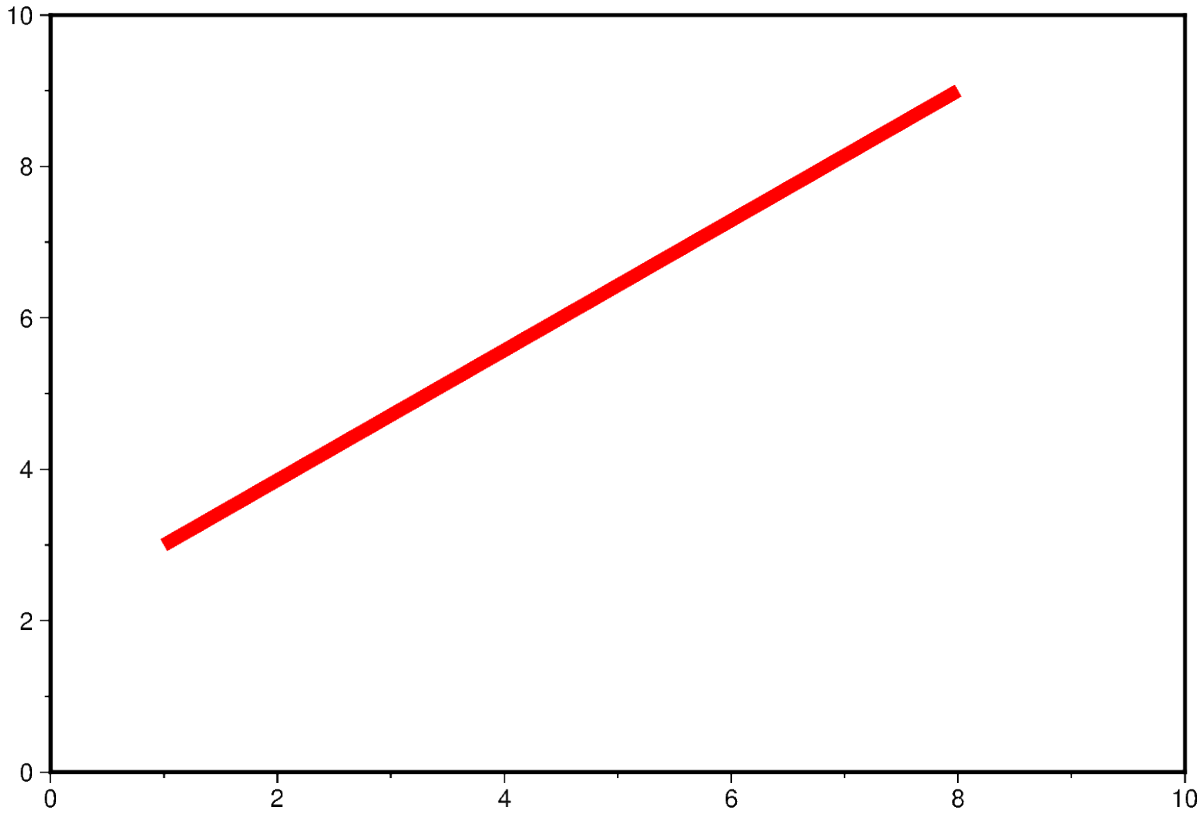
The line attributes can be set by the `pen` parameter which takes a string argument with the optional values *width*,*color*,*style*.

```
fig = pygmt.Figure()
fig.basemap(region=[0, 10, 0, 10], projection="X15c/10c", frame=True)

# Set the pen width to "5p" (5 points), and use the default color "black" and the
# default style "solid"
fig.plot(x=[1, 8], y=[3, 9], pen="5p")

fig.show()
```



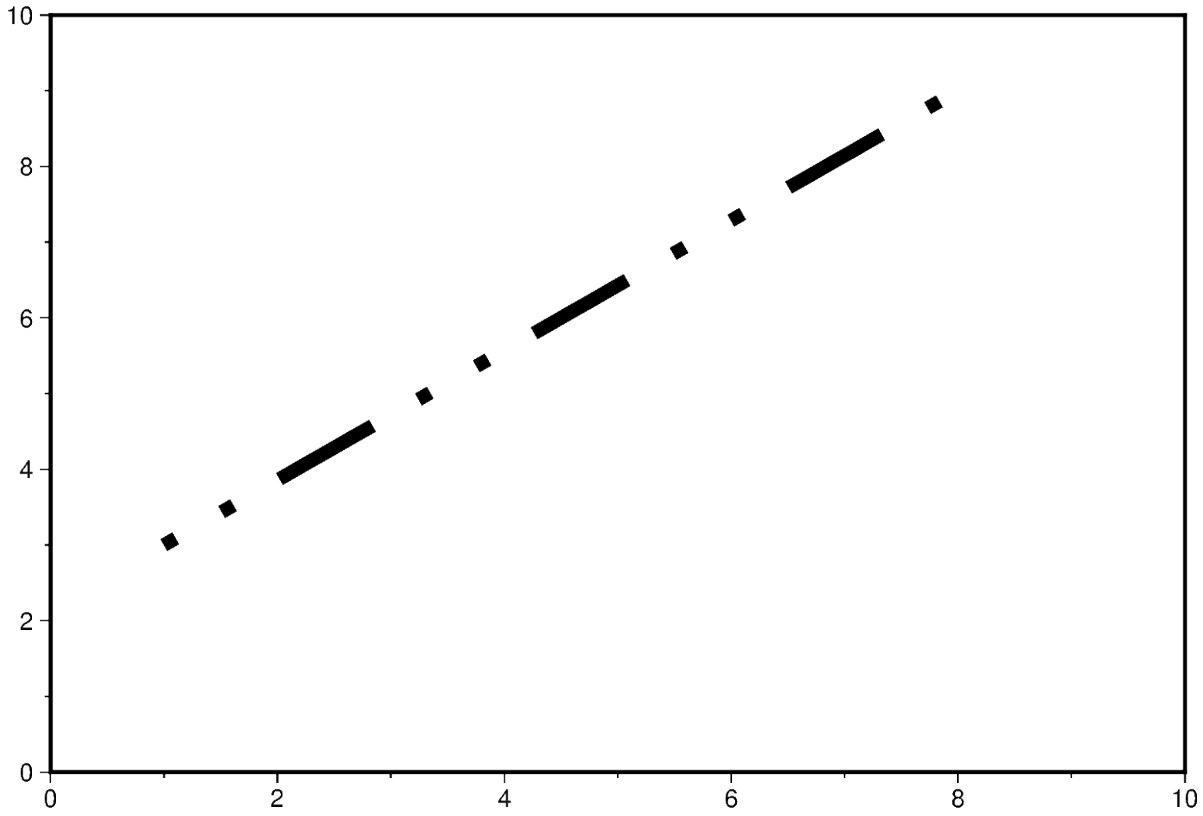
The line color can be set and is added after the line width to the `pen` parameter.

```
fig = pygmt.Figure()
fig.basemap(region=[0, 10, 0, 10], projection="X15c/10c", frame=True)

# Set the line color to "red", use the default style "solid"
fig.plot(x=[1, 8], y=[3, 9], pen="5p,red")

fig.show()
```



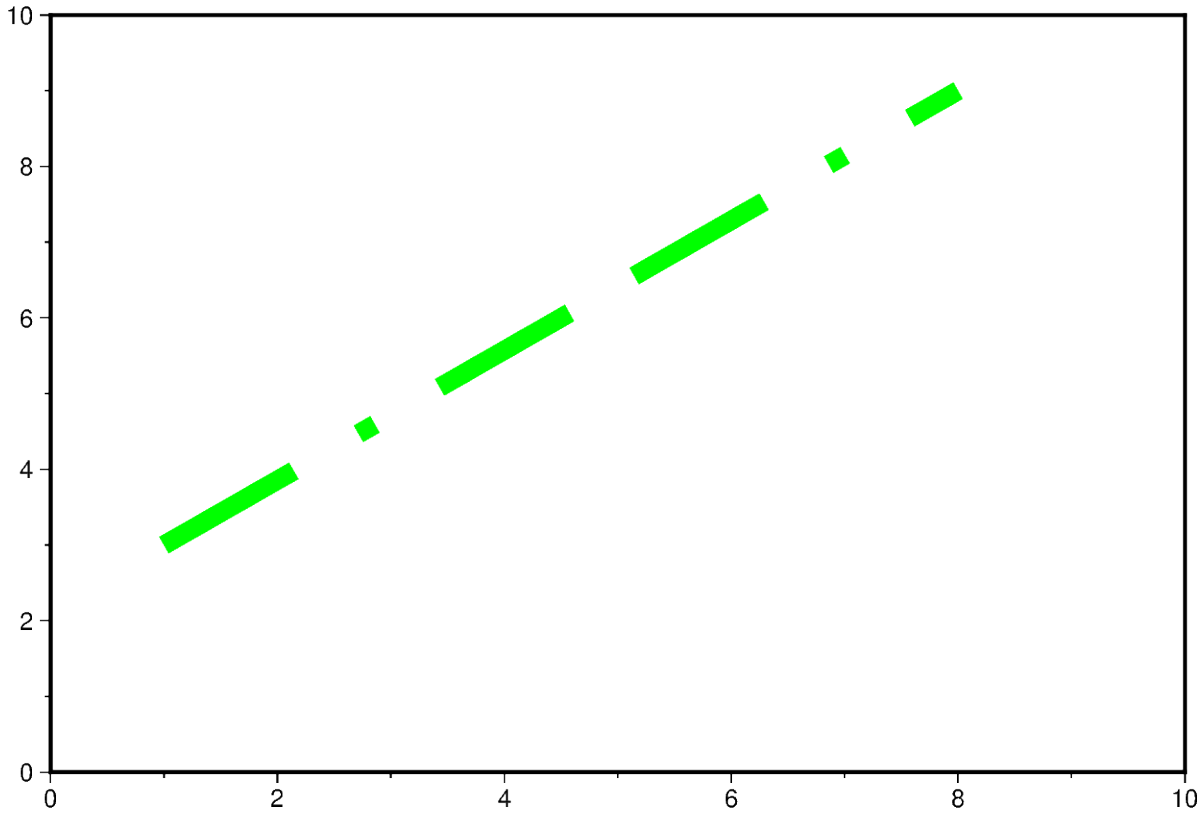
The line style can be set and is added after the line width or color to the `pen` parameter.

```
fig = pygmt.Figure()
fig.basemap(region=[0, 10, 0, 10], projection="X15c/10c", frame=True)

# Set the line style to "..-" (dot dot dash), use the default color "black"
fig.plot(x=[1, 8], y=[3, 9], pen="5p,..-")

fig.show()
```



The line width, color, and style can all be set in the same `pen` parameter. For a gallery example showing other `pen` settings, see *Line styles*.

```
fig = pygmt.Figure()
fig.basemap(region=[0, 10, 0, 10], projection="X15c/10c", frame=True)

# Draw a 7-points thick, green line with style "-.-" (dash dot dash)
fig.plot(x=[1, 8], y=[3, 9], pen="7p,green,-.-")

fig.show()
```



Total running time of the script: (0 minutes 0.942 seconds)

4.1.5 Plotting polygons

Plotting polygons is handled by the *pygmt.Figure.plot* method.

This tutorial focuses on input data given as NumPy arrays. Besides NumPy arrays, array-like objects are supported. Here, a polygon is a closed shape defined by a series of data points with x and y coordinates, connected by line segments, with the start and end points being identical. For plotting a `geopandas.GeoDataFrame` object with polygon geometries, e.g., to create a choropleth map, see the gallery example *Choropleth map*.

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

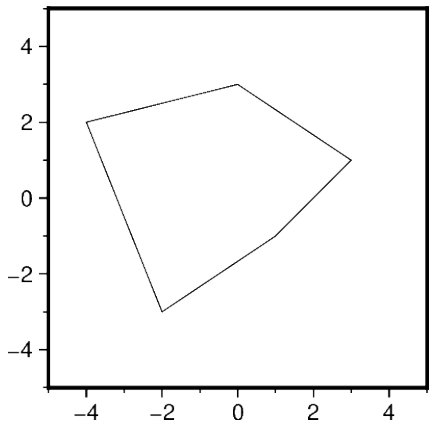
Plot polygons

Set up sample data points as NumPy arrays for the x and y values.

```
x = np.array([-2, 1, 3, 0, -4, -2])
y = np.array([-3, -1, 1, 3, 2, -3])
```

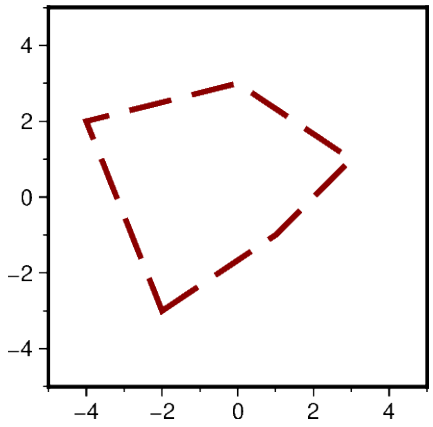
Create a Cartesian plot via the *pygmt.Figure.basemap* method. Pass arrays to the `x` and `y` parameters of the *pygmt.Figure.plot* method. Without further adjustments, lines are drawn between the data points. By default, the lines are 0.25-points thick, black, and solid. In this example, the data points are chosen to make the lines form a polygon.

```
fig = pygmt.Figure()
fig.basemap(region=[-5, 5, -5, 5], projection="X5c", frame=True)
fig.plot(x=x, y=y)
fig.show()
```



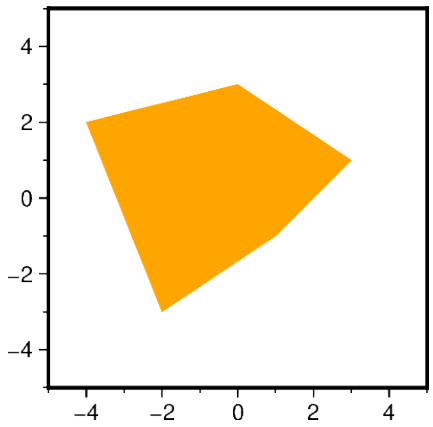
The `pen` parameter can be used to adjust the lines or outline of the polygon. The argument passed to `pen` is one string with the comma-separated optional values *width*,*color*,*style*.

```
fig = pygmt.Figure()
fig.basemap(region=[-5, 5, -5, 5], projection="X5c", frame=True)
# Use a 2-points thick, darkred, dashed outline
fig.plot(x=x, y=y, pen="2p,darkred,dashed")
fig.show()
```



Use the `fill` parameter to fill the polygon with a color or *pattern*. Note, that there are no lines drawn between the data points by default if `fill` is used. Use the `pen` parameter to add an outline around the polygon.

```
fig = pygmt.Figure()
fig.basemap(region=[-5, 5, -5, 5], projection="X5c", frame=True)
# Fill the polygon with color "orange"
fig.plot(x=x, y=y, fill="orange")
fig.show()
```



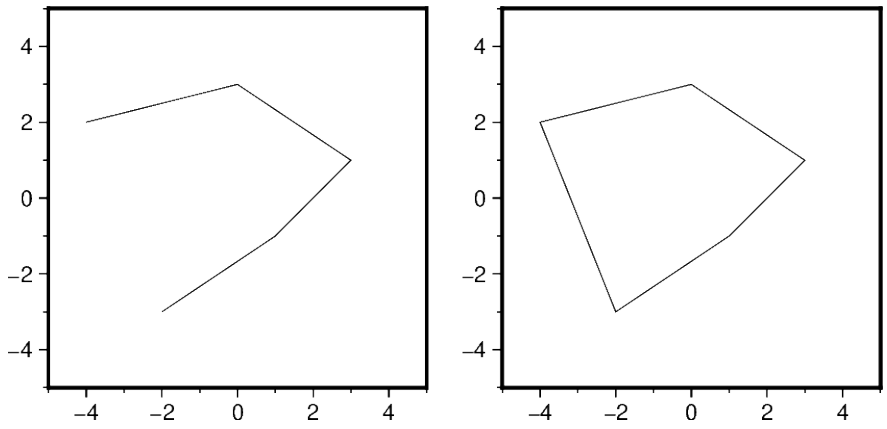
Close polygons

Set up sample data points as NumPy arrays for the x and y values. Now, the data points do not form a polygon.

```
x = np.array([-2, 1, 3, 0, -4])
y = np.array([-3, -1, 1, 3, 2])
```

The `close` parameter can be used to force the polygon to be closed.

```
fig = pygmt.Figure()
fig.basemap(region=[-5, 5, -5, 5], projection="X5c", frame=True)
fig.plot(x=x, y=y, pen=True)

fig.shift_origin(xshift="w+1c")

fig.basemap(region=[-5, 5, -5, 5], projection="X5c", frame=True)
fig.plot(x=x, y=y, pen=True, close=True)
fig.show()
```



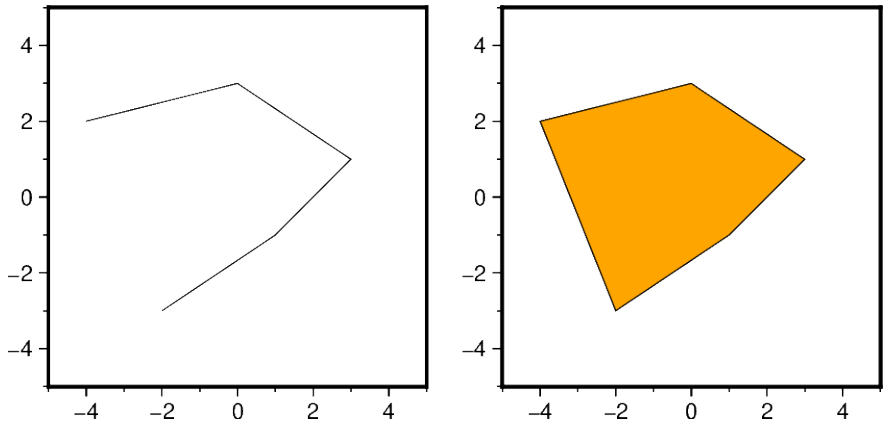
When using the `fill` parameter, the polygon is automatically closed.

```
fig = pygmt.Figure()
fig.basemap(region=[-5, 5, -5, 5], projection="X5c", frame=True)
fig.plot(x=x, y=y, pen=True)

fig.shift_origin(xshift="w+1c")
```

(continues on next page)

(continued from previous page)

```
fig.basemap(region=[-5, 5, -5, 5], projection="X5c", frame=True)
fig.plot(x=x, y=y, pen=True, fill="orange")
fig.show()
```



Total running time of the script: (0 minutes 0.519 seconds)

4.1.6 Plotting text

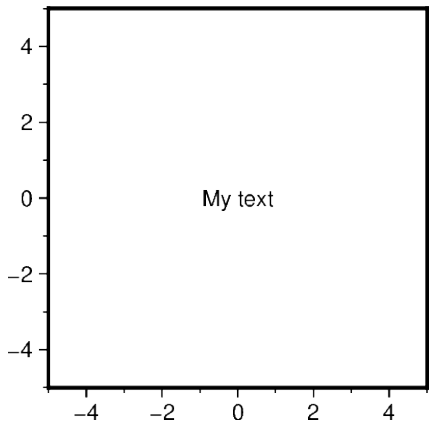
It is often useful to add text annotations to a plot or map. This is handled by the *pygmt.Figure.text* method of the *pygmt.Figure* class.

```
from pathlib import Path

import pygmt
```

Adding a single text label

To add a single text label to a plot, use the `text` and `x` and `y` parameters to specify the text and position.

```
fig = pygmt.Figure()
fig.basemap(region=[-5, 5, -5, 5], projection="X5c", frame=True)
fig.text(x=0, y=0, text="My text")
fig.show()
```



Adjusting the text label

There are several optional parameters to adjust the text label:

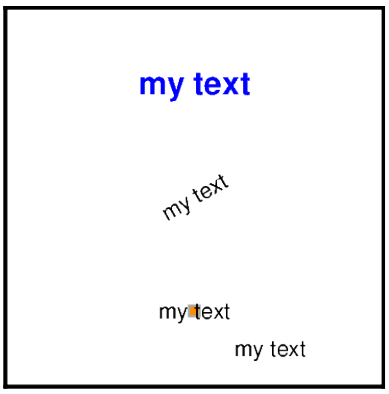
- `font`: Sets the size, family/weight, and color of the font for the text. A list of all recognized fonts can be found at *Supported Fonts*. For details of how to use non-default fonts, refer to PostScript Fonts Used by GMT.
- `angle`: Specifies the rotation of the text. It is measured counter-clockwise from the horizontal in degrees.
- `justify`: Defines the anchor point of the bounding box for the text. It is specified by a two-letter (order independent) code, chosen from:
  - Vertical: **T**(op), **M**(iddle), **B**(ottom)
  - Horizontal: **L**(eft), **C**(entre), **R**(ight)
- `offset`: Shifts the text relatively to the reference point.

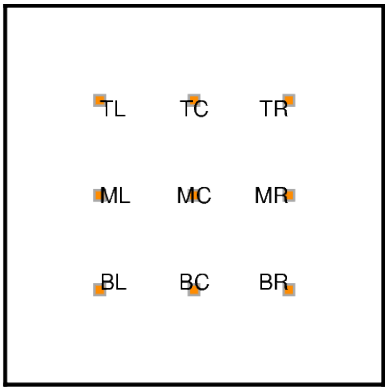
```
fig = pygmt.Figure()

# --------------------------------------------------------------------------
# Left: "font", "angle", and "offset" parameters
fig.basemap(region=[-5, 5, -5, 5], projection="X5c", frame="rtlb")

# Change font size, family/weight, color of the text
fig.text(x=0, y=3, text="my text", font="12p,Helvetica-Bold,blue")

# Rotate the text by 30 degrees counter-clockwise from the horizontal
fig.text(x=0, y=0, text="my text", angle=30)

# Plot marker and text label for reference
fig.plot(x=0, y=-3, style="s0.2c", fill="darkorange", pen="0.7p,darkgray")
fig.text(x=0, y=-3, text="my text")
# Shift the text label relatively to the position given via the x and y parameters
# by 1 centimeter to the right (positive x direction) and 0.5 centimeters down
# (negative y direction)
fig.text(x=0, y=-3, text="my text", offset="1c/-0.5c")

fig.shift_origin(xshift="w+0.5c")

# --------------------------------------------------------------------------
# Right: "justify" parameter
fig.basemap(region=[-1, 1, -1, 1], projection="X5c", frame="rtlb")

# Plot markers for reference
fig.plot(
    x=[-0.5, 0, 0.5, -0.5, 0, 0.5, -0.5, 0, 0.5],
    y=[0.5, 0.5, 0.5, 0, 0, 0, -0.5, -0.5, -0.5],
    style="s0.2c",
    fill="darkorange",
    pen="0.7p,darkgray",
)

# Plot text labels at the x and y positions of the markers while varying the anchor
# point via the justify parameter
fig.text(x=-0.5, y=0.5, text="TL", justify="TL")  # TopLeft
fig.text(x=0, y=0.5, text="TC", justify="TC")  # TopCenter
```

(continues on next page)

(continued from previous page)

```
fig.text(x=0.5, y=0.5, text="TR", justify="TR")  # TopRight
fig.text(x=-0.5, y=0, text="ML", justify="ML")  # MiddleLeft
fig.text(x=0, y=0, text="MC", justify="MC")  # MiddleCenter
fig.text(x=0.5, y=0, text="MR", justify="MR")  # MiddleRight
fig.text(x=-0.5, y=-0.5, text="BL", justify="BL")  # BottomLeft
fig.text(x=0, y=-0.5, text="BC", justify="BC")  # BottomCenter
fig.text(x=0.5, y=-0.5, text="BR", justify="BR")  # BottomRight

fig.show()
```



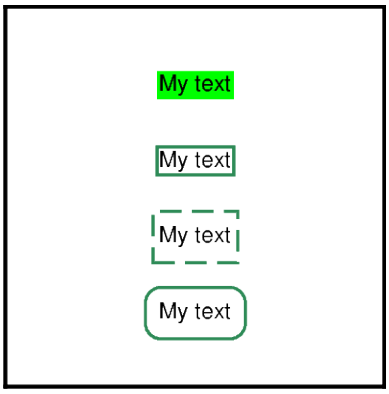


Adding a text box

There are different optional parameters to add and customize a text box:

- `fill`: Fills the text box with a color.
- `pen`: Outlines the text box.
- `clearance`: Adds margins in x and y directions between the text and the outline of the text box. Can be used to get a text box with rounded edges.

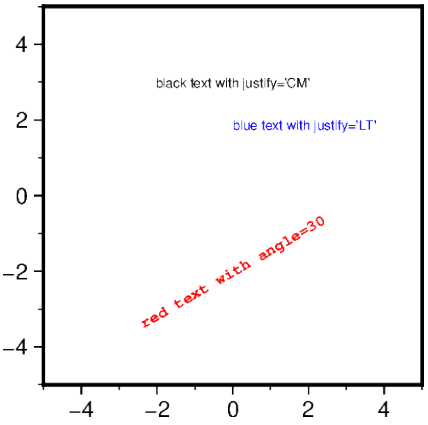
```
fig = pygmt.Figure()

fig.basemap(region=[-5, 5, -5, 5], projection="X5c", frame="rtlb")

# Add a box with a fill in green color
fig.text(x=0, y=3, text="My text", fill="green")

# Add box with a seagreen, 1-point thick, solid outline
fig.text(x=0, y=1, text="My text", pen="1p,seagreen,solid")

# Add margins between the text and the outline of the text box of 0.1
# centimeters in x direction and 0.2 centimeters in y direction
fig.text(x=0, y=-1, text="My text", pen="1p,seagreen,dashed", clearance="0.1c/0.2c")

# Get rounded edges by passing "+tO" to the "clearance" parameter
fig.text(x=0, y=-3, text="My text", pen="1p,seagreen,solid", clearance="0.2c/0.2c+tO")

fig.show()
```



Adding multiple text labels with individual configurations

To add multiple text labels with individual `font`, `angle`, and `justify`, one can provide lists with the corresponding arguments.

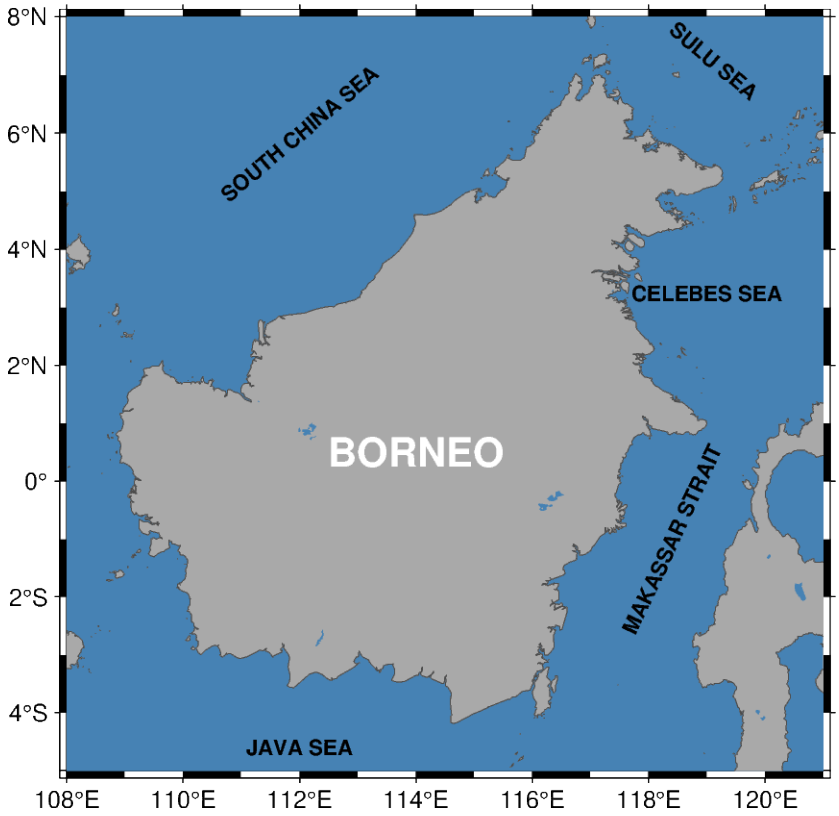
```
fig = pygmt.Figure()
fig.basemap(region=[-5, 5, -5, 5], projection="X5c", frame=True)

fig.text(
    x=[0, 0, 0],
    y=[3, 2, -2],
    font=["5p,Helvetica,black", "5p,Helvetica,blue", "6p,Courier-Bold,red"],
    angle=[0, 0, 30],
    justify=["CM", "LT", "CM"],
    text=[
        "black text with justify='CM'",
        "blue text with justify='LT'",
        "red text with angle=30",
    ],
)

fig.show()
```



Using an external input file

It is also possible to add text labels via an external input file containing `x`, `y`, and `text` columns. Addionaly, columns to set the `angle`, `front`, and `justify` parameters can be provided. Here, we give a complete example.

```
fig = pygmt.Figure()
fig.basemap(region=[108, 121, -5, 8], projection="M10c", frame="a2f1")
fig.coast(land="darkgray", water="steelblue", shorelines="1/0.1p,gray30")

# Create space-delimited file with region / sea names:
# - longitude (x) and latitude (y) coordinates are in the first two columns
# - angle, font, and justify muss be present in this order in the next three columns
# - the text to be printed is given in the last column
with Path.open("examples.txt", "w") as f:
    f.write("114.00  0.50   0 15p,Helvetica-Bold,white CM BORNEO\n")
    f.write("119.00  3.25   0  8p,Helvetica-Bold,black CM CELEBES SEA\n")
    f.write("112.00 -4.60   0  8p,Helvetica-Bold,black CM JAVA SEA\n")
    f.write("112.00  6.00  40  8p,Helvetica-Bold,black CM SOUTH CHINA SEA\n")
    f.write("119.12  7.25 -40  8p,Helvetica-Bold,black CM SULU SEA\n")
    f.write("118.40 -1.00  65  8p,Helvetica-Bold,black CM MAKASSAR STRAIT\n")

# Setting the angle, font, and justify parameters to True indicates that those columns
# are present in the text file
fig.text(textfiles="examples.txt", angle=True, font=True, justify=True)

# Cleanups
Path("examples.txt").unlink()

fig.show()
```



Using the position parameter

Instead of using the `x` and `y` parameters, the `position` parameter can be specified to set the reference point for the text on the plot. As for the `justify` parameter, the `position` parameter is specified by a two-letter (order independent) code, chosen from:

- Vertical: **T**(op), **M**(iddle), **B**(ottom)
- Horizontal: **L**(eft), **C**(entre), **R**(ight)

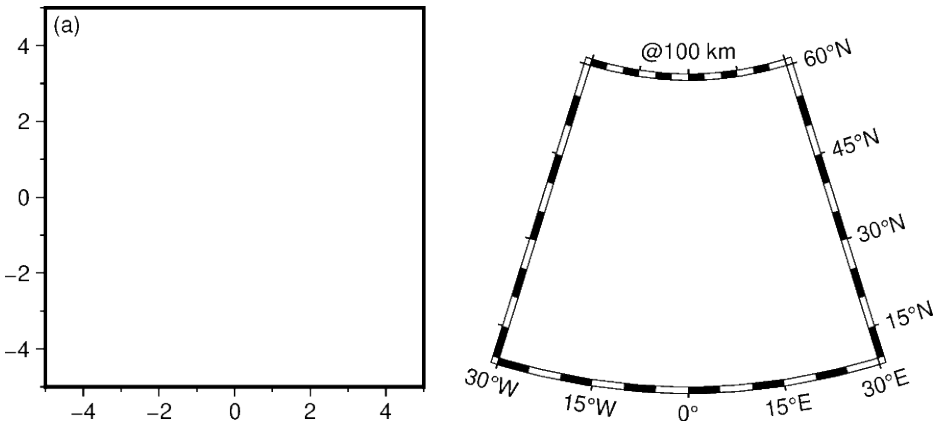
This can be helpful to add a tag to a subplot or text labels out of the plot or map frame, e.g., for depth slices.

```
fig = pygmt.Figure()

# --------------------------------------------------------------------------------
# Left: Add a tag to a subplot
fig.basemap(region=[-5, 5, -5, 5], projection="X5c", frame=["WStr", "af"])

fig.text(
    text="(a)",
    position="TL",  # Top Left
    justify="TL",  # Top Left
    offset="0.1c/-0.1c",
)

fig.shift_origin(xshift="w+1c")

# --------------------------------------------------------------------------------
# Right: Add a text label outside of the plot or map frame
```

(continues on next page)

(continued from previous page)

```
fig.basemap(region=[-30, 30, 10, 60], projection="L0/35/23/47/5c", frame=["wSnE", "af
↪"])

fig.text(
    text="@@100 km",  # "@@" gives "@" in GMT or PyGMT
    position="TC",  # Top Center
    justify="MC",  # Middle Center
    offset="0c/0.2c",
    no_clip=True,  # Allow plotting outside of the map or plot frame
)

fig.show()
```



Advanced configuration

For crafting more advanced styles, including using special symbols and other character sets, be sure to check out the GMT documentation at https://docs.generic-mapping-tools.org/6.5/text.html and also the Technical References at https://docs.generic-mapping-tools.org/6.5/reference/features.html#placement-of-text. Good luck!

Total running time of the script: (0 minutes 0.800 seconds)

4.1.7 Setting the region

Many of the plotting methods 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
```

Coordinates

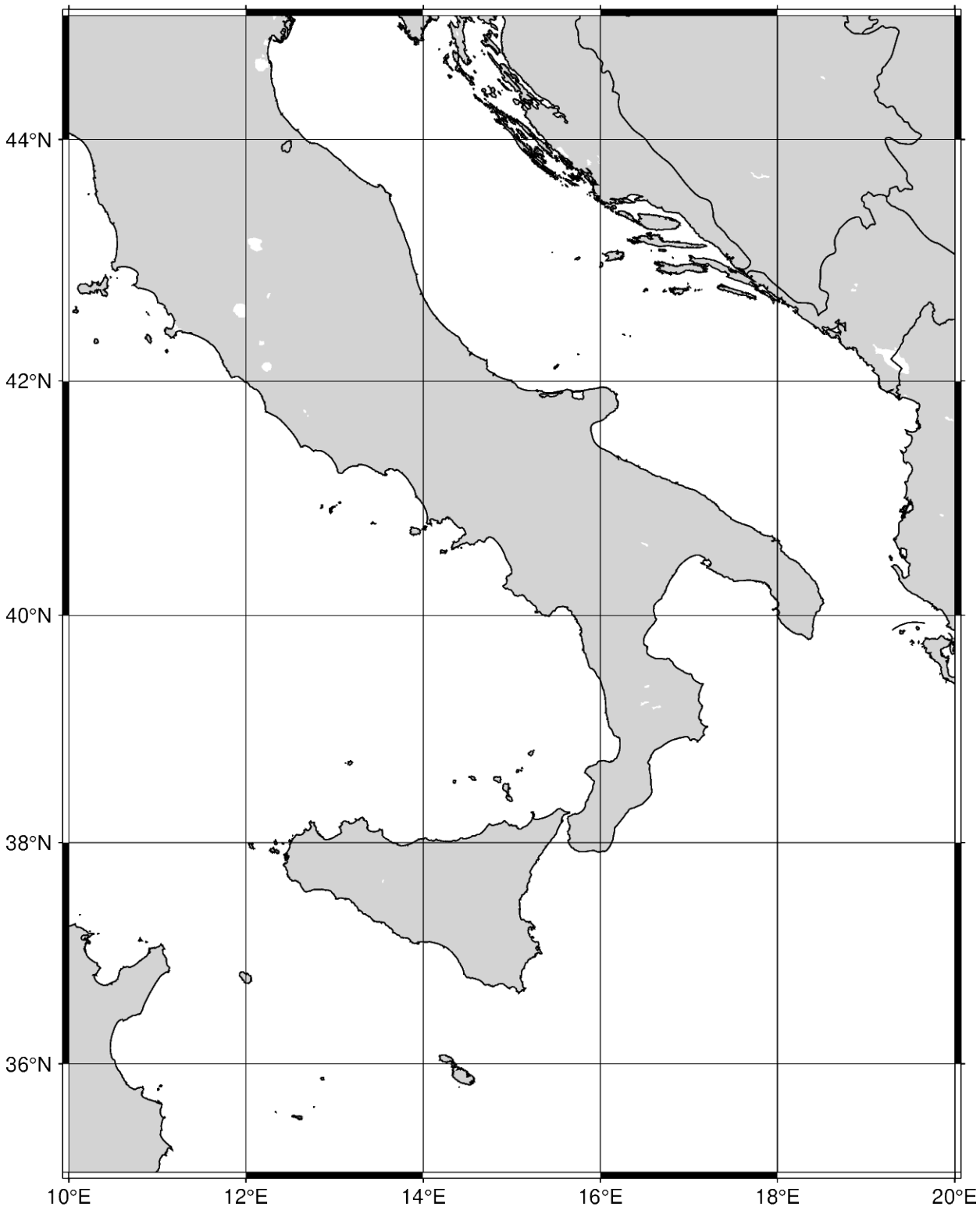
A string of coordinates can be passed to `region`, in the form of *xmin*/*xmax*/*ymin*/*ymax*.

```
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
```

(continues on next page)

(continued from previous page)

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



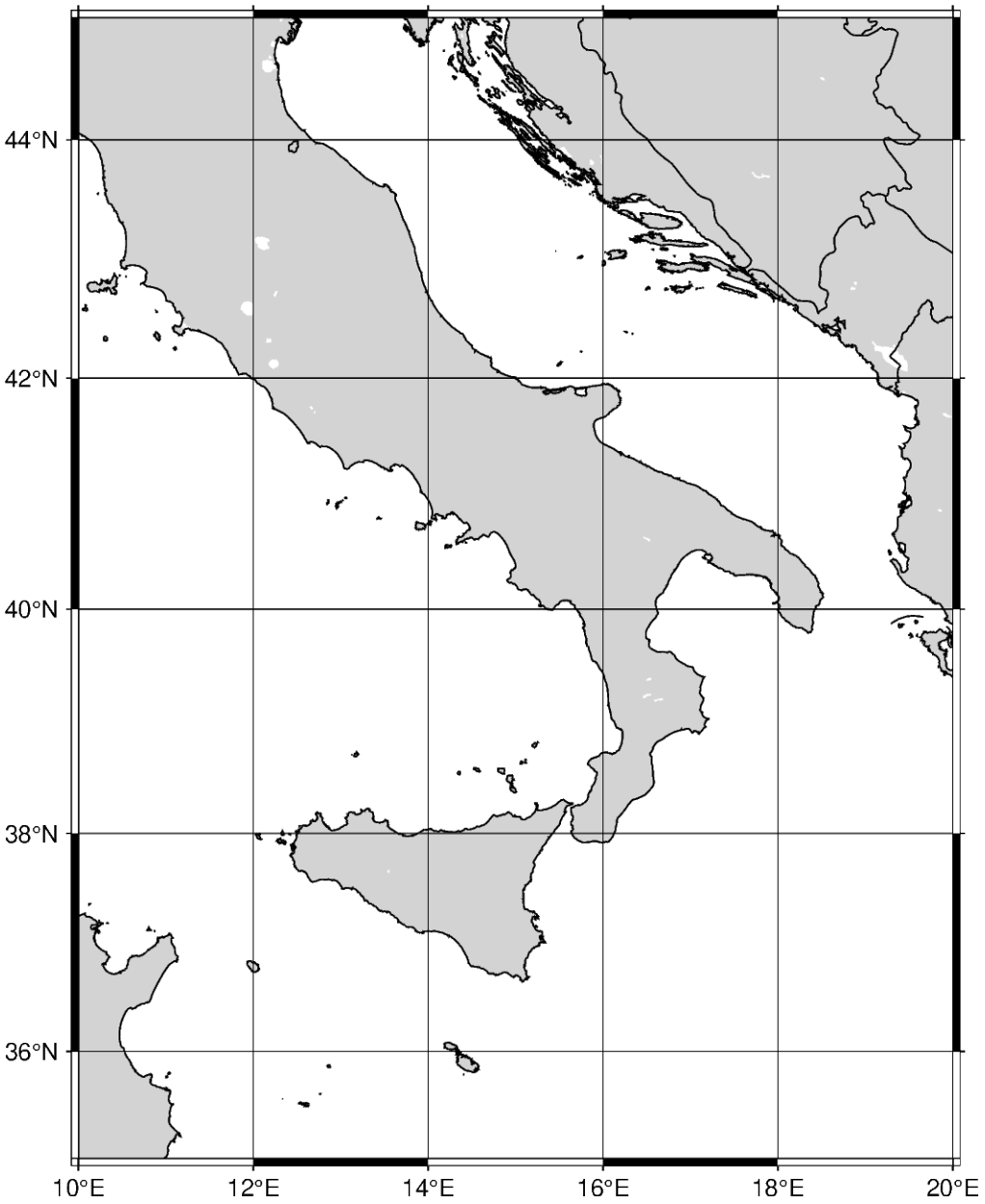
The coordinates can be passed to `region` as a list, in the form of [*xmin*, *xmax*, *ymin*, *ymax*].

```
fig = pygmt.Figure()
fig.coast(
    # Set the x-range from 10E to 20E and the y-range to 35N to 45N
```

(continues on next page)

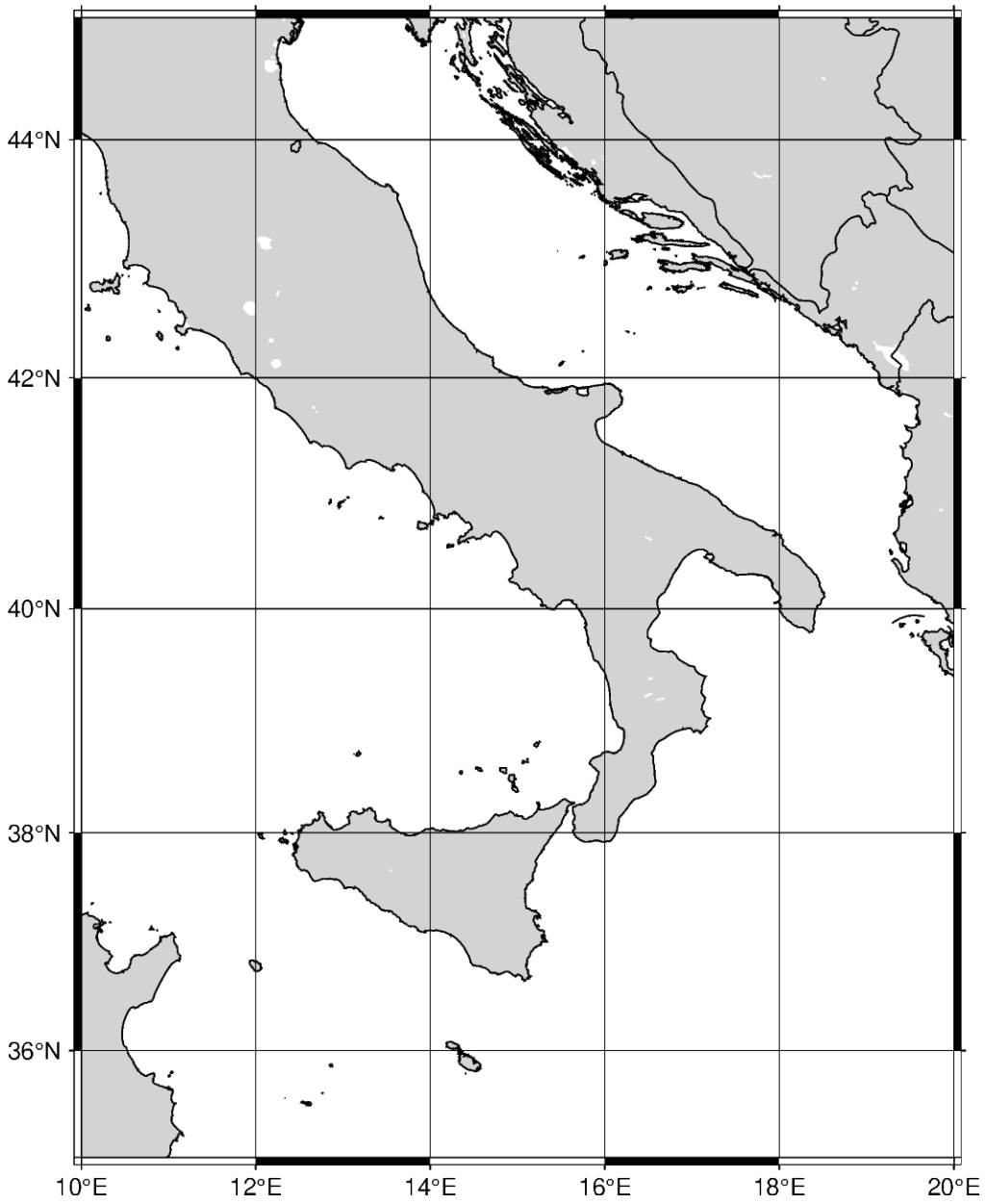
(continued from previous page)

```
    region=[10, 20, 35, 45],
    projection="M12c",
    land="lightgray",
    water="white",
    borders="1/0.5p",
    shorelines="1/0.5p",
    frame="ag",
)
fig.show()
```



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.

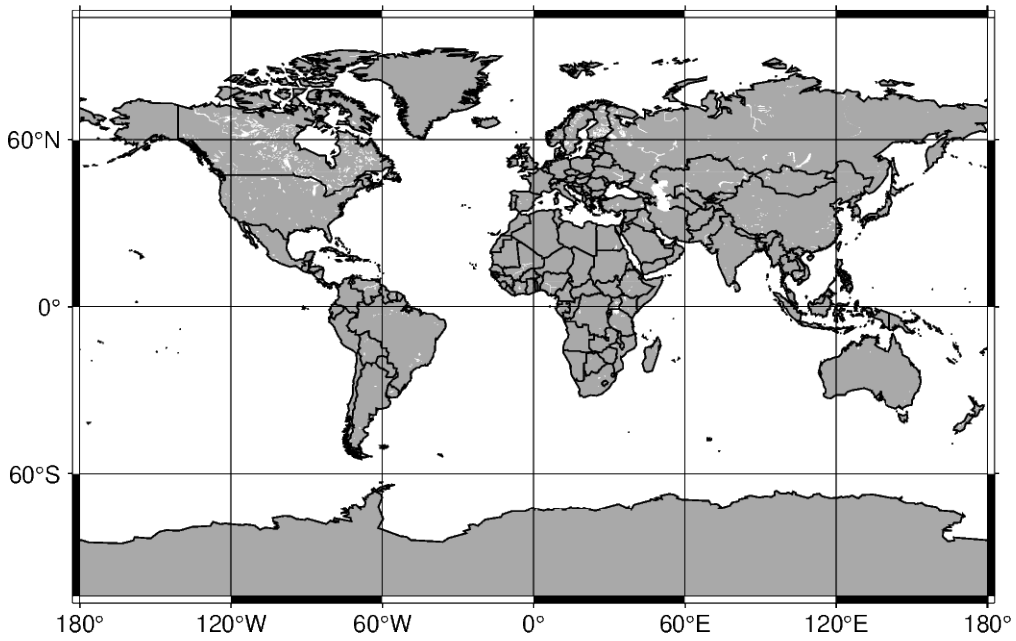
```
fig = pygmt.Figure()
fig.coast(
    # Set the bottom-left corner as 10E, 35N and the top-right corner as
    # 20E, 45N
    region="10/35/20/45+r",
    projection="M12c",
    land="lightgray",
    water="white",
    borders="1/0.5p",
    shorelines="1/0.5p",
    frame="ag",
)
fig.show()
```



Global regions

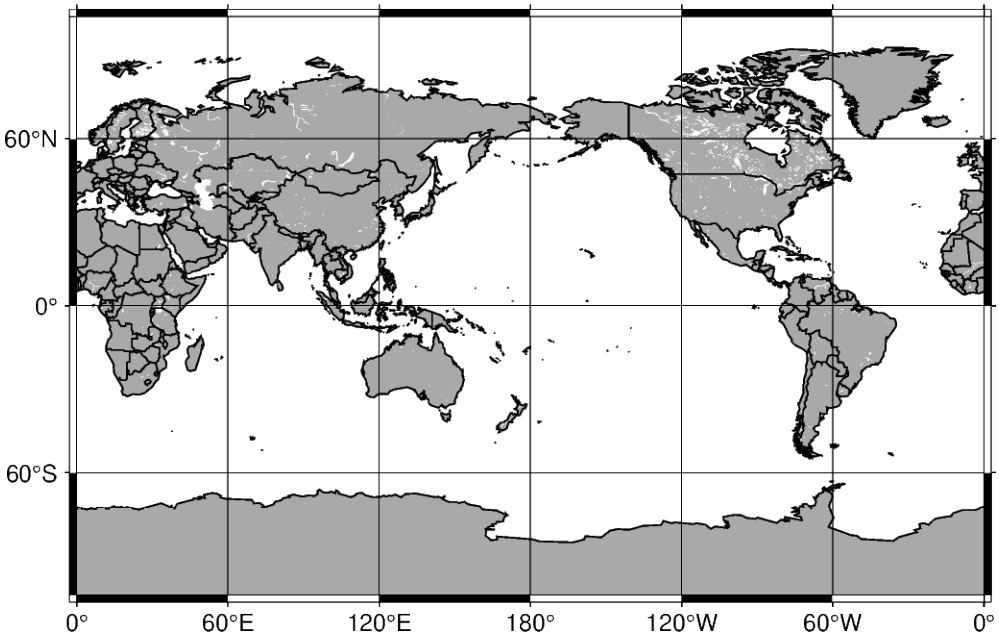
In addition to passing coordinates, the argument **d** can be passed to set the region to the entire globe. The range is 180W to 180E (-180, 180) and 90S to 90N (-90 to 90). With no parameters set for the projection, the figure defaults to be centered at the mid-point of both x- and y-axes. Using **d**,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



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.

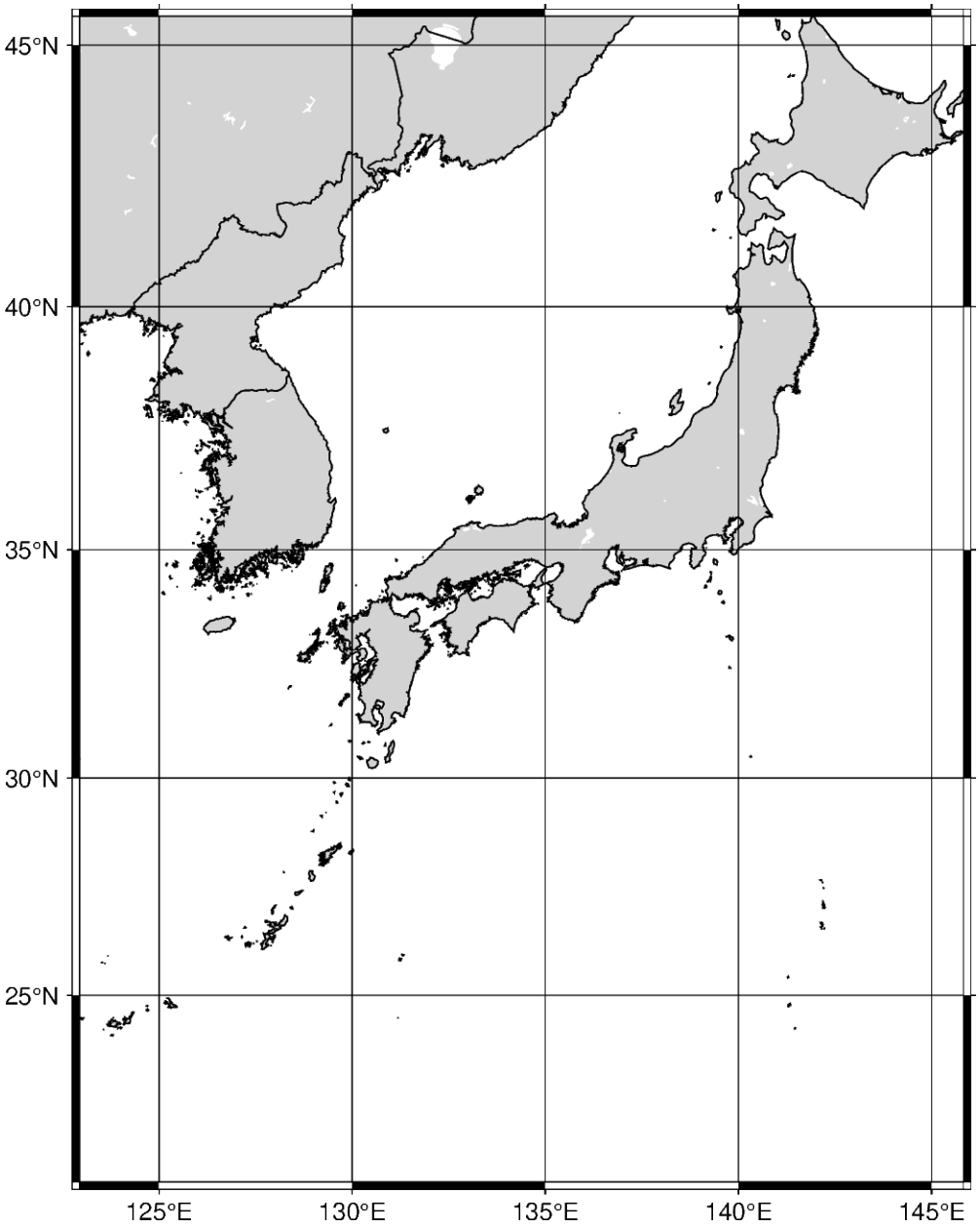
```
fig = pygmt.Figure()
fig.coast(
    region="g",
    projection="Cyl_stere/12c",
    land="darkgray",
    water="white",
    borders="1/0.5p",
    shorelines="1/0.5p",
    frame="ag",
)
fig.show()
```



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()
```



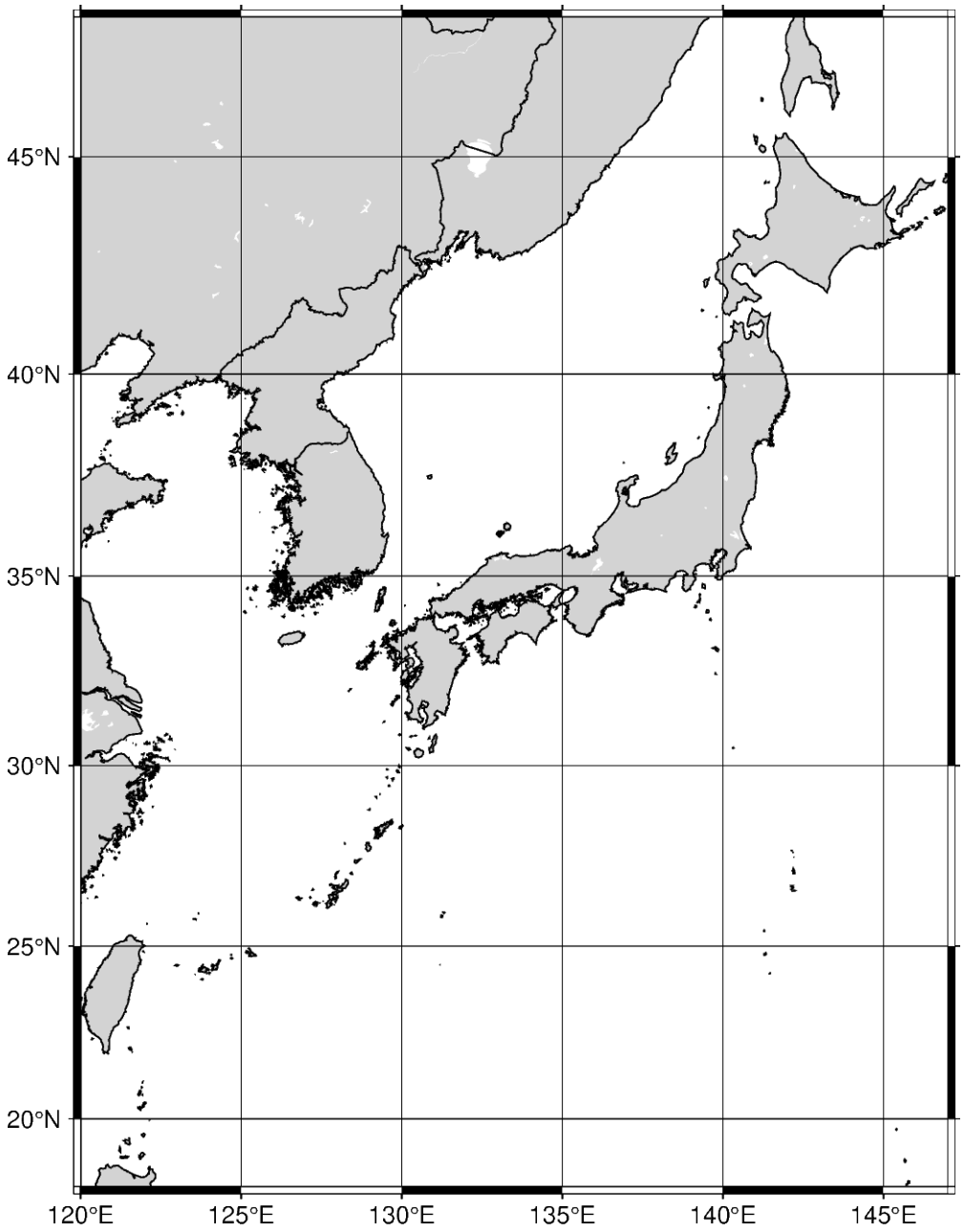
The area encompassed by the ISO code can be expanded by appending **+r***increment* to the ISO code. The *increment* unit is in degrees, and if only one value is added it expands the range of the region in all directions. Using **+r** expands the final region boundaries to be multiples of *increment*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

```
fig = pygmt.Figure()
fig.coast(
    # Expand the region boundaries to be multiples of 3 degrees in all
    # directions
    region="JP+r3",
    projection="M12c",
    land="lightgray",
    water="white",
    borders="1/0.5p",
    shorelines="1/0.5p",
```

(continues on next page)

(continued from previous page)

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



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",
```

(continues on next page)

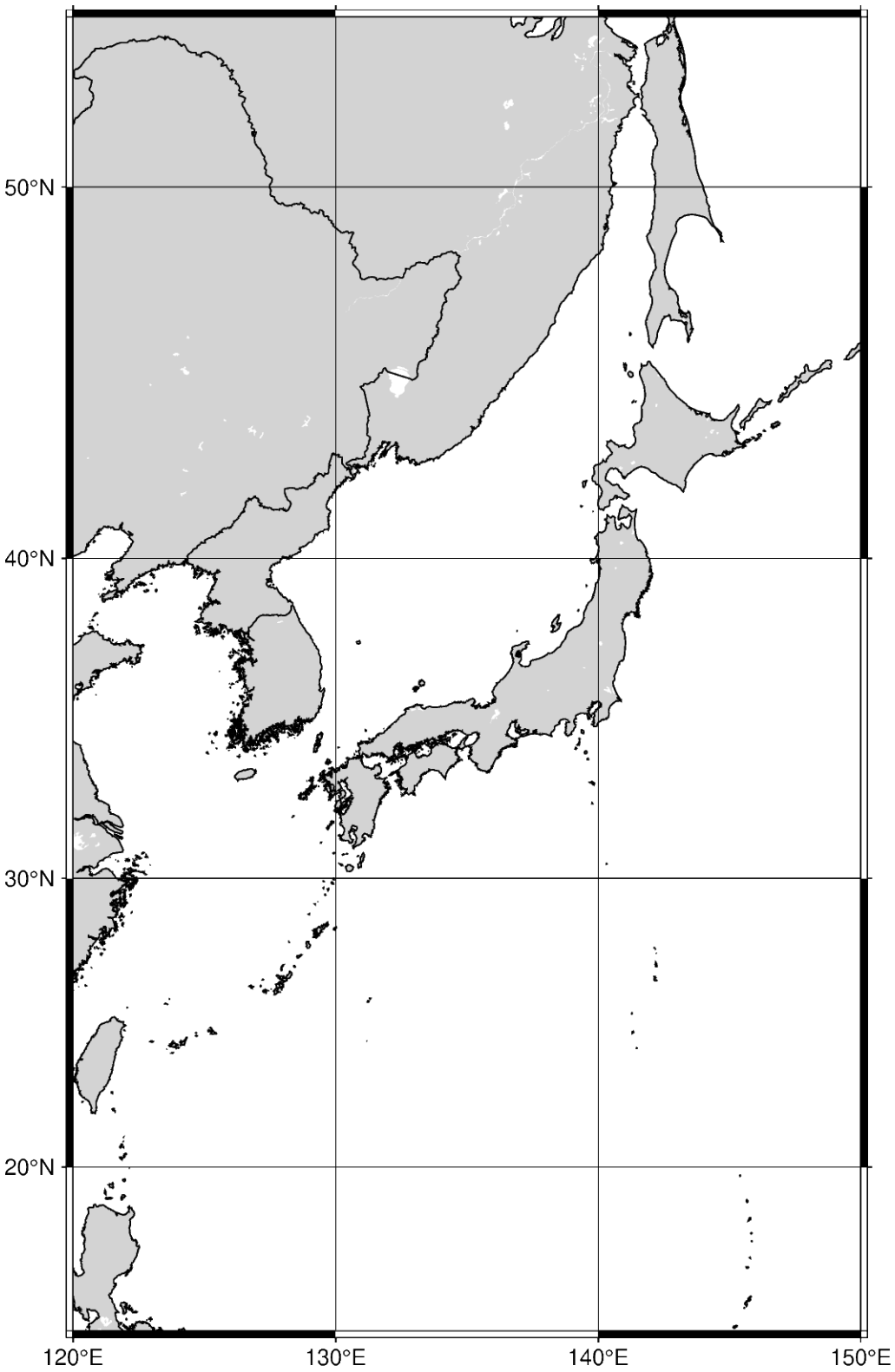
(continued from previous page)

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



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



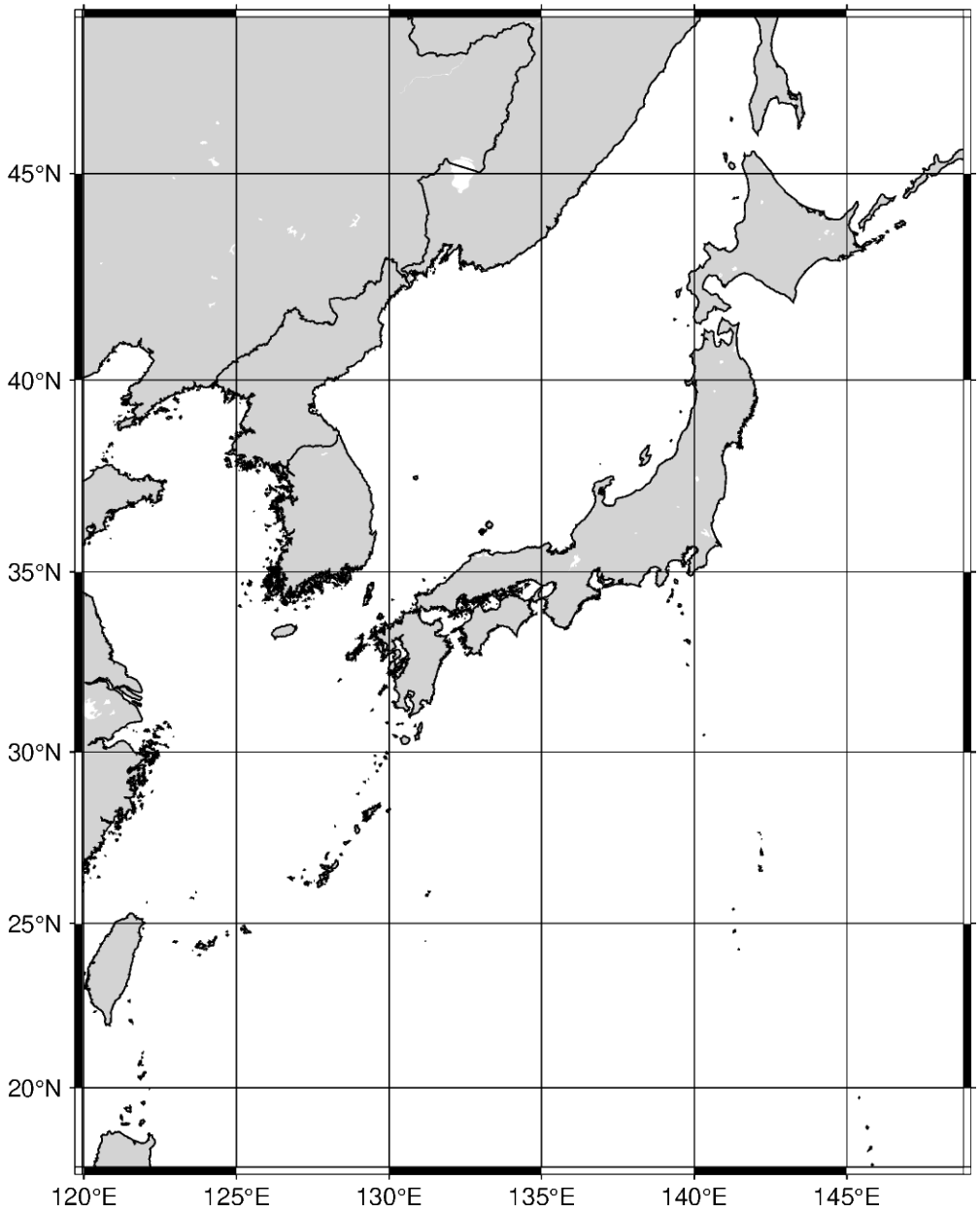
The `region` increment can be appended with **+R**, which adds the increment without rounding.

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

(continues on next page)

(continued from previous page)

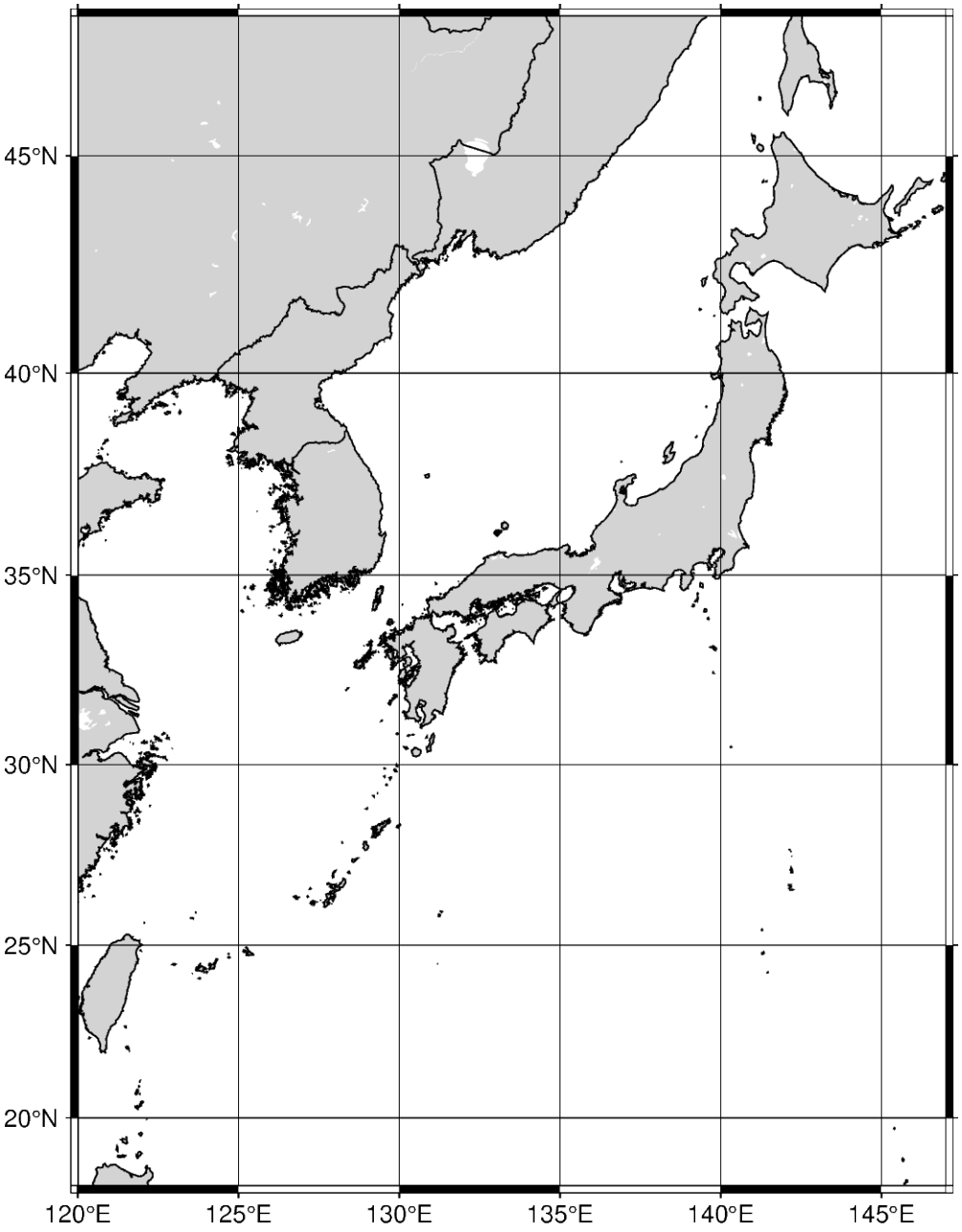
```
    # Expand the region setting outside the range of Japan by 3 degrees in all
    # directions, without rounding to the nearest increment.
    region="JP+R3",
    projection="M12c",
    land="lightgray",
    water="white",
    borders="1/0.5p",
    shorelines="1/0.5p",
    frame="ag",
)
fig.show()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



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



Total running time of the script: (0 minutes 3.448 seconds)

4.2 Advanced

4.2.1 Adding an inset to the figure

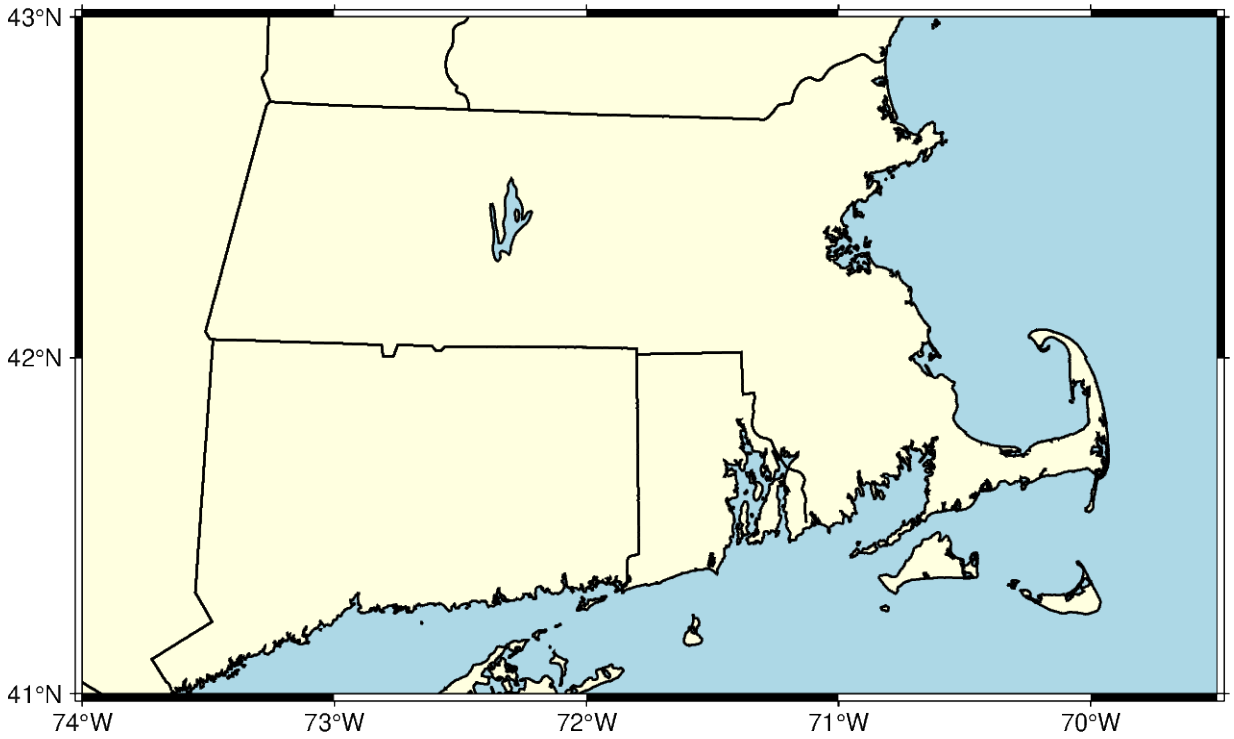
To plot an inset figure inside another larger figure, we can use the *pygmt.Figure.inset* method. After a large figure has been created, call `inset` using a `with` statement, and new plot elements will be added to the inset figure instead of the larger figure.

```
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()
```



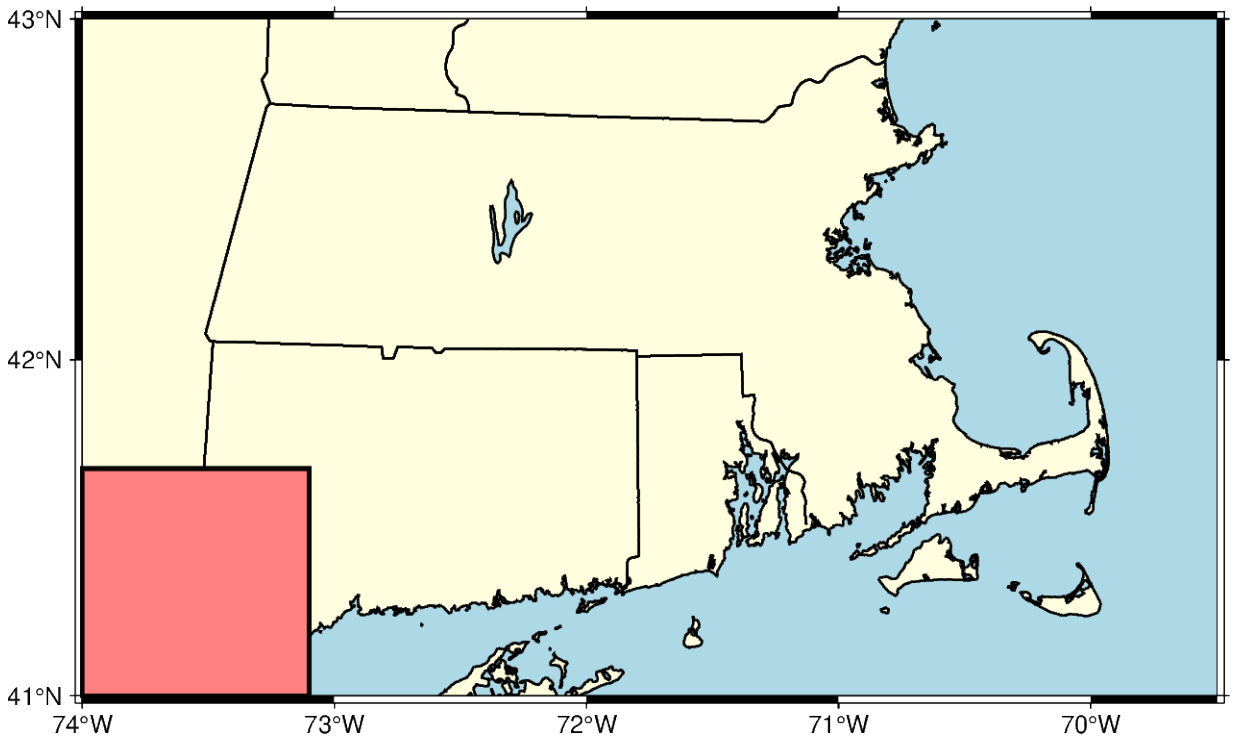
The *pygmt.Figure.inset* method uses a context manager, and is called using a `with` statement. The `position` parameter, including the inset width, is required to plot the inset. Using the **j** modifier, the location of the inset is set to one of the 9 anchors (Top - Middle - Bottom and Left - Center - Right). In the example below, `BL` places the inset at the Bottom Left corner. The `box` parameter can set the fill and border of the inset. In the example below, `+pblack` sets the border color to black and `+glightred` sets the fill to light 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",
)
```

(continues on next page)

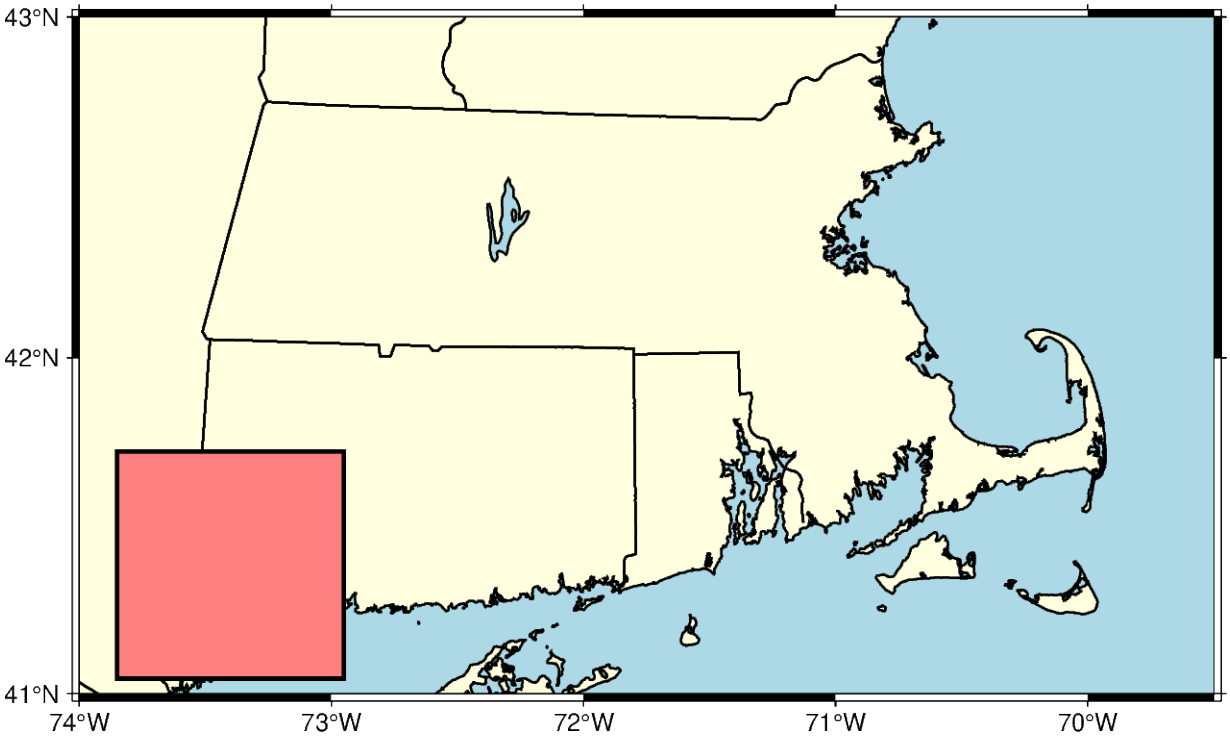
(continued from previous page)

```
with fig.inset(position="jBL+w3c", box="+pblack+glightred"):
    # pass is used to exit the with statement as no plotting methods are
    # called
    pass
fig.show()
```



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.

```
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()
```



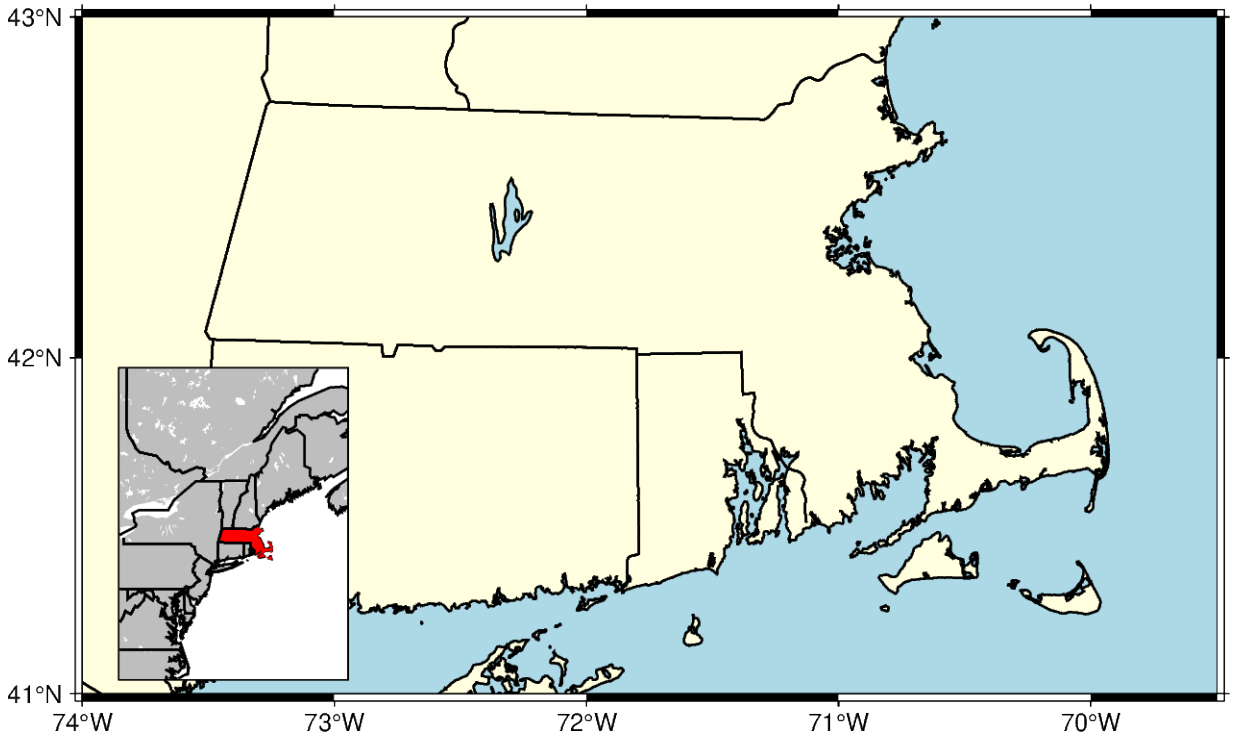
Standard plotting methods can be called from within the `inset` context manager. The example below uses *pygmt.Figure.coast* to plot a zoomed out map that selectively paints the state of Massachusetts to show its location relative to other states.

```
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
# Inset width/height are determined by the ``region`` and ``projection``
# parameters.
with fig.inset(
    position="jBL+o0.5c/0.2c",
    box="+pblack",
    region=[-80, -65, 35, 50],
    projection="M3c",
):
    # Use a plotting method to create a figure inside the inset.
    fig.coast(
        land="gray",
        borders=[1, 2],
        shorelines="1/thin",
        water="white",
        # Use dcw to selectively highlight an area
```

(continues on next page)

(continued from previous page)

```
        dcw="US.MA+gred",
    )
fig.show()
```



Total running time of the script: (0 minutes 1.010 seconds)

4.2.2 Cartesian histograms

Cartesian histograms can be generated using the *pygmt.Figure.histogram* method. In this tutorial, different histogram related aspects are addressed:

- Using vertical and horizontal bars
- Using stair-steps
- Showing counts and frequency percent
- Adding annotations to the bars
- Showing cumulative values
- Using color and pattern as fill for the bars
- Using overlaid, stacked, and grouped bars

Import the required packages

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

Generate random data from a normal distribution:

```
rng = np.random.default_rng(seed=100)

# Mean of distribution
mean = 100
# Standard deviation of distribution
stddev = 20

# Create two data sets
data01 = rng.normal(loc=mean, scale=stddev, size=42)
data02 = rng.normal(loc=mean, scale=stddev * 2, size=42)
```

Vertical and horizontal bars

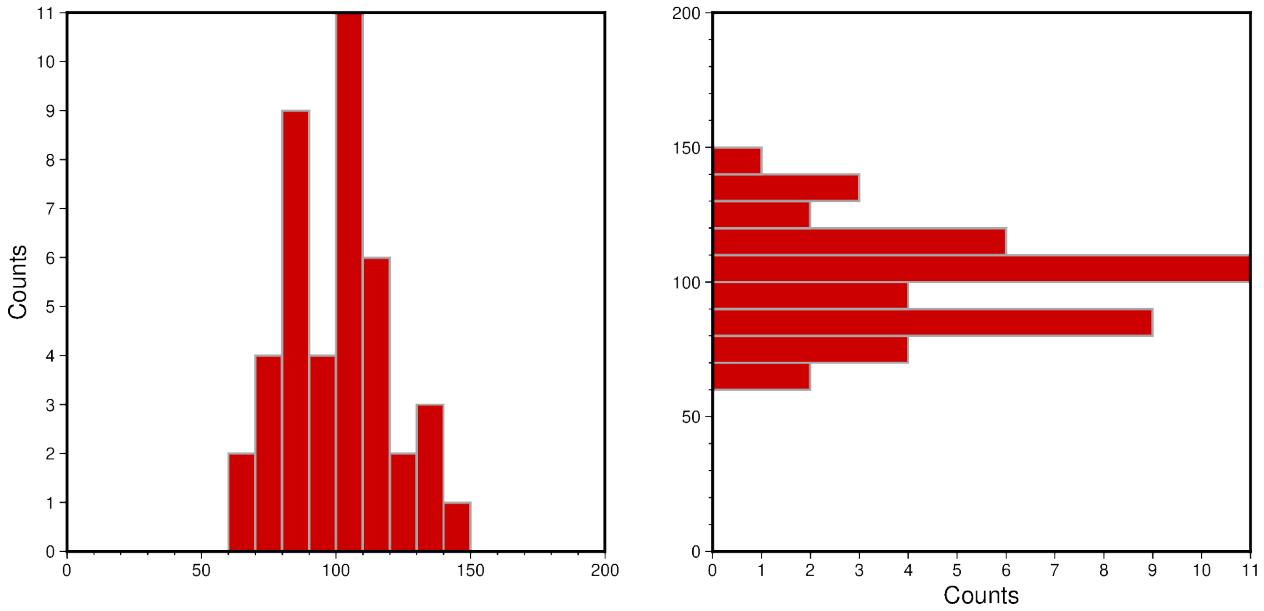
To define the width of the bins, the `series` parameter has to be specified. The bars can be filled via the `fill` parameter with either a color or a pattern (see later in this tutorial). Use the `pen` parameter to adjust width, color, and style of the outlines. By default, a histogram with vertical bars is created. Horizontal bars can be achieved via `horizontal=True`.

```
fig = pygmt.Figure()

# Create histogram for data01 with vertical bars
fig.histogram(
    # Define the plot range as a list of xmin, xmax, ymin, ymax
    # Let ymin and ymax determined automatically by setting both to the same value
    region=[0, 200, 0, 0],
    projection="X10c",  # Cartesian projection with a width of 10 centimeters
    # Add frame, annotations ("a"), ticks ("f"), and y-axis label ("+l") "Counts"; the
    # numbers give the steps of annotations and ticks
    frame=["WStr", "xaf10", "ya1f1+lCounts"],
    data=data01,
    # Set the bin width via the "series" parameter
    series=10,
    # Fill the bars with color "red3"
    fill="red3",
    # Draw a 1-point thick, solid outline in "darkgray" around the bars
    pen="1p,darkgray,solid",
    # Choose counts via the "histtype" parameter
    histtype=0,
)

# Shift plot origin by the figure width ("w") plus 2 centimeters to the right
fig.shift_origin(xshift="w+2c")

# Create histogram for data01 with horizontal bars
fig.histogram(
    region=[0, 200, 0, 0],
    projection="X10c",
    frame=["WStr", "xaf10", "ya1f1+lCounts"],
    data=data01,
    series=10,
    fill="red3",
    pen="1p,darkgray,solid",
    histtype=0,
    # Use horizontal bars. Note that the x- and y-axis are flipped, with the x-axis
```

(continues on next page)

(continued from previous page)

```
    # plotted vertically and the y-axis plotted horizontally.
    horizontal=True,
)

fig.show()
```



Stair-steps

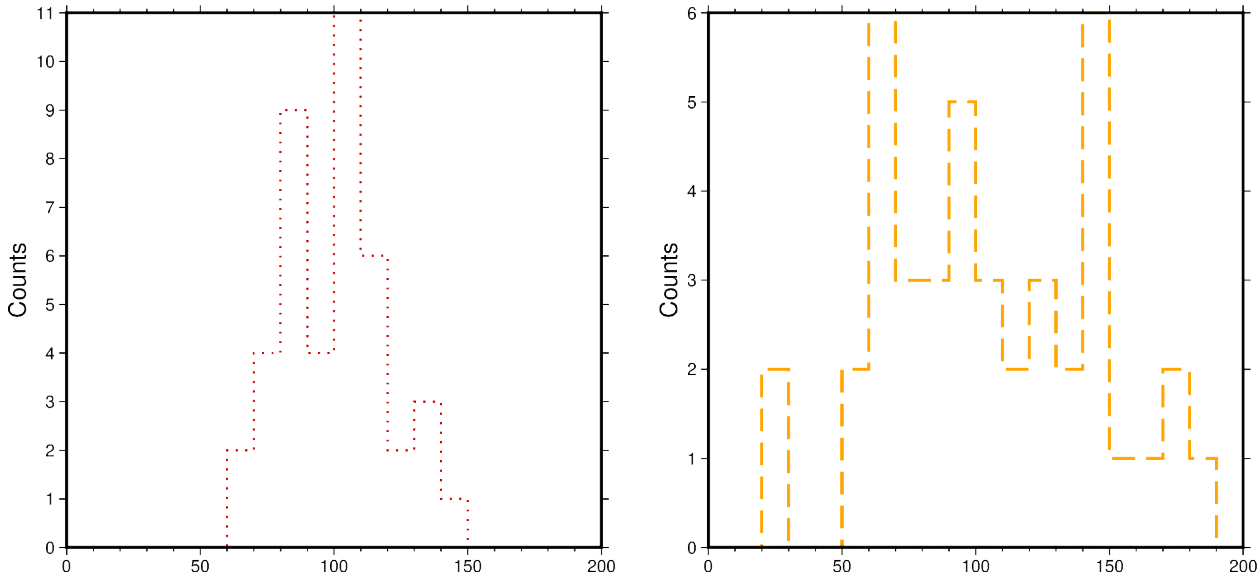
A stair-step diagram can be created by setting `stairs=True`. Then only the outer outlines of the bars are drawn, and no internal bars are visible.

```
fig = pygmt.Figure()

# Create histogram for data01
fig.histogram(
    region=[0, 200, 0, 0],
    projection="X10c",
    frame=["WSne", "xaf10", "ya1f1+lCounts"],
    data=data01,
    series=10,
    # Draw a 1-point thick, dotted outline in "red3"
    pen="1p,red3,dotted",
    histtype=0,
    # Draw stair-steps in stead of bars
    stairs=True,
)

fig.shift_origin(xshift="w+2c")

# Create histogram for data02
fig.histogram(
    region=[0, 200, 0, 0],
    projection="X10c",
```

(continues on next page)

(continued from previous page)

```
    frame=["WSne", "xaf10", "ya1f1+lCounts"],
    data=data02,
    series=10,
    # Draw a 1.5-points thick, dashed outline in "orange"
    pen="1.5p,orange,dashed",
    histtype=0,
    stairs=True,
)

fig.show()
```



Counts and frequency percent

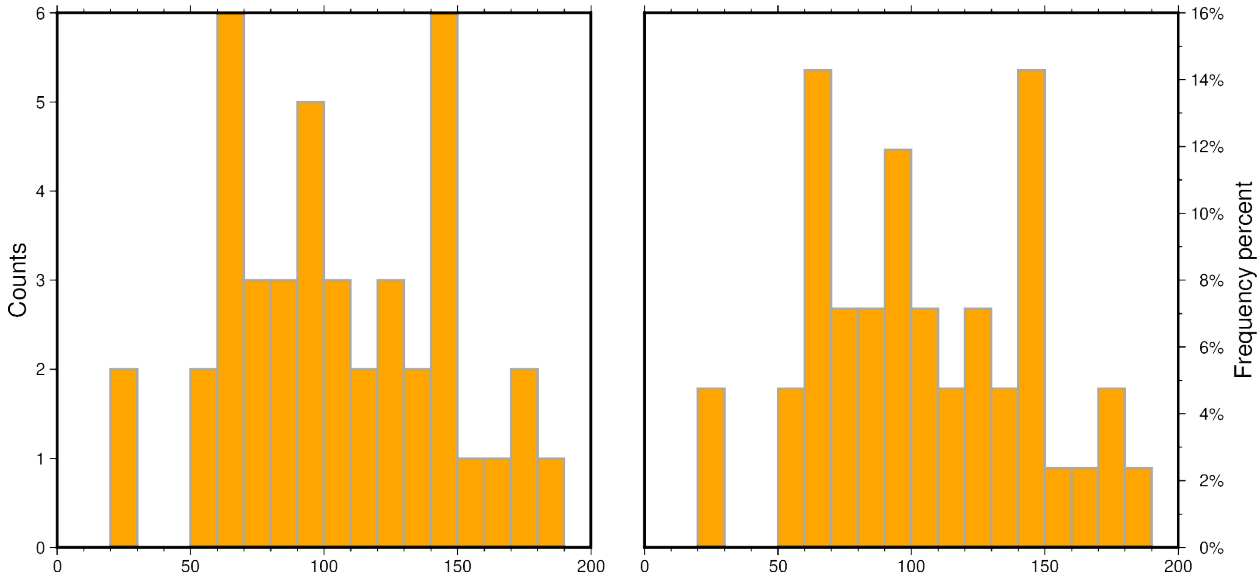
By default, a histogram showing the counts in each bin is created (`histtype=0`). To show the frequency percent set the `histtype` parameter to `1`. For further options please have a look at the documentation of *pygmt.Figure.histogram*.

```
fig = pygmt.Figure()

# Create histogram for data02 showing counts
fig.histogram(
    region=[0, 200, 0, 0],
    projection="X10c",
    frame=["WSnr", "xaf10", "ya1f1+lCounts"],
    data=data02,
    series=10,
    fill="orange",
    pen="1p,darkgray,solid",
    # Choose counts via the "histtype" parameter
    histtype=0,
)

fig.shift_origin(xshift="w+1c")

# Create histogram for data02 showing frequency percent
```

(continues on next page)

(continued from previous page)

```
fig.histogram(
    region=[0, 200, 0, 0],
    projection="X10c",
    # Add suffix % (+u)
    frame=["lSnE", "xaf10", "ya2f1+u%+lFrequency percent"],
    data=data02,
    series=10,
    fill="orange",
    pen="1p,darkgray,solid",
    # Choose frequency percent via the "histtype" parameter
    histtype=1,
)

fig.show()
```



Cumulative values

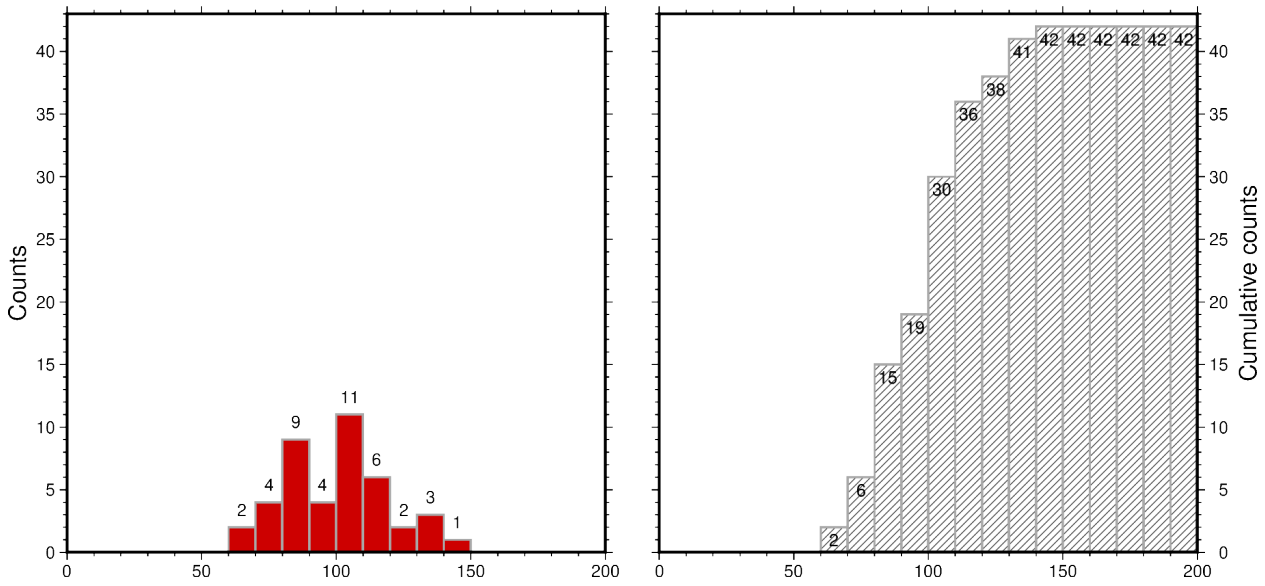
To create a histogram showing the cumulative values set `cumulative=True`. Here, the bars of the cumulative histogram are filled with a pattern via the `fill` parameter. Annotate each bar with the counts it represents using the `annotate` parameter.

```
fig = pygmt.Figure()

# Create histogram for data01 showing the counts per bin
fig.histogram(
    region=[0, 200, 0, len(data01) + 1],
    projection="X10c",
    frame=["WSne", "xaf10", "ya5f1+lCounts"],
    data=data01,
    series=10,
    fill="red3",
    pen="1p,darkgray,solid",
    histtype=0,
    # Annotate each bar with the counts it represents
```

(continues on next page)

(continued from previous page)

```
    annotate=True,
)

fig.shift_origin(xshift="w+1c")

# Create histogram for data01 showing the cumulative counts
fig.histogram(
    region=[0, 200, 0, len(data01) + 1],
    projection="X10c",
    frame=["wSnE", "xaf10", "ya5f1+lCumulative counts"],
    data=data01,
    series=10,
    # Use pattern ("p") number 8 as fill for the bars
    # Set the background ("+b") to white [Default]
    # Set the foreground ("+f") to black [Default]
    fill="p8+bwhite+fblack",
    pen="1p,darkgray,solid",
    histtype=0,
    # Show cumulative counts
    cumulative=True,
    # Offset ("+o") the label by 10 points in negative y-direction
    annotate="+o-10p",
)

fig.show()
```



Overlaid bars

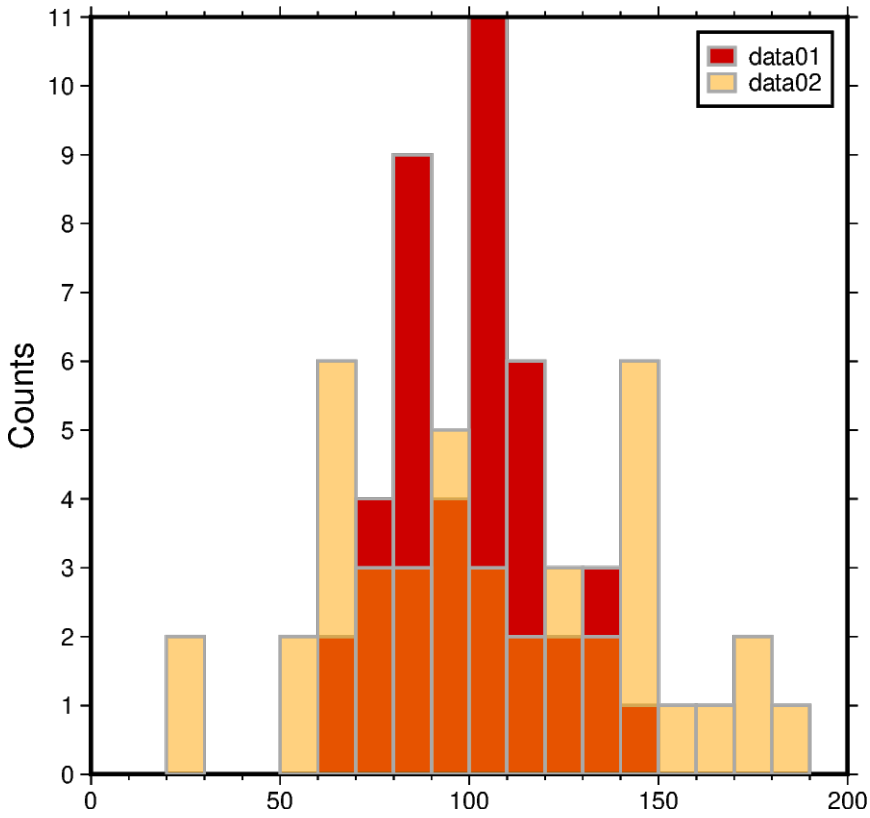
Overlaid or overlapping bars can be achieved by plotting two or several histograms, each for one data set, on top of each other. The legend entry can be specified via the `label` parameter.

Limitations of histograms with overlaid bars are:

- Mixing of colors or/and patterns

- Visually more colors or/and patterns than data sets
- Visually a “third histogram” (or more in case of more than two data sets)

```
fig = pygmt.Figure()

# Create histogram for data01
fig.histogram(
    region=[0, 200, 0, 0],
    projection="X10c",
    frame=["WSne", "xaf10", "ya1f1+lCounts"],
    data=data01,
    series=10,
    fill="red3",
    pen="1p,darkgray,solid",
    histtype=0,
    # Set legend entry
    label="data01",
)

# Create histogram for data02
# It is plotted on top of the histogram for data01
fig.histogram(
    data=data02,
    series=10,
    # Fill bars with color "orange", use a transparency of 50% ("@50")
    fill="orange@50",
    pen="1p,darkgray,solid",
    histtype=0,
    label="data02",
)

# Add legend
fig.legend()

fig.show()
```



Stacked bars

Histograms with stacked bars are not directly supported by PyGMT. Thus, before plotting, combined data sets have to be created from the single data sets. Then, stacked bars can be achieved similar to overlaid bars via plotting two or several histograms on top of each other.

Limitations of histograms with stacked bars are:

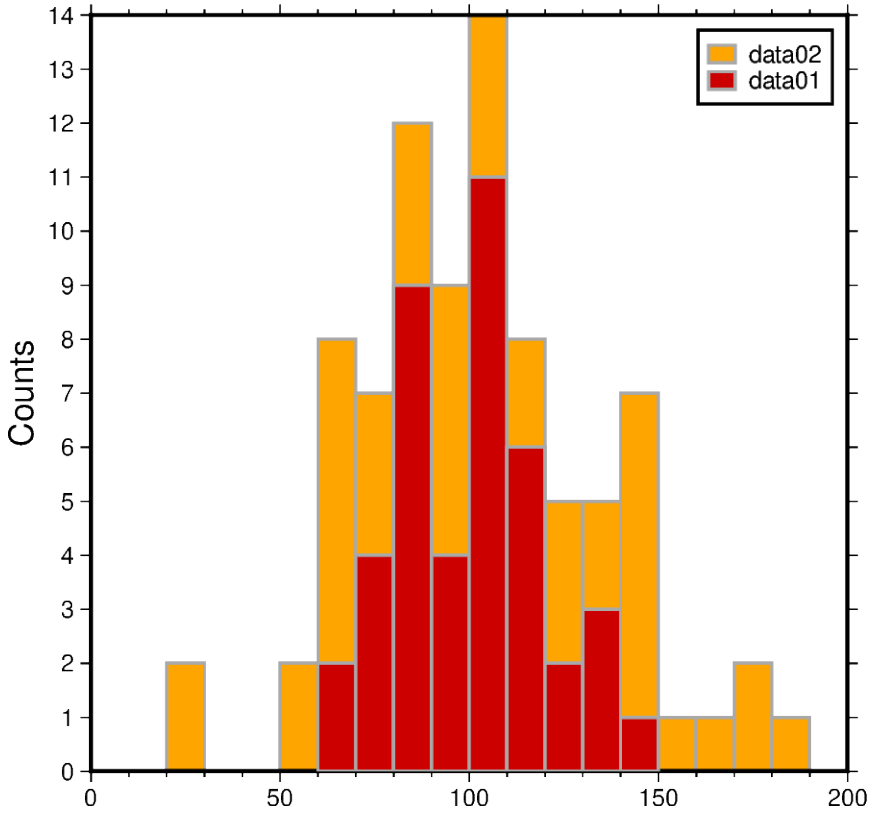
- No common baseline
- Partly not directly clear whether overlaid or stacked bars

```
# Combine the two data sets to one data set
data_merge = np.concatenate((data01, data02), axis=None)

fig = pygmt.Figure()

# Create histogram for data02 by using the combined data set
fig.histogram(
    region=[0, 200, 0, 0],
    projection="X10c",
    frame=["WSne", "xaf10", "ya1f1+lCounts"],
    data=data_merge,
    series=10,
    fill="orange",
    pen="1p,darkgray,solid",
    histtype=0,
    # The combined data set appears in the final histogram visually as data set data02
    label="data02",
```

(continues on next page)

(continued from previous page)

```
)

# Create histogram for data01
# It is plotted on top of the histogram for data02
fig.histogram(
    data=data01,
    series=10,
    fill="red3",
    pen="1p,darkgray,solid",
    histtype=0,
    label="data01",
)

# Add legend
fig.legend()

fig.show()
```



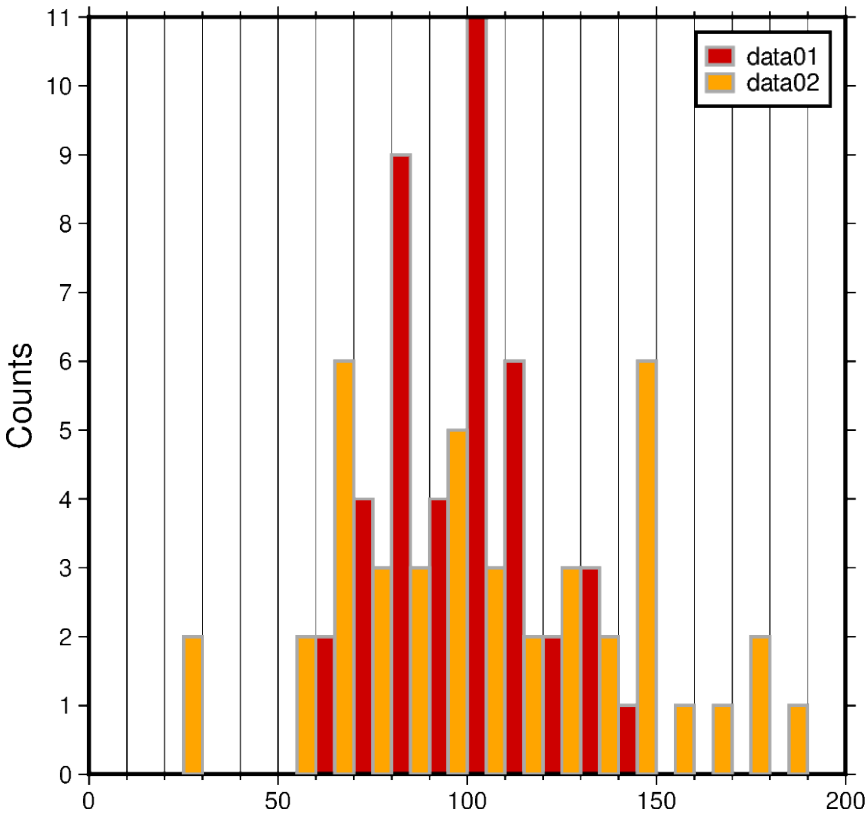
Grouped bars

By setting the `barwidth` parameter in respect to the values passed to the `series` parameter histograms with grouped bars can be created.

Limitations of histograms with grouped bars are:

- Careful setting width and position of the bars in respect to the bin width
- Difficult to see the variations of the single data sets

```
# Width used for binning the data
binwidth = 10

fig = pygmt.Figure()

# Create histogram for data01
fig.histogram(
    region=[0, 200, 0, 0],
    projection="X10c",
    frame=["WSne", "xaf10g10", "ya1f1+lCounts"],
    data=data01,
    series=binwidth,
    fill="red3",
    pen="1p,darkgray,solid",
    histtype=0,
    # Calculate the bar width in respect to the bin width, here for two data sets half
    # of the bin width
    # Offset ("+o") the bars to align each bar with the left limit of the␣
↪corresponding
    # bin
    barwidth=f"{binwidth / 2}+o-{binwidth / 4}",
    label="data01",
)

# Create histogram for data02
fig.histogram(
    data=data02,
    series=binwidth,
    fill="orange",
    pen="1p,darkgray,solid",
    histtype=0,
    barwidth=f"{binwidth / 2}+o{binwidth / 4}",
    label="data02",
)

# Add legend
fig.legend()

fig.show()
```



Total running time of the script: (0 minutes 1.279 seconds)

4.2.3 Configuring PyGMT defaults

Default GMT parameters can be set globally or locally using *pygmt.config*.

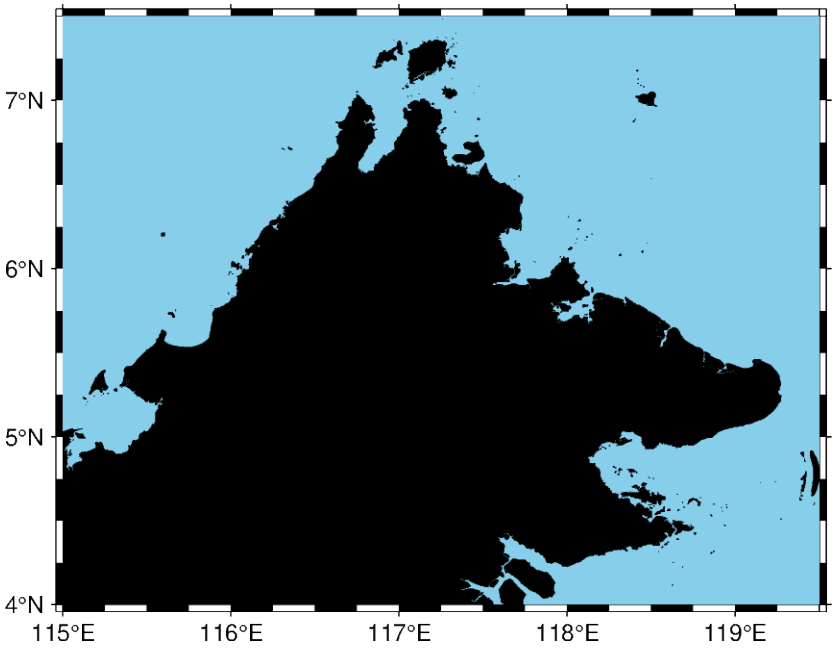
```
import pygmt
```

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/6.5/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()
```

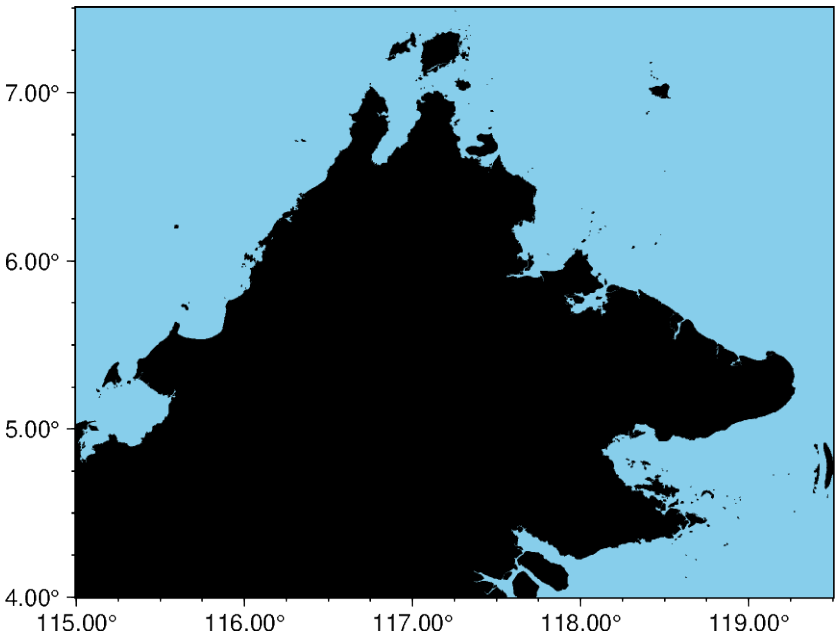


Globally overriding defaults

The `MAP_FRAME_TYPE` parameter specifies the style of map frame to use, of which there are 5 options: `fancy` (default, see 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.

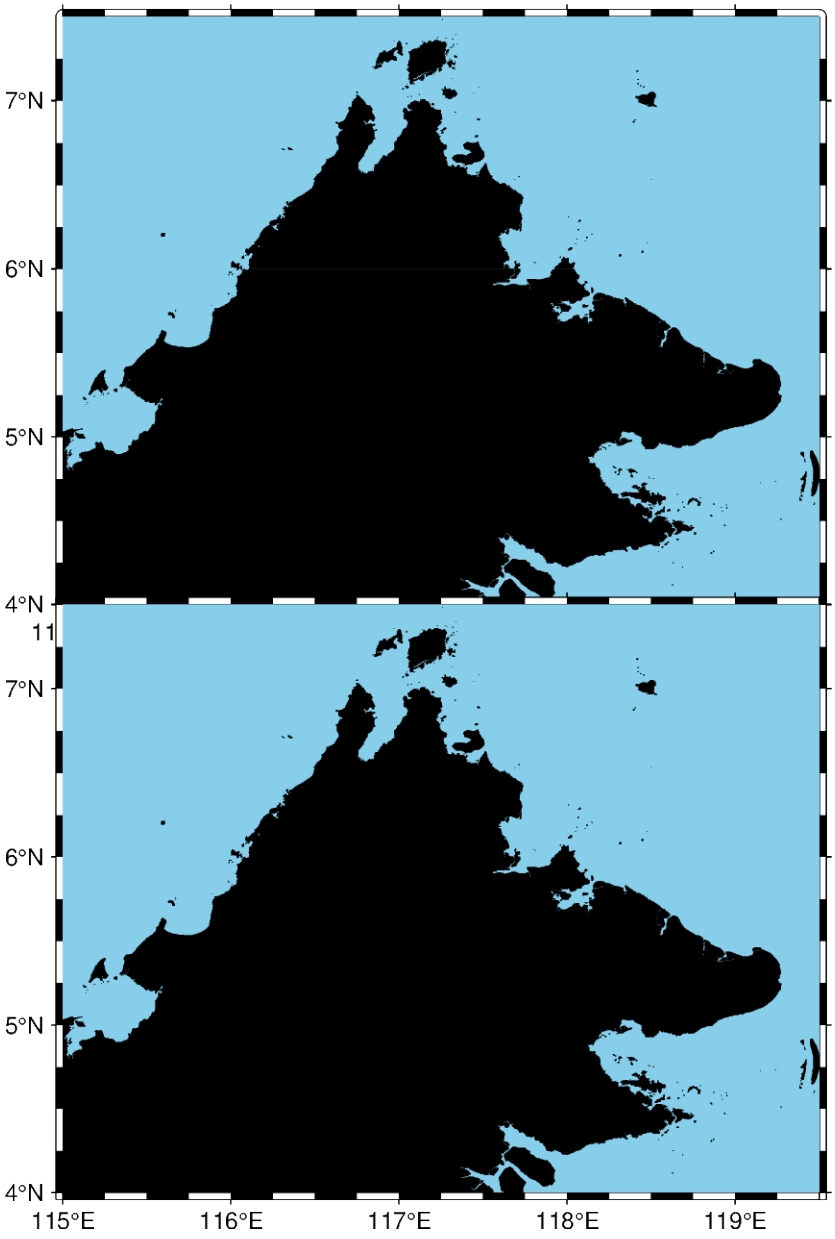
```
fig = pygmt.Figure()

# 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()
```



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()

# 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 the height of the figure to plot another map
fig.shift_origin(yshift="-h")

# 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()
```



Total running time of the script: (0 minutes 0.549 seconds)

4.2.4 Creating a 3-D perspective image

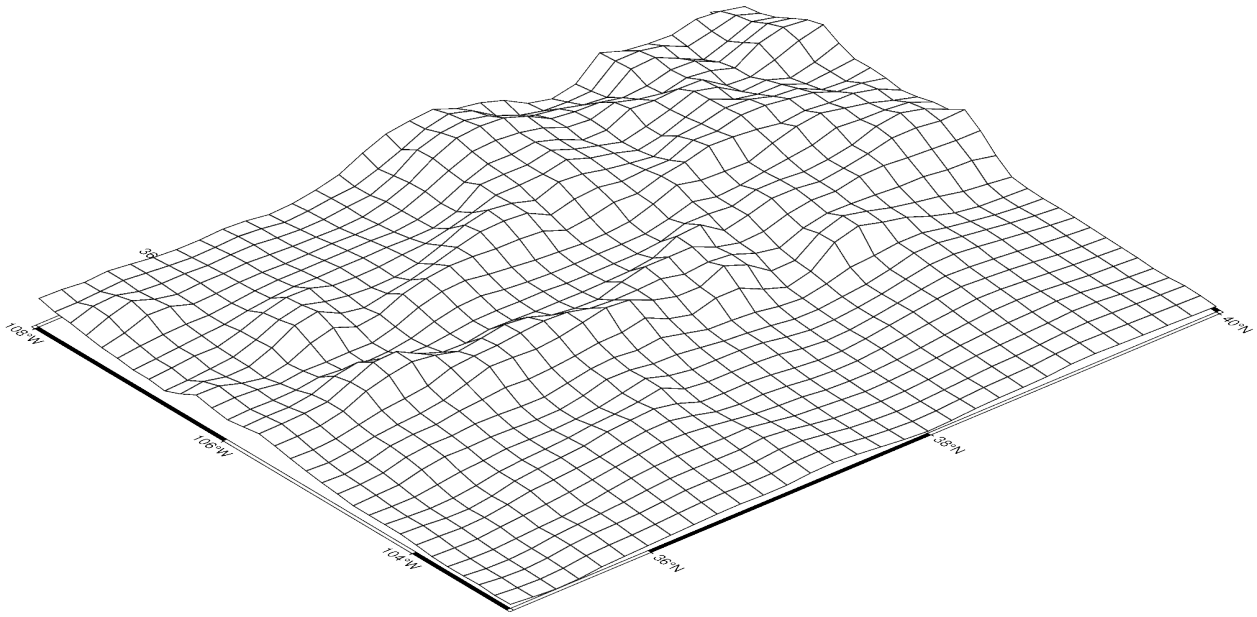
Create 3-D perspective image or surface mesh from a grid using *pygmt.Figure.grdview*.

```
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.

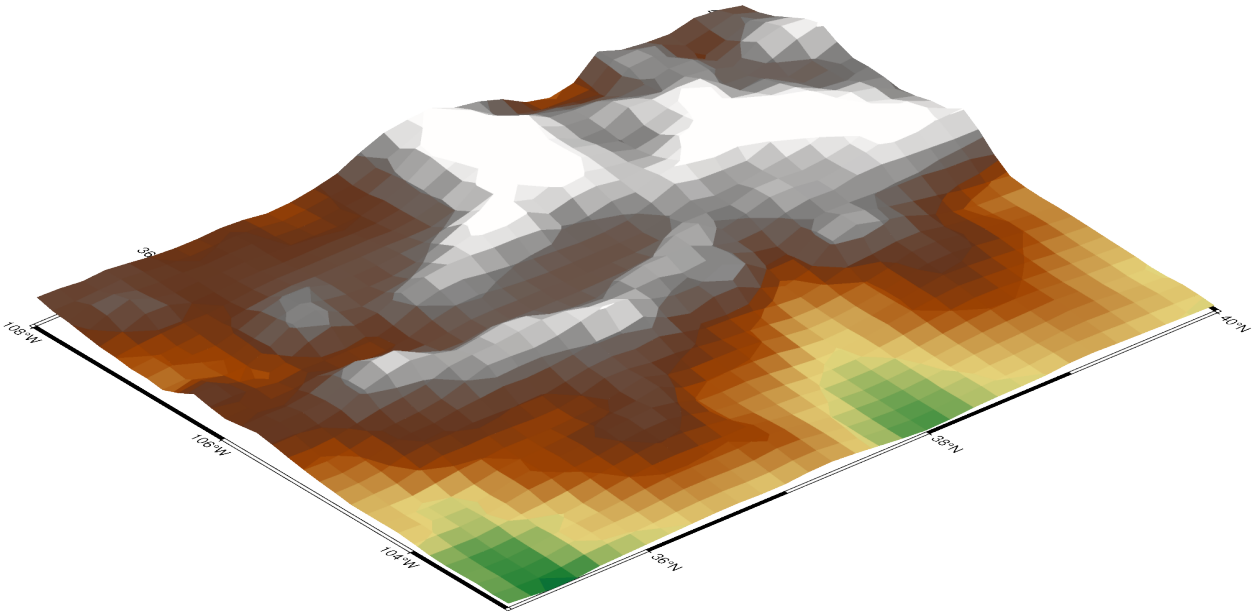
The default grid surface type is *mesh plot*.

```
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()
```



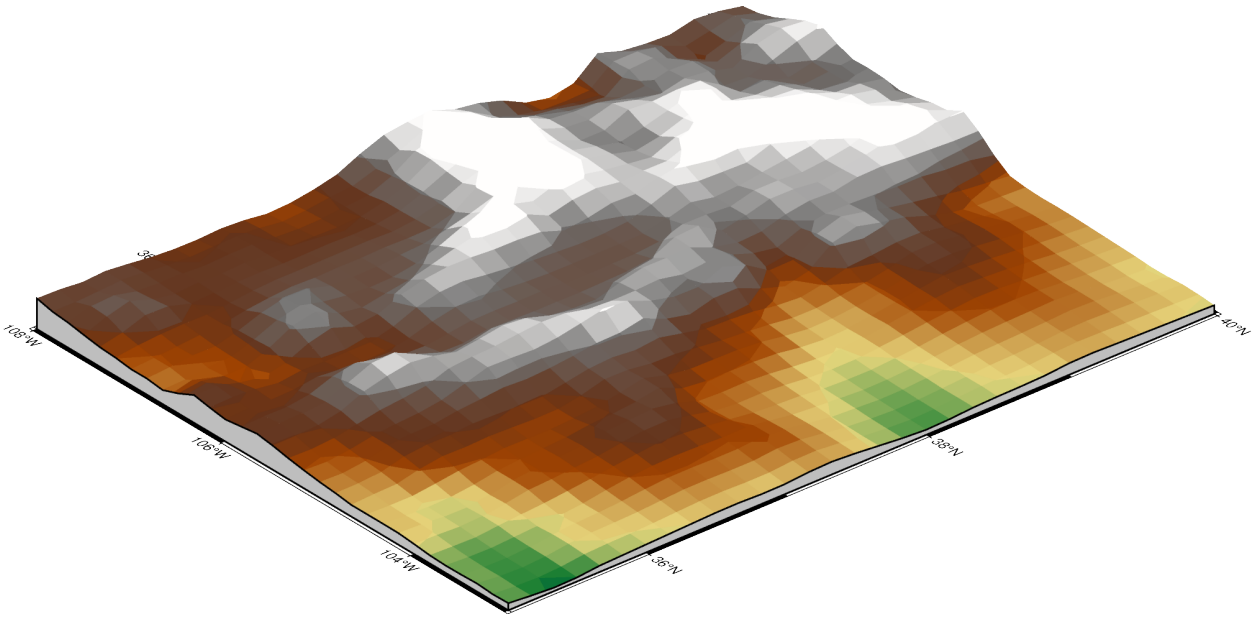
The grid surface type can be set with the `surftype` parameter. The default CPT is *turbo* and can be customized with the `cmap` parameter.

```
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()
```



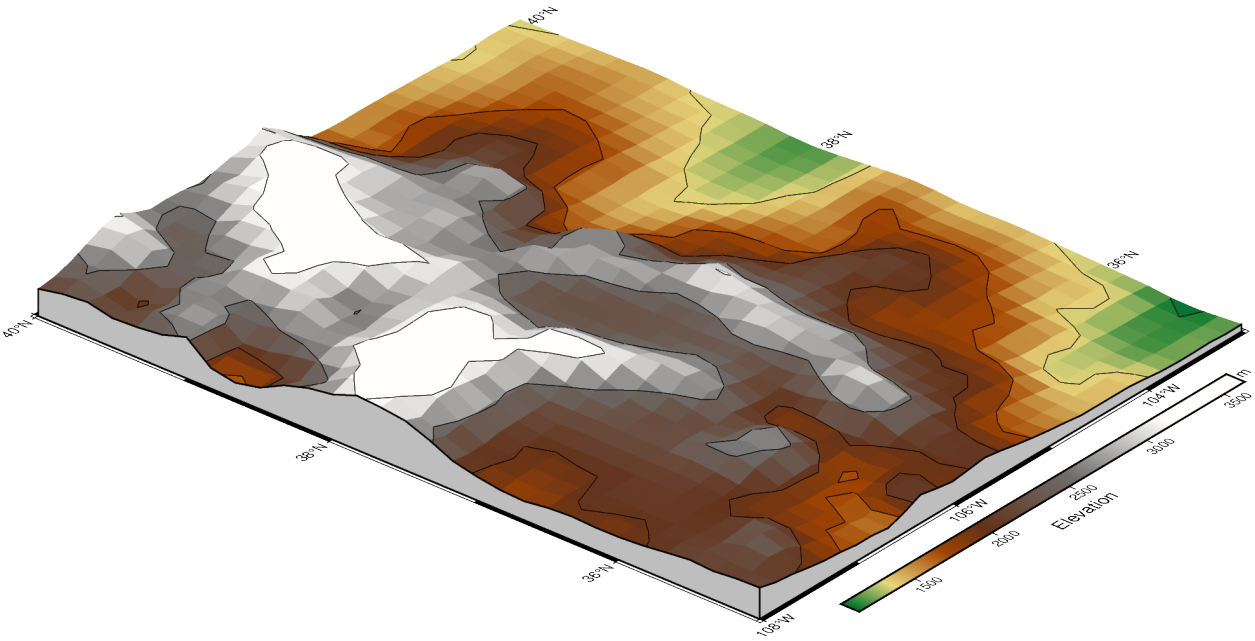
The `plane` parameter sets the elevation and color of a plane that provides a fill below the surface relief.

```
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",
)
fig.show()
```



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 argument for 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()
```



Total running time of the script: (0 minutes 0.974 seconds)

4.2.5 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])
```

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 1,000 meters. By default, it plots the map with the equidistant cylindrical projection and with no frame.

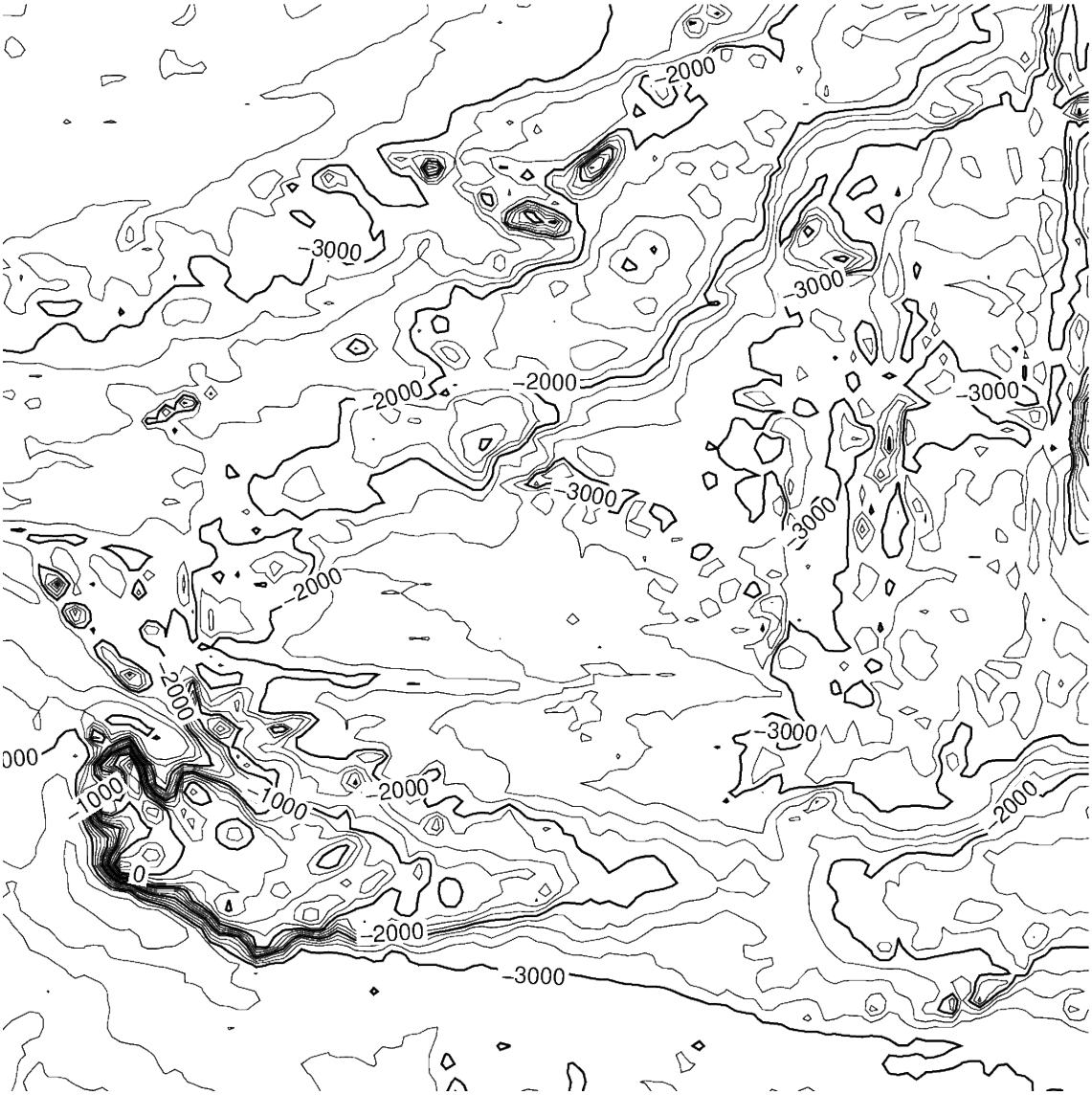
```
fig = pygmt.Figure()
fig.grdcontour(grid=grid)
fig.show()
```



Contour line settings

Use the `annotation` and `levels` parameters to adjust contour line intervals. In the example below, there are contour intervals every 250 meters and annotated contour lines every 1,000 meters.

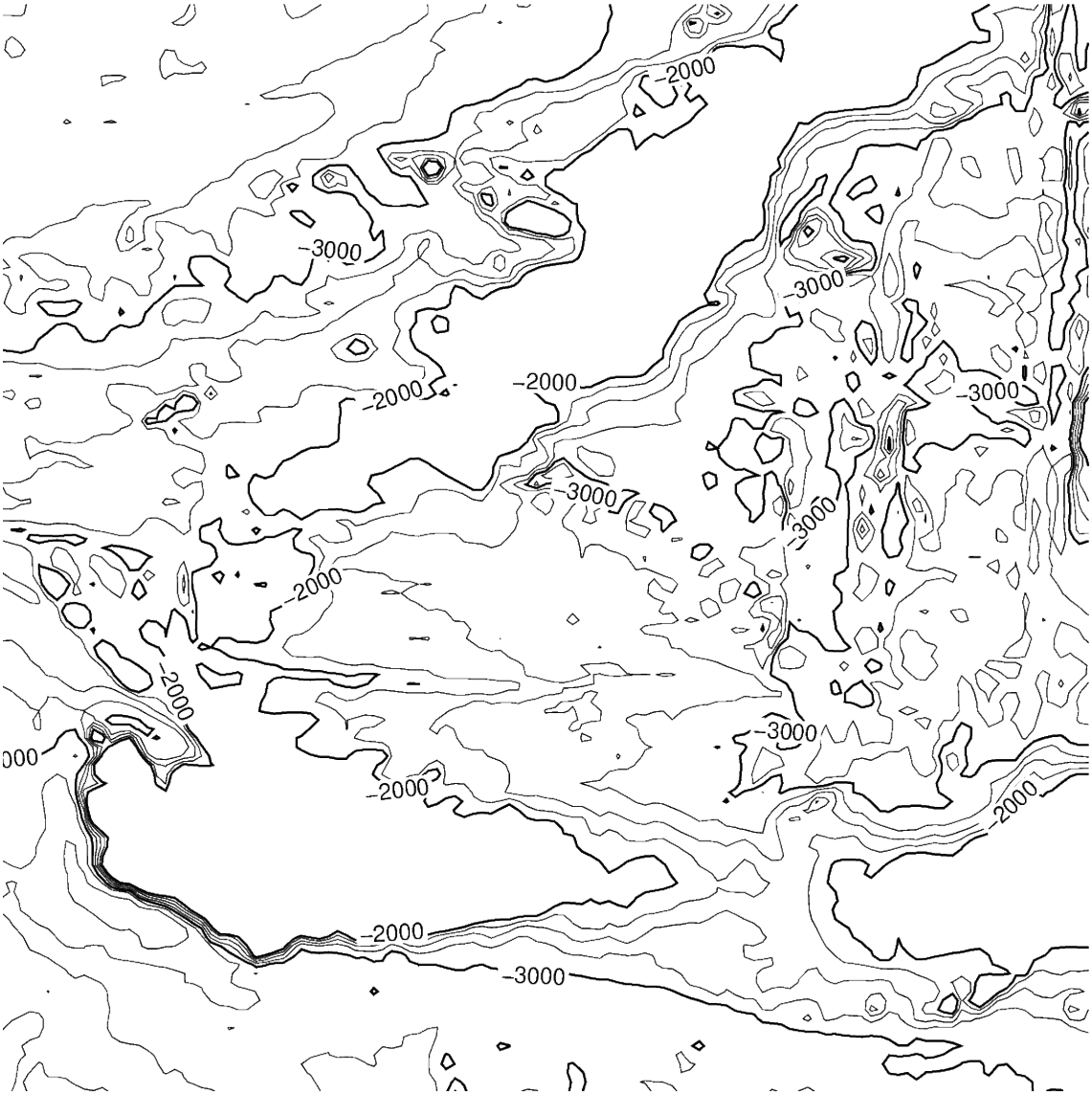
```
fig = pygmt.Figure()
fig.grdcontour(grid=grid, annotation=1000, levels=250)
fig.show()
```



Contour limits

The `limit` parameter sets the minimum and maximum values for the contour lines. The parameter takes the low and high values, and is either a list (as below) or a string `limit="-4000/-2000"`.

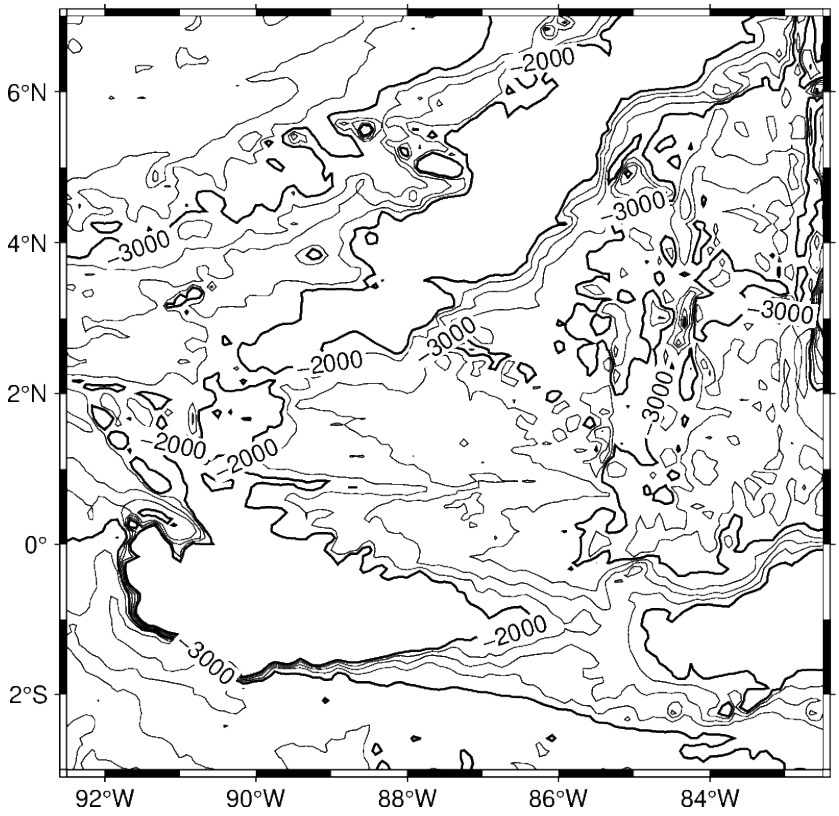
```
fig = pygmt.Figure()
fig.grdcontour(grid=grid, annotation=1000, levels=250, limit=[-4000, -2000])
fig.show()
```



Map settings

The *pygmt.Figure.grdcontour* method accepts additional parameters, including setting the projection and frame.

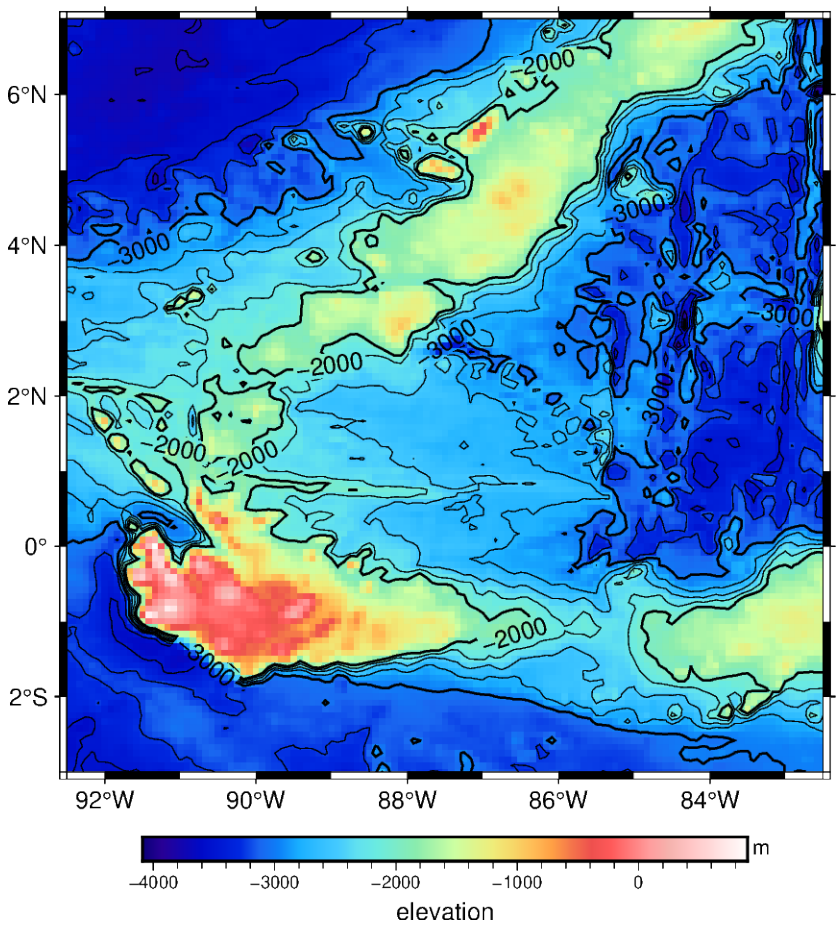
```
fig = pygmt.Figure()
fig.grdcontour(
    grid=grid,
    annotation=1000,
    levels=250,
    limit=[-4000, -2000],
    projection="M10c",
    frame=True,
)
fig.show()
```



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` parameter is specified in the *pygmt.Figure.grdimage* method, it does not need to be repeated in the *pygmt.Figure.grdcontour* method. Finally, a colorbar is added using the *pygmt.Figure.colorbar* method.

```
fig = pygmt.Figure()
fig.grdimage(grid=grid, cmap="haxby", projection="M10c", frame=True)
fig.grdcontour(grid=grid, annotation=1000, levels=250, limit=[-4000, -2000])
fig.colorbar(frame=["x+lelevation", "y+lm"])
fig.show()
```



Total running time of the script: (0 minutes 1.456 seconds)

4.2.6 Creating legends

The *pygmt.Figure.legend* method creates legends, whereby auto-legends as well as manually created legends are supported.

```
import io

import pygmt
```

Create an auto-legend

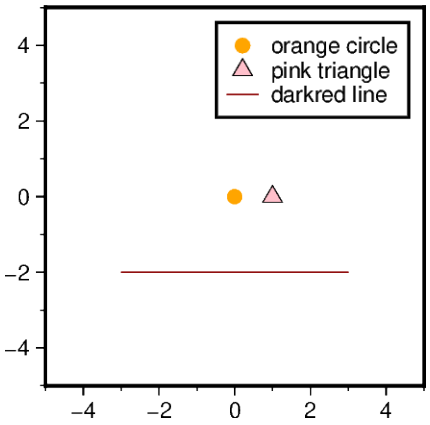
An auto-legend can be created for the methods *pygmt.Figure.plot* and *pygmt.Figure.plot3d*, *pygmt.Figure.hlines* and *pygmt.Figure.vlines* as well as *pygmt.Figure.histogram*. Therefore the `label` parameter has to be specified to state the desired text for the legend entry (white spaces are supported). Here, we use *pygmt.Figure.plot*, exemplary. By default, the legend is placed in the Upper Right corner with an offset of 0.1 centimeters in both x and y directions, and surrounded by a box with a white fill and a 1-point thick, black, solid outline. The order of the legend entries (top to bottom) is determine by the plotting order. Optionally, to adjust the legend, append different modifiers to the string passed to `label`. For a list of available modifiers see https://docs.generic-mapping-tools.org/6.5/gmt.html#l-full. To create a *multiple-column legend* **+N** is used with the desired number of columns.

```
fig = pygmt.Figure()
fig.basemap(region=[-5, 5, -5, 5], projection="X5c", frame=True)
```

(continues on next page)

(continued from previous page)

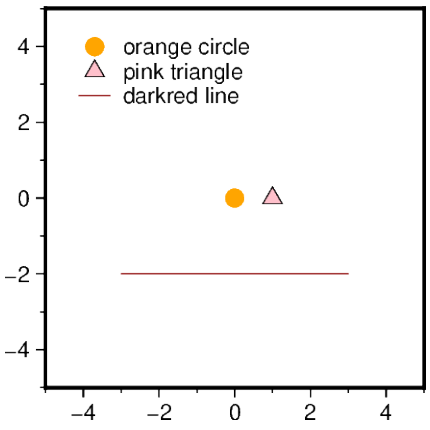
```
# Plot two data points and one line
fig.plot(x=0, y=0, style="c0.2c", fill="orange", label="orange circle")
fig.plot(x=1, y=0, style="t0.3c", fill="pink", pen="black", label="pink triangle")
fig.plot(x=[-3, 3], y=[-2, -2], pen="darkred", label="darkred line")

# Add a legend based on the explanation text given via the "label" parameter.
fig.legend()

fig.show()
```



Adjust the position

Use the `position` parameter to adjust the position of the legend. Add an offset via **+o** for the x and y directions. Additionally append **+w** to adjust the width of the legend. Note, no box is drawn by default if `position` is used.

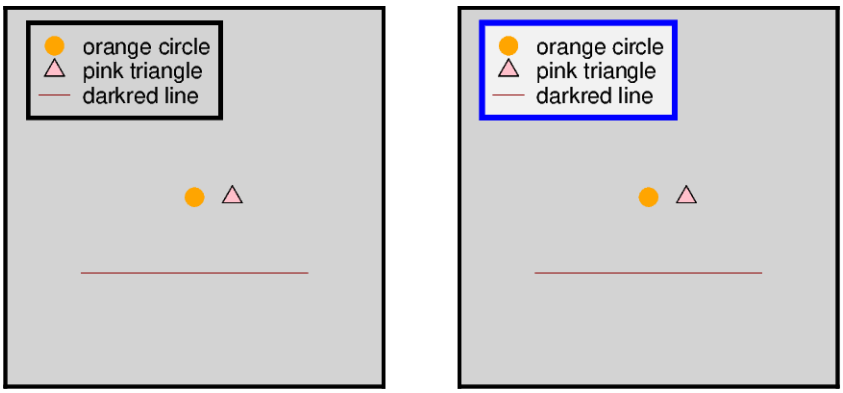
```
fig = pygmt.Figure()
fig.basemap(region=[-5, 5, -5, 5], projection="X5c", frame=True)

fig.plot(x=0, y=0, style="c0.25c", fill="orange", label="orange circle")
fig.plot(x=1, y=0, style="t0.3c", fill="pink", pen="black", label="pink triangle")
fig.plot(x=[-3, 3], y=[-2, -2], pen="darkred", label="darkred line")

# Set the reference point to the Top Left corner within (lowercase "j") the bounding␣
↪box
# of the plot and use offsets of 0.3 and 0.2 centimeters in the x and y directions,
# respectively.
fig.legend(position="jTL+o0.3c/0.2c")

fig.show()
```



Add a box

Use the `box` parameter for adjusting the box around the legend. The outline of the box can be adjusted by appending **+p**. Append **+g** to fill the legend with a color (or pattern) [Default is no fill]. The default of `position` is preserved.

```
fig = pygmt.Figure()
fig.basemap(region=[-5, 5, -5, 5], projection="X5c", frame="rltb+glightgray")

fig.plot(x=0, y=0, style="c0.25c", fill="orange", label="orange circle")
fig.plot(x=1, y=0, style="t0.3c", fill="pink", pen="black", label="pink triangle")
fig.plot(x=[-3, 3], y=[-2, -2], pen="darkred", label="darkred line")

fig.legend(position="jTL+o0.3c/0.2c", box=True)

fig.shift_origin(xshift="w+1c")
fig.basemap(region=[-5, 5, -5, 5], projection="X5c", frame="rltb+glightgray")

fig.plot(x=0, y=0, style="c0.25c", fill="orange", label="orange circle")
fig.plot(x=1, y=0, style="t0.3c", fill="pink", pen="black", label="pink triangle")
fig.plot(x=[-3, 3], y=[-2, -2], pen="darkred", label="darkred line")

# Add a box with a 2-points thick blue, solid outline and a white fill with a
# transparency of 70 percentage ("@30").
fig.legend(position="jTL+o0.3c/0.2c", box="+p2p,blue+gwhite@30")

fig.show()
```



Create a manual legend

For more complicated legends, users need to prepare a legend specification with instructions for the layout of the legend entries. In PyGMT, the legend specification can be either an ASCII file or an `io.StringIO` object. Both are passed to the `spec` parameter of *`pygmt.Figure.legend`*. Multiple legend codes are available to create complicated legends. In the example below we show an subset; a full overview can be found at https://docs.generic-mapping-tools.org/6.5/legend.html#legend-codes. It's also supported to include length scales (for geographic projections), faults, and images as well as to add specific lines.

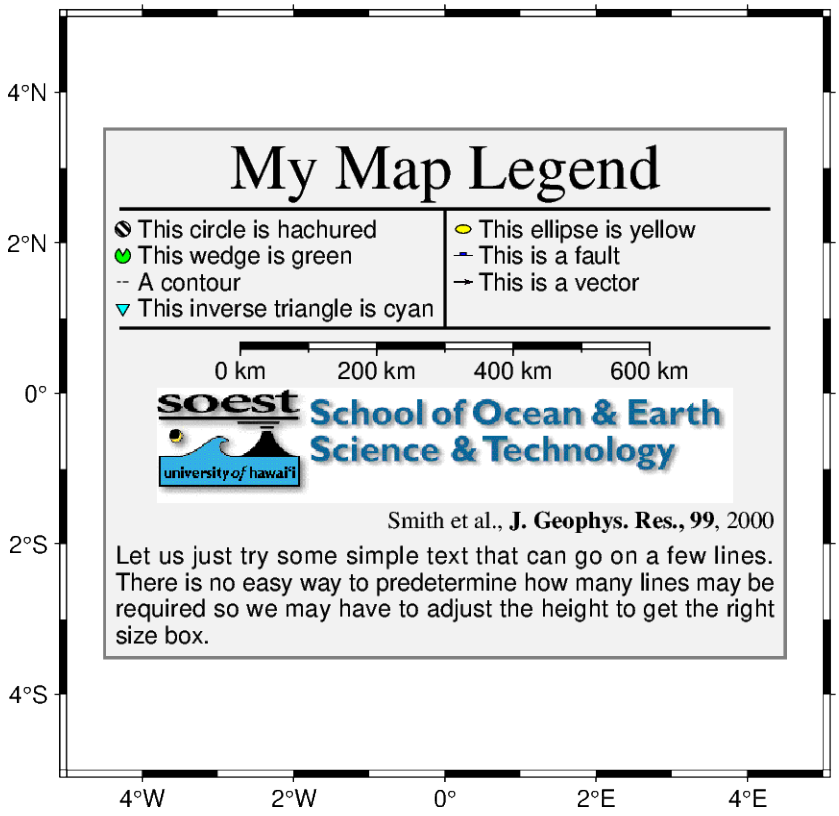
The following example is orientated on the related GMT example at https://docs.generic-mapping-tools.org/6.5/legend.html#examples, but modified to use an `io.StringIO` object.

We start with setting up the `io.StringIO` object.

```
spec_io = io.StringIO(
    """
G -0.1c
H 24p,Times-Roman My Map Legend
D 0.2c 1p
N 2
V 0 1p
S 0.1c c 0.20c p300/12 0.25p 0.3c This circle is hachured
S 0.1c e 0.20c yellow 0.25p 0.3c This ellipse is yellow
S 0.1c w 0.20c green 0.25p 0.3c This wedge is green
S 0.1c f 0.25c blue 0.25p 0.3c This is a fault
S 0.1c - 0.15c - 0.25p,- 0.3c A contour
S 0.1c v 0.25c magenta 0.5p 0.3c This is a vector
S 0.1c i 0.20c cyan 0.25p 0.3c This inverse triangle is cyan
D 0.2c 1p
V 0 1p
N 1
G 0.1c
M 5 5 600+u+f
G 0.1c
I @SOEST_block4.png 3i CT
G 0.05c
L 9p,Times-Roman R Smith et al., @%5%J. Geophys. Res., 99@%%, 2000
G 0.1c
T Let us just try some simple text that can go on a few lines.
T There is no easy way to predetermine how many lines may be required
T so we may have to adjust the height to get the right size box.
"""
)
```

Now, we can add a legend based on this `io.StringIO` object. For multi-columns legends, the width (**+w**) has to be specified via a the `position` parameter.

```
fig = pygmt.Figure()
# Note, that we are now using a Mercator projection
fig.basemap(region=[-5, 5, -5, 5], projection="M10c", frame=True)

# Pass the io.StringIO object to the "spec" parameter
fig.legend(spec=spec_io, position="jMC+w9c", box="+p1p,gray50+ggray95")

fig.show()
```



Total running time of the script: (0 minutes 1.364 seconds)

4.2.7 Draping a dataset on top of a topographic surface

It can be visually appealing to "drape" a dataset over a topographic surface. This can be accomplished using the `drapegrid` parameter of *pygmt.Figure.grdview*.

This tutorial consists of two examples:

1. Draping a grid
2. Draping an image

```
# Load the required packages
import pygmt
import rasterio
import xarray as xr
```

1. Drapping a grid

In the first example, the seafloor crustal age is plotted with color-coding on top of the topographic map of an area of the Mid-Atlantic Ridge.

```
# Define the study area in degrees East or North
region_2d = [-50, 0, 36, 70]  # [lon_min, lon_max, lat_min, lat_max]

# Download elevation and crustal age grids for the study region with a resolution of␣
↪10
# arc-minutes and load them into xarray.DataArrays
```

(continues on next page)

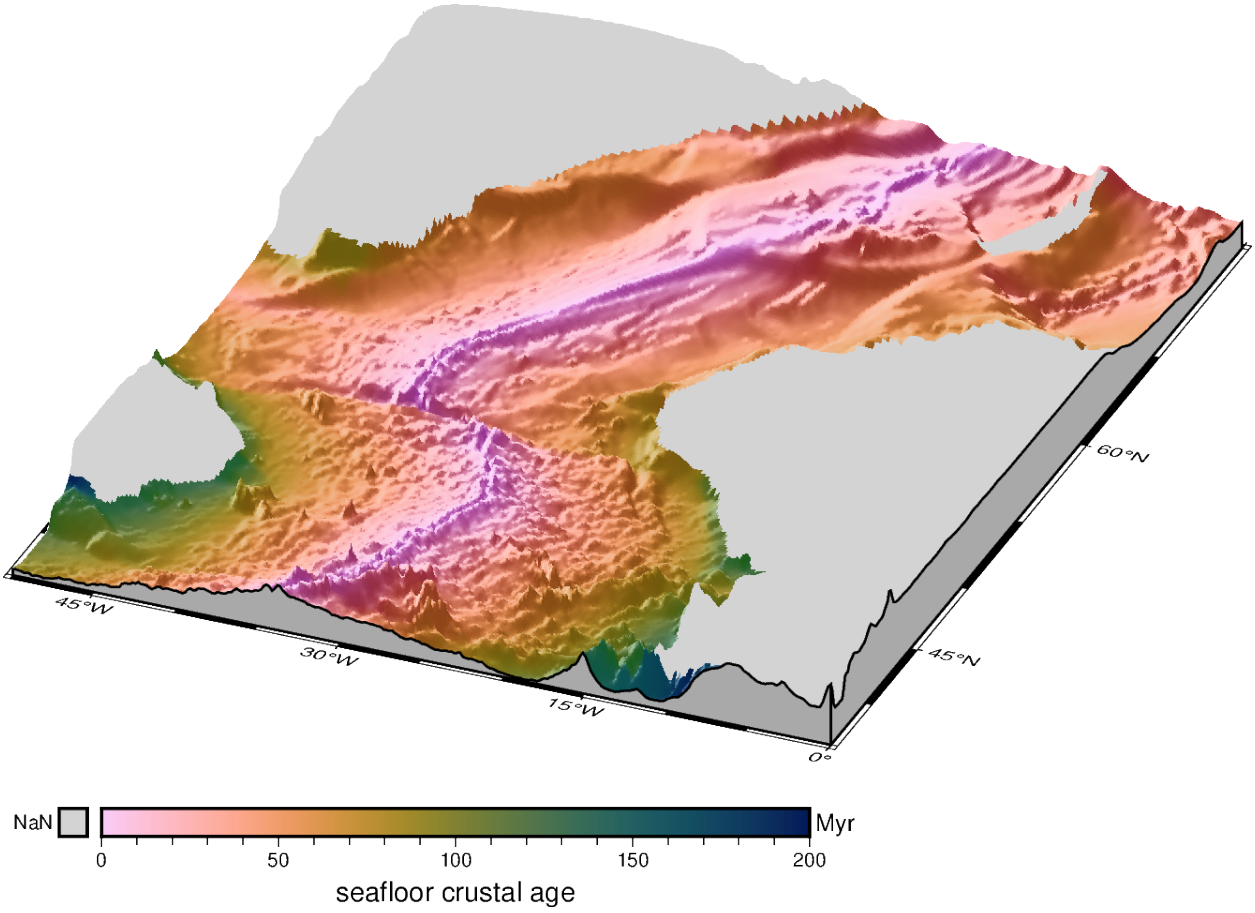
(continued from previous page)

```
grd_relief = pygmt.datasets.load_earth_relief(resolution="10m", region=region_2d)
grd_age = pygmt.datasets.load_earth_age(resolution="10m", region=region_2d)

# Determine the 3-D region from the minimum and maximum values of the relief grid
region_3d = [*region_2d, grd_relief.min().to_numpy(), grd_relief.max().to_numpy()]
```

```
gmtread [NOTICE]: Remote data courtesy of GMT data server oceania [http://oceania.
↪generic-mapping-tools.org]
gmtread [NOTICE]: EarthByte Earth Seafloor Age at 10x10 arc minutes reduced by␣
↪Gaussian Cartesian filtering (52.4 km fullwidth) [Seton et al., 2020].
gmtread [NOTICE]:   -> Download grid file [1.3M]: earth_age_10m_g.grd
```

The topographic surface is created based on the grid passed to the `grid` parameter of *pygmt.Figure.grdview*; here we use a grid of the Earth relief. To add a color-coding based on *another* grid we have to pass a second grid to the `drapegrid` parameter; here we use a grid of the crustal age. In this case the colormap specified via the `cmap` parameter applies to the grid passed to `drapegrid`, not to `grid`. The azimuth and elevation of the 3-D plot are set via the `perspective` parameter.

```
fig = pygmt.Figure()

# Set up colormap for the crustal age
pygmt.config(COLOR_NAN="lightgray")
pygmt.makecpt(cmap="batlow", series=[0, 200, 1], reverse=True, overrule_bg=True)

fig.grdview(
    projection="M12c",  # Mercator projection with a width of 12 centimeters
    region=region_3d,
    grid=grd_relief,  # Use elevation grid for z values
    drapegrid=grd_age,  # Use crustal age grid for color-coding
    cmap=True,  # Use colormap created for the crustal age
    surftype="i",  # Create an image plot
    # Use an illumination from the azimuthal directions 0° (north) and 270°
    # (west) with a normalization via a cumulative Laplace distribution for
    # the shading
    shading="+a0/270+ne0.6",
    perspective=[157.5, 30],  # Azimuth and elevation for the 3-D plot
    zsize="1.5c",
    plane="+gdarkgray",
    frame=True,
)

# Add colorbar for the crustal age
fig.colorbar(frame=["x+lseafloor crustal age", "y+lMyr"], position="+n")

fig.show()
```



2. Draping an image

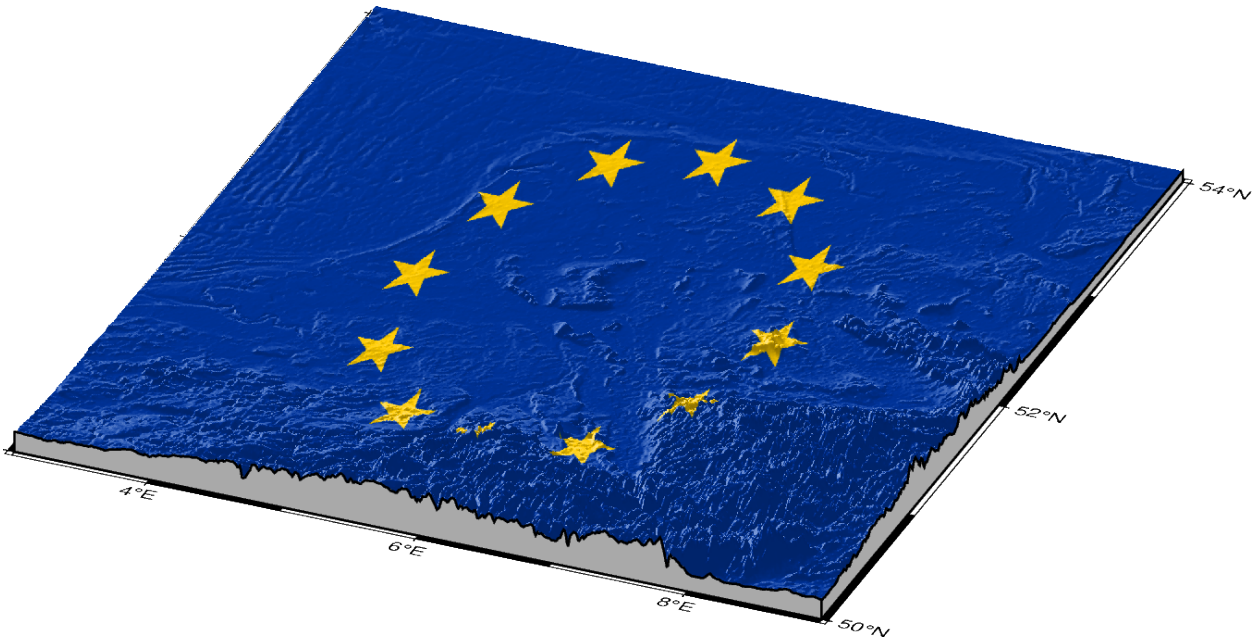
In the second example, the flag of the European Union (EU) is plotted on top of a topographic map of northwest Europe. This example is modified from GMT example 32. We have to consider the dimension of the image we want to drap. The image we will download in this example has 1000 x 667 pixels, i.e. an aspect ratio of 3 x 2.

```
# Define the study area in degrees East or North, with an extend of 6 degrees for
# the longitude and 4 degrees for the latitude
region_2d = [3, 9, 50, 54]  # [lon_min, lon_max, lat_min, lat_max]

# Download elevation grid for the study region with a resolution of 30 arc-seconds and
# pixel registration and load it into an xarray.DataArray
grd_relief = pygmt.datasets.load_earth_relief(resolution="30s", region=region_2d)

# Determine the 3-D region from the minimum and maximum values of the relief grid
region_3d = [*region_2d, grd_relief.min().to_numpy(), grd_relief.max().to_numpy()]

# Download an PNG image of the flag of the EU using rasterio and load it into a
# xarray.DataArray
url_to_image = "https://upload.wikimedia.org/wikipedia/commons/thumb/b/b7/Flag_of_
↪Europe.svg/1000px-Flag_of_Europe.svg.png"
with rasterio.open(url_to_image) as dataset:
    data = dataset.read()
    drapegrid = xr.DataArray(data, dims=("band", "y", "x"))
```

```
grdblend [NOTICE]: Remote data courtesy of GMT data server oceania [http://oceania.
↪generic-mapping-tools.org]
grdblend [NOTICE]: SRTM15 Earth Relief v2.6 at 30x30 arc seconds reduced by Gaussian␣
↪Cartesian filtering (2.6 km fullwidth) [Tozer et al., 2019].
grdblend [NOTICE]:   -> Download 15x15 degree grid tile (earth_relief_30s_g): N45E000
/home/runner/micromamba/envs/pygmt/lib/python3.13/site-packages/rasterio/__init__.
↪py:356: NotGeoreferencedWarning: Dataset has no geotransform, gcps, or rpcs. The␣
↪identity matrix will be returned.
  dataset = DatasetReader(path, driver=driver, sharing=sharing, **kwargs)
```

Again we create a 3-D plot with *pygmt.Figure.grdview* and pass an Earth relief grid to the `grid` parameter to create the topographic surface. But now we pass the PNG image which was loaded into an `xarray.DataArray` to the `drapgrid` parameter.

```
fig = pygmt.Figure()

# Set up a colormap with two colors for the EU flag: blue (0/51/153) for the␣
↪background
# (value 0 in the netCDF file -> lower half of 0-255 range) and yellow (255/204/0) for
# the stars (value 255 -> upper half)
pygmt.makecpt(cmap="0/51/153,255/204/0", series=[0, 256, 128])

fig.grdview(
    projection="M12c",  # Mercator projection with a width of 12 centimeters
    region=region_3d,
    grid=grd_relief,  # Use elevation grid for z values
    drapegrid=drapegrid,  # Drap image grid for the EU flag on top
    cmap=True,  # Use colormap defined for the EU flag
    surftype="i",  # Create an image plot
    # Use an illumination from the azimuthal directions 0° (north) and 270° (west)␣
↪with
    # a normalization via a cumulative Laplace distribution for the shading
    shading="+a0/270+ne0.6",
    perspective=[157.5, 30],  # Define azimuth, elevation for the 3-D plot
    zsize="1c",
    plane="+gdarkgray",
    frame=True,
)

fig.show()
```




```
/home/runner/micromamba/envs/pygmt/lib/python3.13/site-packages/rasterio/__init__.
↪py:366: NotGeoreferencedWarning: The given matrix is equal to Affine.identity or␣
↪its flipped counterpart. GDAL may ignore this matrix and save no geotransform␣
↪without raising an error. This behavior is somewhat driver-specific.
  dataset = writer(
```

Total running time of the script: (0 minutes 6.668 seconds)

4.2.8 Interactive data visualization using `Panel`

> **Note**
>
> Please run the following code examples in a notebook environment otherwise the interactive parts of this tutorial will not work. You can use the button "Download Jupyter notebook" at the bottom of this page to download this script as a Jupyter notebook.

The library Panel can be used to create interactive dashboards by connecting user-defined widgets to plots. `Panel` can be used as an extension to Jupyter notebook/lab.

This tutorial is split into three parts:

- Make a static map
- Make an interactive map
- Add a grid for Earth relief

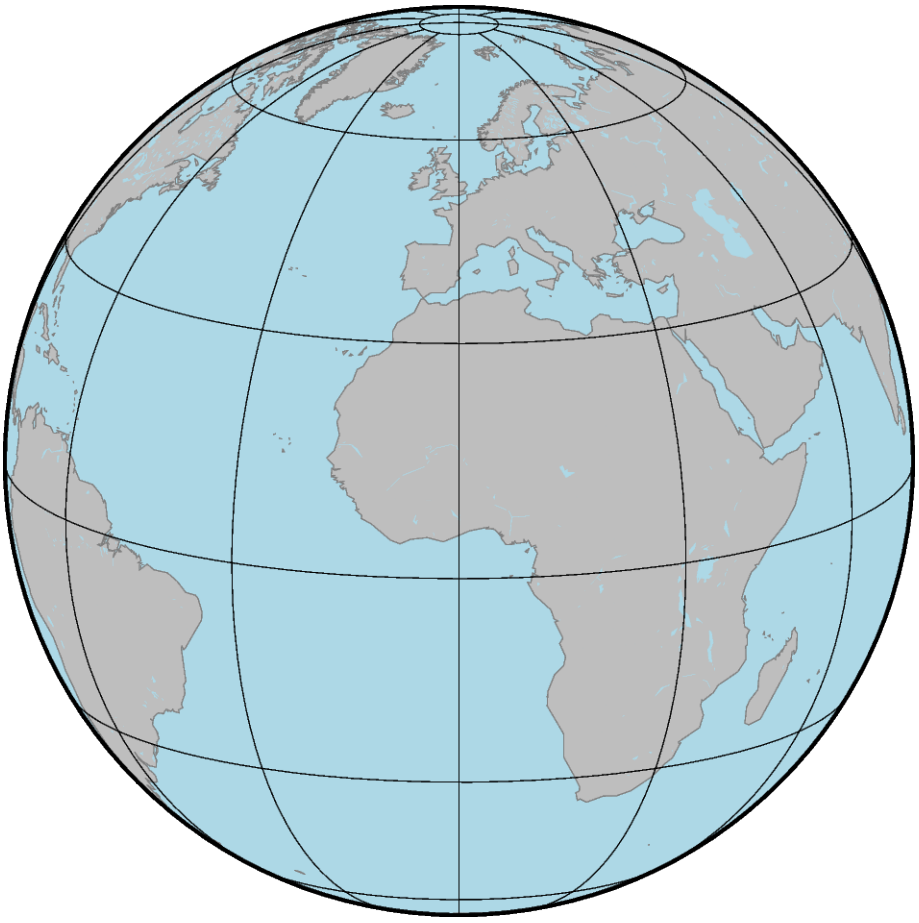
Import the required packages

```
import numpy as np
import panel as pn
import pygmt

pn.extension()
```

Make a static map

The Orthographic projection can be used to show the Earth as a globe. Land and water masses are filled with colors via the `land` and `water` parameters of *pygmt.Figure.coast*, respectively. Coastlines are added using the `shorelines` parameter.

```
# Create a new instance or object of the pygmt.Figure() class
fig = pygmt.Figure()
fig.coast(
    # Orthographic projection (G) with projection center at 0° East and
    # 15° North and a width of 12 centimeters
    projection="G0/15/12c",
    region="g",  # global
    frame="g30",  # Add frame and gridlines in steps of 30 degrees on top
    land="gray",  # Color land masses in "gray"
    water="lightblue",  # Color water masses in "lightblue"
    # Add coastlines with a 0.25 points thick pen in "gray50"
    shorelines="1/0.25p,gray50",
)
fig.show()
```



Make an interactive map

To generate a rotation of the Earth around the vertical axis, the central longitude of the Orthographic projection is varied iteratively in steps of 10 degrees. The library `Panel` is used to create an interactive dashboard with a slider (works only in a notebook environment, e.g., Jupyter notebook).

```
# Create a slider
slider_lon = pn.widgets.DiscreteSlider(
    name="Central longitude",  # Give name for quantity shown at the slider
    options=list(np.arange(0, 361, 10)),  # Range corresponding to longitude
    value=0,  # Set start value
)


@pn.depends(central_lon=slider_lon)
def view(central_lon):
    """
    Define a function for plotting the single slices.
    """
    # Create a new instance or object of the pygmt.Figure() class
    fig = pygmt.Figure()
    fig.coast(
        # Vary the central longitude used for the Orthographic projection
        projection=f"G{central_lon}/15/12c",
        region="g",
        frame="g30",
        land="gray",
        water="lightblue",
        shorelines="1/0.25p,gray50",
    )
    return fig


# Make an interactive dashboard
pn.Column(slider_lon, view)
```

```
Column
    [0] DiscreteSlider(formatter='%d', name='Central longitude', options=[np.int64(0),
↪ ...], value=0)
    [1] ParamFunction(function, _pane=PNG, defer_load=False)
```

Add a grid for Earth relief

Instead of using colors as fill for the land and water masses a grid can be displayed. Here, the Earth relief is shown by color-coding the elevation.

```
# Download a grid for Earth relief with a resolution of 10 arc-minutes
grd_relief = pygmt.datasets.load_earth_relief(resolution="10m")

# Create a slider
slider_lon = pn.widgets.DiscreteSlider(
    name="Central longitude",
    options=list(np.arange(0, 361, 10)),
    value=0,
```

(continues on next page)

(continued from previous page)

```
)


@pn.depends(central_lon=slider_lon)
def view(central_lon):
    """
    Define a function for plotting the single slices.
    """
    # Create a new instance or object of the pygmt.Figure() class
    fig = pygmt.Figure()
    # Set up a colormap for the elevation in meters
    pygmt.makecpt(
        cmap="oleron",
        # minimum, maximum, step
        series=[int(grd_relief.data.min()) - 1, int(grd_relief.data.max()) + 1, 100],
    )
    # Plot the grid for the elevation
    fig.grdimage(
        projection=f"G{central_lon}/15/12c",
        region="g",
        grid=grd_relief,  # Use grid downloaded above
        cmap=True,  # Use colormap defined above
        frame="g30",
    )
    # Add a horizontal colorbar for the elevation
    # with annotations (a) in steps of 2000 and ticks (f) in steps of 1000
    # and labels (+l) at the x-axis "Elevation" and y-axis "m" (meters)
    fig.colorbar(frame=["a2000f1000", "x+lElevation", "y+lm"])
    return fig


# Make an interactive dashboard
pn.Column(slider_lon, view)
```

```
Column
    [0] DiscreteSlider(formatter='%d', name='Central longitude', options=[np.int64(0),
↪ ...], value=0)
    [1] ParamFunction(function, _pane=PNG, defer_load=False)
```

Total running time of the script: (0 minutes 1.687 seconds)

4.2.9 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()
```

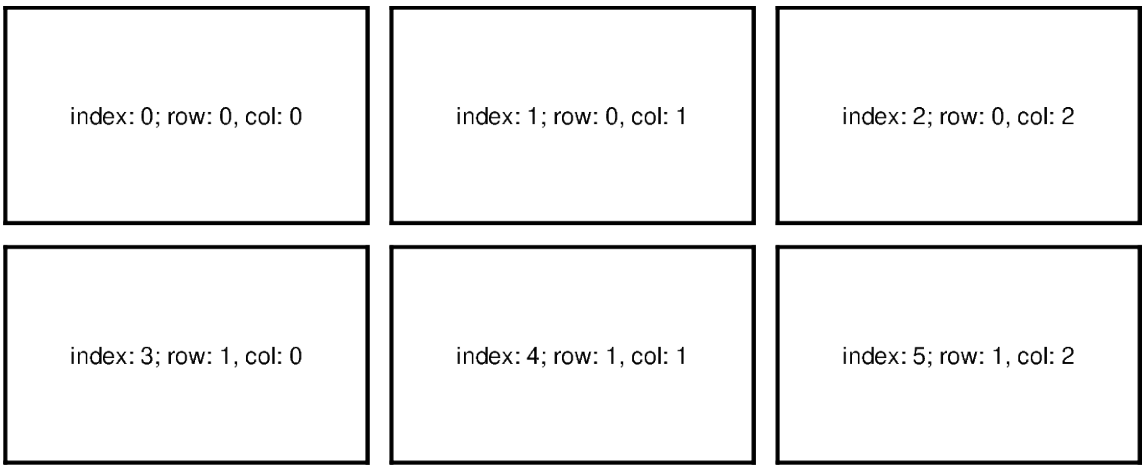
Define subplot layout

The *pygmt.Figure.subplot* method 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()
```



The *pygmt.Figure.set_panel* method activates a specified subplot, and all subsequent plotting methods 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.

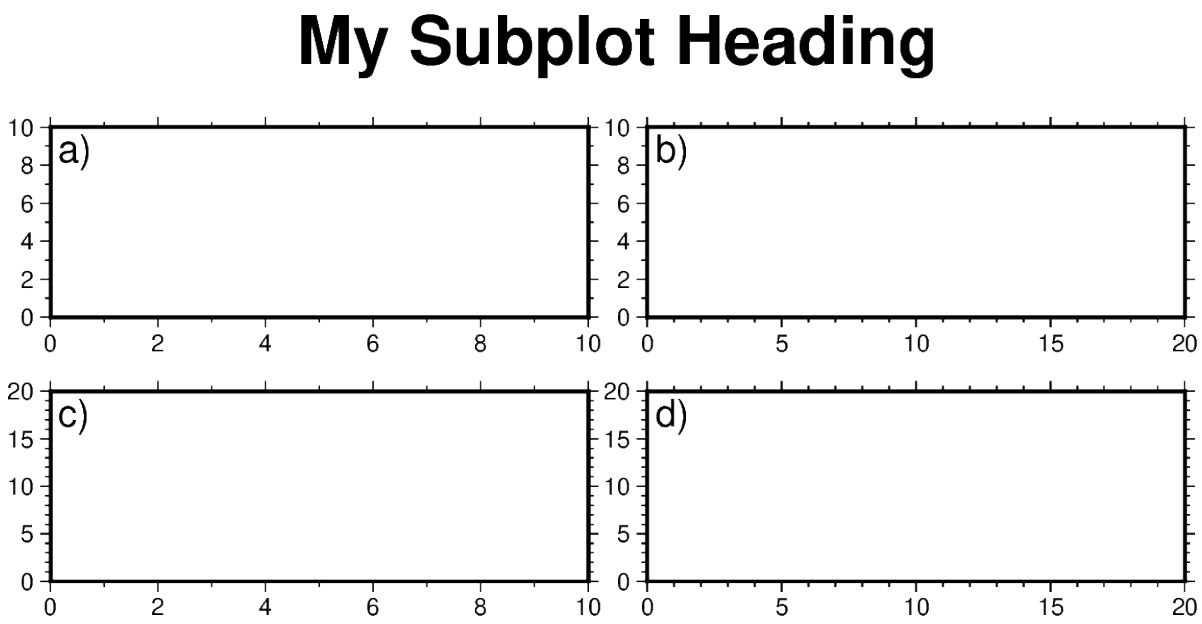
For example, to activate the subplot on the top right corner (index: 2) at *row*=0 and *col*=2, so that all subsequent plotting commands happen there, you can use the following command:

```
with fig.set_panel(panel=[0, 2]):
    ...
```

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()
```



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 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.

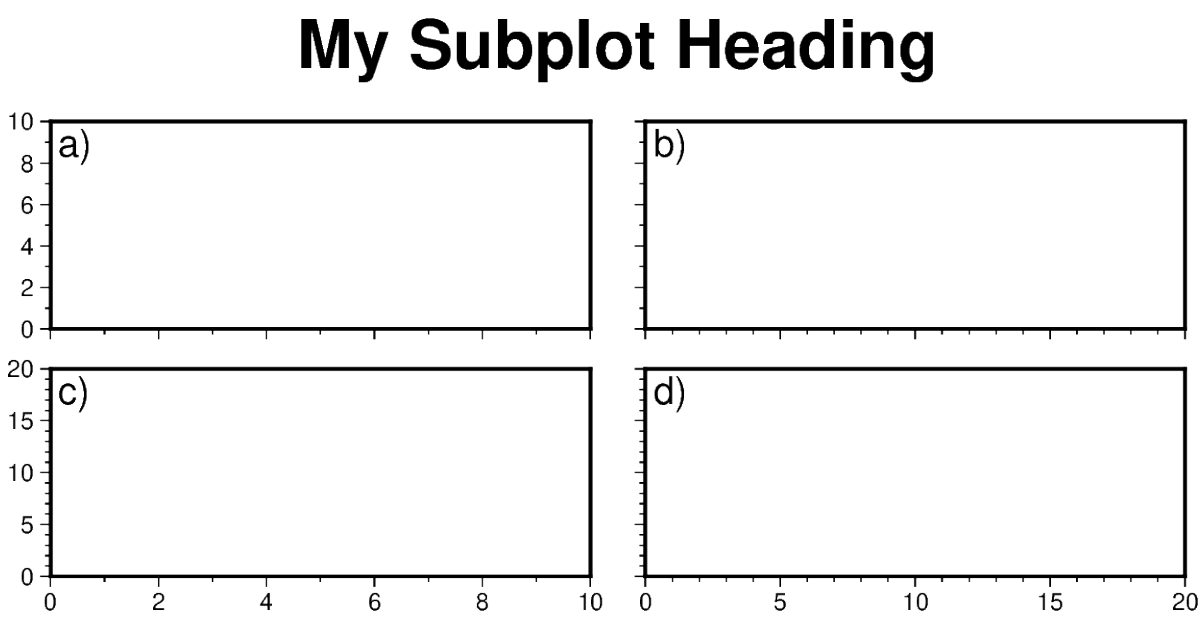
Note

All plotting methods (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.

Shared x- and y-axes

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()
```



`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.

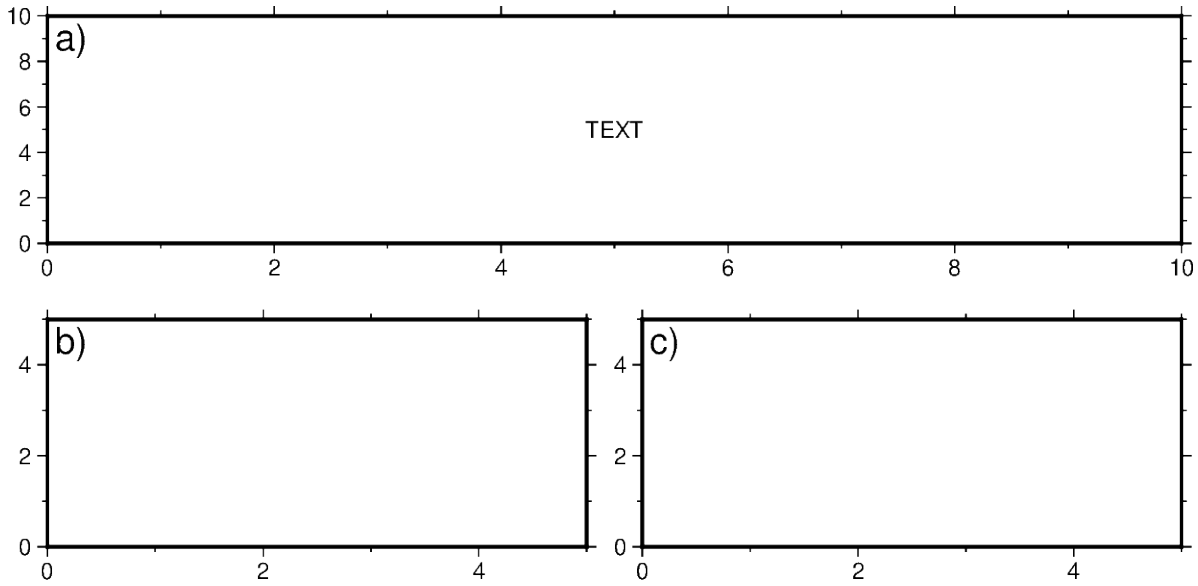
Of course, instead of using the `sharex`/`sharey` parameters, you can also set a different `frame` for each subplot to control the axis properties individually for each subplot.

Advanced subplot layouts

Nested subplots 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.

```
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)

fig.show()
```



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 0.594 seconds)

4.2.10 Performing grid histogram equalization

The *pygmt.grdhisteq.equalize_grid* method creates a grid using statistics based on a cumulative distribution function.

```
import pygmt
```

Load sample data

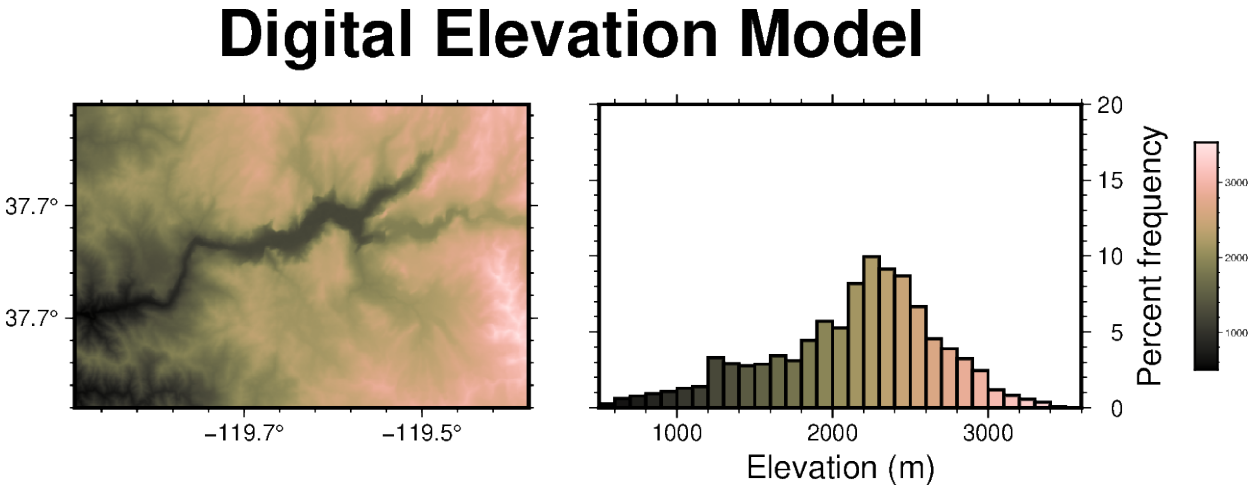
Load the sample Earth relief data for a region around Yosemite Valley and use *pygmt.grd2xyz* to create a `pandas.Series` with the z-values.

```
grid = pygmt.datasets.load_earth_relief(
    resolution="03s", region=[-119.825, -119.4, 37.6, 37.825]
)
grid_dist = pygmt.grd2xyz(grid=grid, output_type="pandas")["z"]
```

Plot the original digital elevation model and data distribution

For comparison, we will create a map of the original digital elevation model and a histogram showing the distribution of elevation data values.

```
# Create an instance of the Figure class
fig = pygmt.Figure()
# Define figure configuration
pygmt.config(FORMAT_GEO_MAP="ddd.x", MAP_FRAME_TYPE="plain")
# Define the colormap for the figure
pygmt.makecpt(series=[500, 3540], cmap="turku")
# Setup subplots with two panels
with fig.subplot(
    nrows=1, ncols=2, figsize=("13.5c", "4c"), title="Digital Elevation Model"
):
    # Plot the original digital elevation model in the first panel
    with fig.set_panel(panel=0):
        fig.grdimage(grid=grid, projection="M?", frame="WSne", cmap=True)
    # Plot a histogram showing the z-value distribution in the original digital
    # elevation model
    with fig.set_panel(panel=1):
        fig.histogram(
            data=grid_dist,
            projection="X?",
            region=[500, 3600, 0, 20],
            series=[500, 3600, 100],
            frame=["wnSE", "xaf+lElevation (m)", "yaf+lPercent frequency"],
            cmap=True,
            histtype=1,
            pen="1p,black",
        )
        fig.colorbar(position="JMR+o1.5c/0c+w3c/0.3c", frame=True)
fig.show()
```



Equalize grid based on a linear distribution

The *pygmt.grdhisteq.equalize_grid* method creates a new grid with the z-values representing the position of the original z-values in a given cumulative distribution. By default, it computes the position in a linear distribution. Here, we equalize the grid into nine divisions based on a linear distribution and produce a `pandas.Series` with the z-values for the new grid.

```
divisions = 9
linear = pygmt.grdhisteq.equalize_grid(grid=grid, divisions=divisions)
linear_dist = pygmt.grd2xyz(grid=linear, output_type="pandas")["z"]
```

Calculate the bins used for data transformation

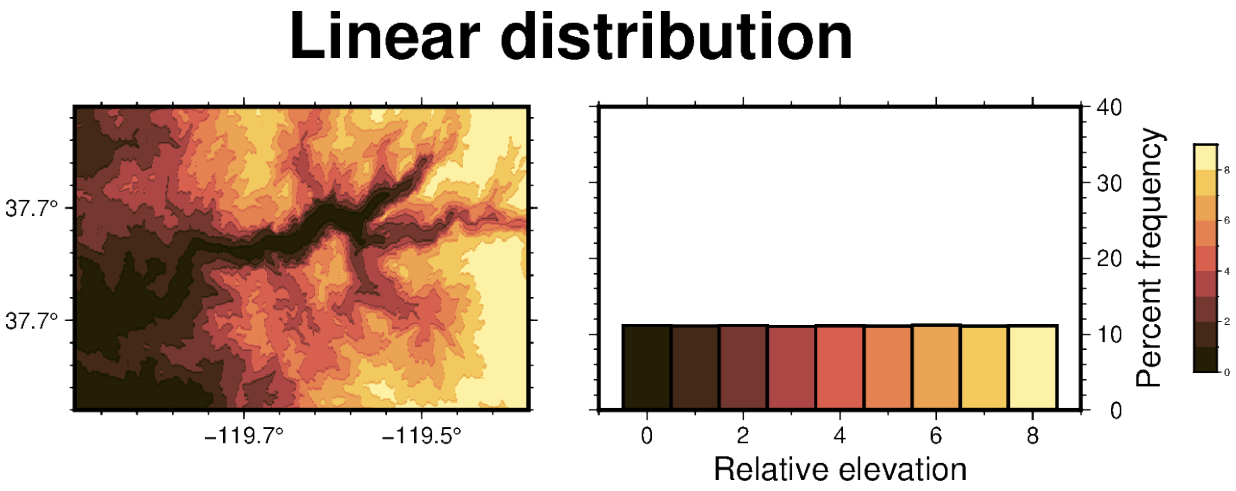
The *pygmt.grdhisteq.compute_bins* method reports statistics about the grid equalization. Here, we report the bins that would linearly divide the original data into 9 divisions with equal area. In our new grid produced by *pygmt.grdhisteq.equalize_grid*, all the grid cells with values between `start` and `stop` of `bin_id=0` are assigned the value 0, all grid cells with values between `start` and `stop` of `bin_id=1` are assigned the value 1, and so on.

```
pygmt.grdhisteq.compute_bins(grid=grid, divisions=divisions)
```

Plot the equally distributed data

Here we create a map showing the grid that has been transformed to have a linear distribution with nine divisions and a histogram of the data values.

```
# Create an instance of the Figure class
fig = pygmt.Figure()
# Define figure configuration
pygmt.config(FORMAT_GEO_MAP="ddd.x", MAP_FRAME_TYPE="plain")
# Define the colormap for the figure
pygmt.makecpt(series=[0, divisions, 1], cmap="lajolla")
# Setup subplots with two panels
with fig.subplot(
    nrows=1, ncols=2, figsize=("13.5c", "4c"), title="Linear distribution"
):
    # Plot the grid with a linear distribution in the first panel
    with fig.set_panel(panel=0):
        fig.grdimage(grid=linear, projection="M?", frame="WSne", cmap=True)
    # Plot a histogram showing the linear z-value distribution
    with fig.set_panel(panel=1):
        fig.histogram(
            data=linear_dist,
            projection="X?",
            region=[-1, divisions, 0, 40],
            series=[0, divisions, 1],
            frame=["wnSE", "xaf+lRelative elevation", "yaf+lPercent frequency"],
            cmap=True,
            histtype=1,
            pen="1p,black",
            center=True,
        )
        fig.colorbar(position="JMR+o1.5c/0c+w3c/0.3c", frame=True)
fig.show()
```



Transform grid based on a normal distribution

The `gaussian` parameter of *pygmt.grdhisteq.equalize_grid* can be used to transform the z-values relative to their position in a normal distribution rather than a linear distribution. In this case, the output data are continuous rather than discrete.

```
normal = pygmt.grdhisteq.equalize_grid(grid=grid, gaussian=True)
normal_dist = pygmt.grd2xyz(grid=normal, output_type="pandas")["z"]
```

Plot the normally distributed data

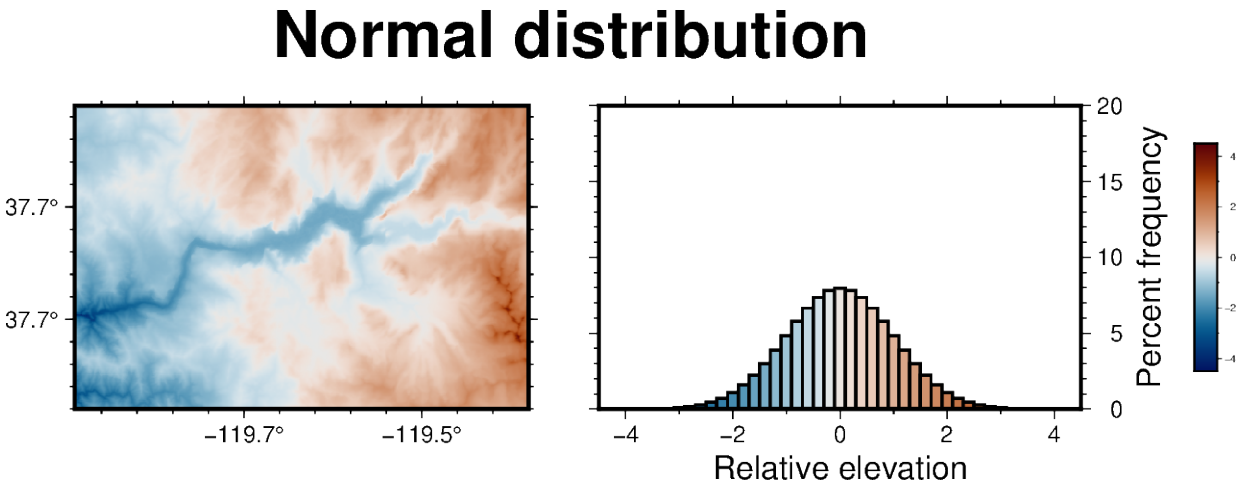
Here we create a map showing the grid that has been transformed to have a normal distribution and a histogram of the data values.

```
# Create an instance of the Figure class
fig = pygmt.Figure()
# Define figure configuration
pygmt.config(FORMAT_GEO_MAP="ddd.x", MAP_FRAME_TYPE="plain")
# Define the colormap for the figure
pygmt.makecpt(series=[-4.5, 4.5], cmap="vik")
# Setup subplots with two panels
with fig.subplot(
    nrows=1, ncols=2, figsize=("13.5c", "4c"), title="Normal distribution"
):
    # Plot the grid with a normal distribution in the first panel
    with fig.set_panel(panel=0):
        fig.grdimage(grid=normal, projection="M?", frame="WSne", cmap=True)
    # Plot a histogram showing the normal z-value distribution
    with fig.set_panel(panel=1):
        fig.histogram(
            data=normal_dist,
            projection="X?",
            region=[-4.5, 4.5, 0, 20],
            series=[-4.5, 4.5, 0.2],
            frame=["wnSE", "xaf+lRelative elevation", "yaf+lPercent frequency"],
            cmap=True,
            histtype=1,
            pen="1p,black",
```

(continues on next page)

(continued from previous page)

```
        )
        fig.colorbar(position="JMR+o1.5c/0c+w3c/0.3c", frame=True)
fig.show()
```



Equalize grid based on a quadratic distribution

The `quadratic` parameter of *pygmt.grdhisteq.equalize_grid* can be used to transform the z-values relative to their position in a quadratic distribution rather than a linear distribution. Here, we equalize the grid into nine divisions based on a quadratic distribution and produce a `pandas.Series` with the z-values for the new grid.

```
quadratic = pygmt.grdhisteq.equalize_grid(
    grid=grid, quadratic=True, divisions=divisions
)
quadratic_dist = pygmt.grd2xyz(grid=quadratic, output_type="pandas")["z"]
```

Calculate the bins used for data transformation

We can also use the `quadratic` parameter of *pygmt.grdhisteq.compute_bins* to report the bins used for dividing the grid into 9 divisions based on their position in a quadratic distribution.

```
pygmt.grdhisteq.compute_bins(grid=grid, divisions=divisions, quadratic=True)
```

Plot the quadratic distribution of data

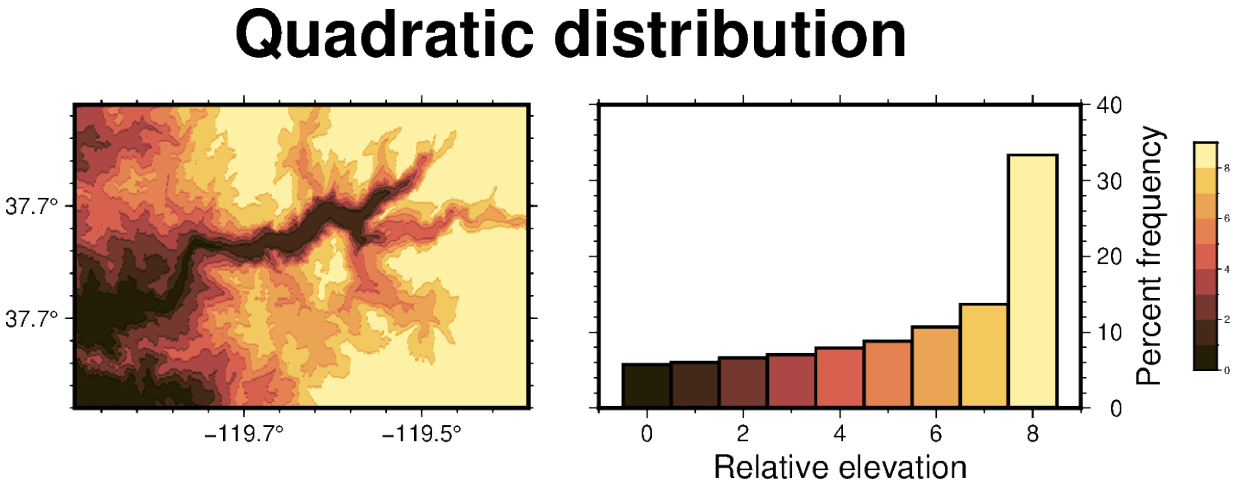
Here we create a map showing the grid that has been transformed to have a quadratic distribution and a histogram of the data values.

```
# Create an instance of the Figure class
fig = pygmt.Figure()
# Define figure configuration
pygmt.config(FORMAT_GEO_MAP="ddd.x", MAP_FRAME_TYPE="plain")
# Define the colormap for the figure
pygmt.makecpt(series=[0, divisions, 1], cmap="lajolla")
# Setup subplots with two panels
with fig.subplot(
    nrows=1, ncols=2, figsize=("13.5c", "4c"), title="Quadratic distribution"
```

(continues on next page)

(continued from previous page)

```
):
    # Plot the grid with a quadratic distribution in the first panel
    with fig.set_panel(panel=0):
        fig.grdimage(grid=quadratic, projection="M?", frame="WSne", cmap=True)
    # Plot a histogram showing the quadratic z-value distribution
    with fig.set_panel(panel=1):
        fig.histogram(
            data=quadratic_dist,
            projection="X?",
            region=[-1, divisions, 0, 40],
            series=[0, divisions, 1],
            frame=["wnSE", "xaf+lRelative elevation", "yaf+lPercent frequency"],
            cmap=True,
            histtype=1,
            pen="1p,black",
            center=True,
        )
        fig.colorbar(position="JMR+o1.5c/0c+w3c/0.3c", frame=True)
fig.show()
```



Total running time of the script: (0 minutes 1.255 seconds)

4.2.11 Plotting Earth relief

PyGMT provides the *pygmt.datasets.load_earth_relief* function to download the Earth relief data from the GMT remote server and load as an `xarray.DataArray` object. The data can then be plotted using the *pygmt.Figure.grdimage* method.

```
import pygmt
```

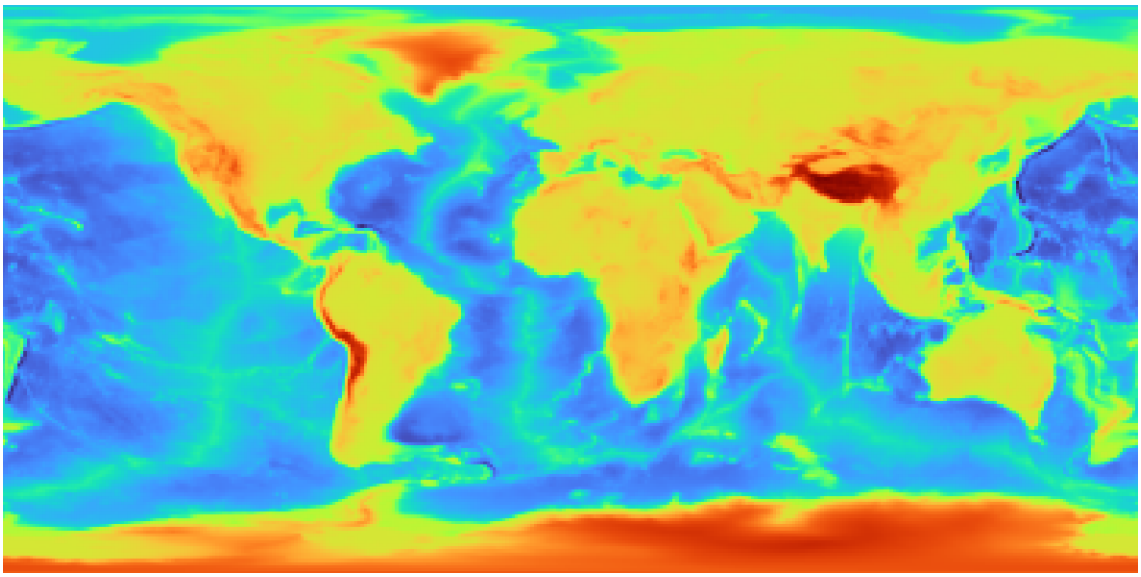
Load sample Earth relief data for the entire globe at a resolution of 1 arc-degree. Refer to *pygmt.datasets.load_earth_relief* for the other available resolutions.

```
grid = pygmt.datasets.load_earth_relief(resolution="01d")
```

Create a plot

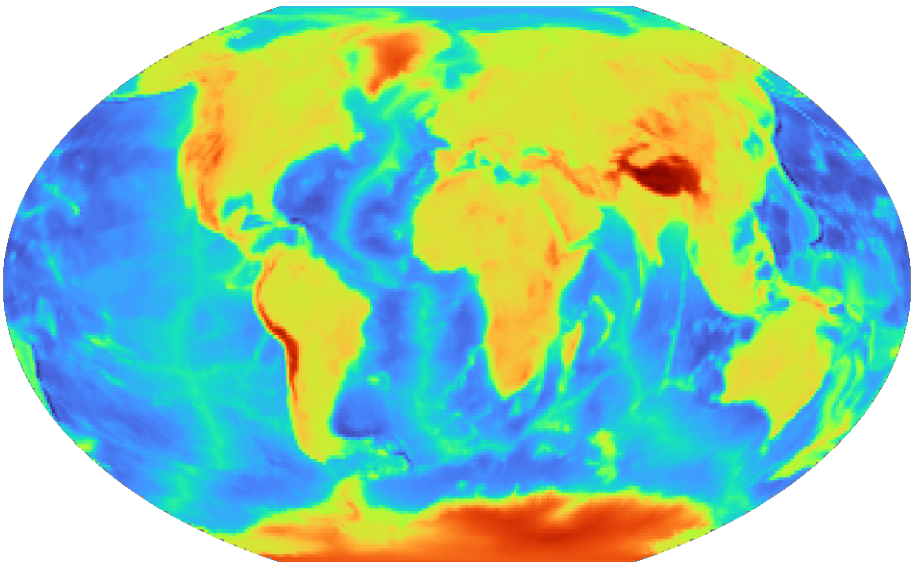
The *pygmt.Figure.grdimage* method takes the `grid` input to create a figure. It creates and applies a color palette to the figure based upon the z-values of the data. By default, it plots the map with the *turbo* CPT, an equidistant cylindrical projection, and with no frame.

```
fig = pygmt.Figure()
fig.grdimage(grid=grid)
fig.show()
```



pygmt.Figure.grdimage can take the optional parameter `projection` for the map. In the example below, `projection` is set to `"R12c"` for a 12-centimeters-wide figure with a Winkel Tripel projection. For a list of available projections, see *GMT Map Projections*.

```
fig = pygmt.Figure()
fig.grdimage(grid=grid, projection="R12c")
fig.show()
```

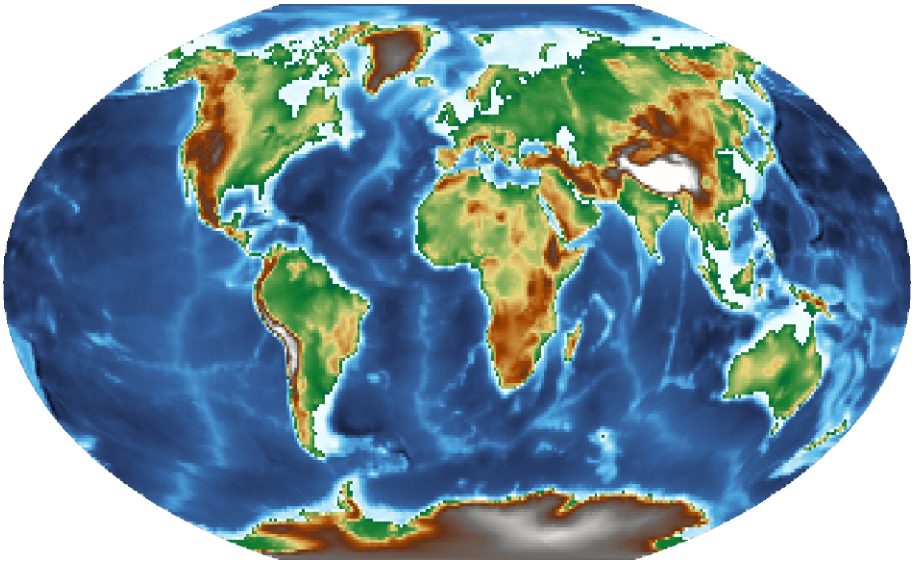


Set a color map

pygmt.Figure.grdimage takes the `cmap` parameter 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/6.5/reference/cpts.html.

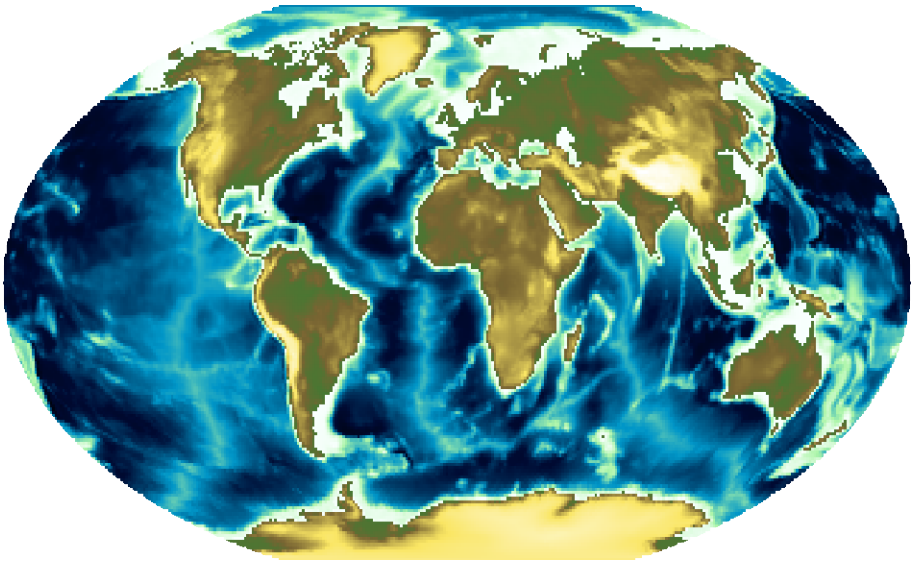
Using the *geo* CPT:

```
fig = pygmt.Figure()
fig.grdimage(grid=grid, projection="R12c", cmap="geo")
fig.show()
```



Using the *relief* CPT:

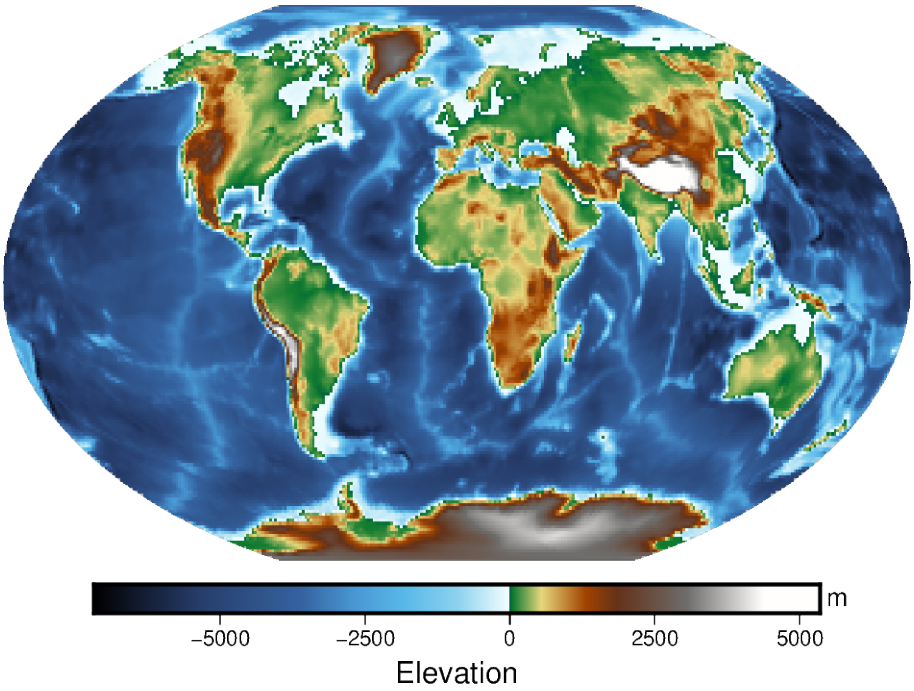
```
fig = pygmt.Figure()
fig.grdimage(grid=grid, projection="R12c", cmap="relief")
fig.show()
```



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` parameter for *pygmt.Figure.grdimage*. The `frame` parameter 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()
```

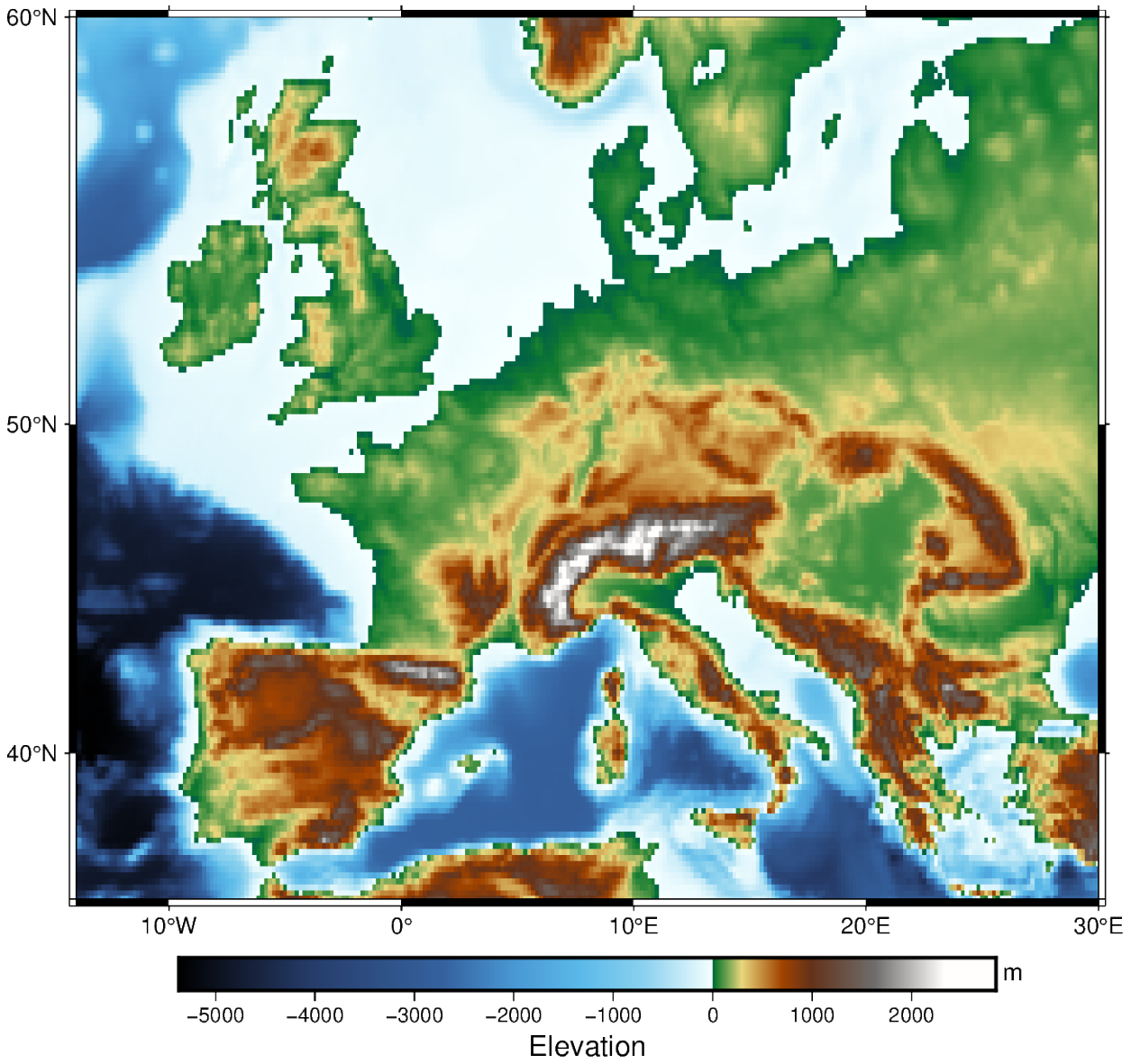


Create a region map

In addition to providing global data, the `region` parameter of *pygmt.datasets.load_earth_relief* can be used to provide data for a specific area. The `region` parameter is required for resolutions at 5 arc-minutes or higher, and accepts a list in the form of [*xmin*, *xmax*, *ymin*, *ymax*].

The example below uses data with a 10 arc-minutes resolution, and plots it on a 15-centimeters-wide figure with a Mercator projection and a CPT set to *geo*. `frame="a"` is used to add a frame with annotations 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()
```



Total running time of the script: (0 minutes 1.205 seconds)

4.2.12 Plotting datetime charts

PyGMT accepts a variety of datetime objects to plot data and create charts. Aside from the built-in Python `datetime` module, PyGMT supports inputs containing ISO formatted strings as well as objects generated with `numpy`, `pandas`, and `xarray`. 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.

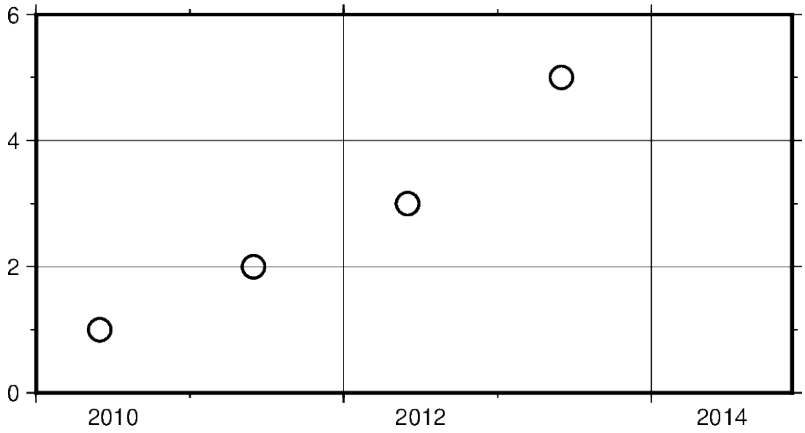
The following examples will demonstrate how to create plots using these different datetime objects.

```
import datetime

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

Using Python's `datetime`

In this example, Python's built-in `datetime` module is used to create data points stored in the 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 plotted as circles with a diameter of 0.3 centimeters.

```
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="c0.3c",
    pen="1p",
)
fig.show()
```



In addition to specifying the date, `datetime` supports the 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:

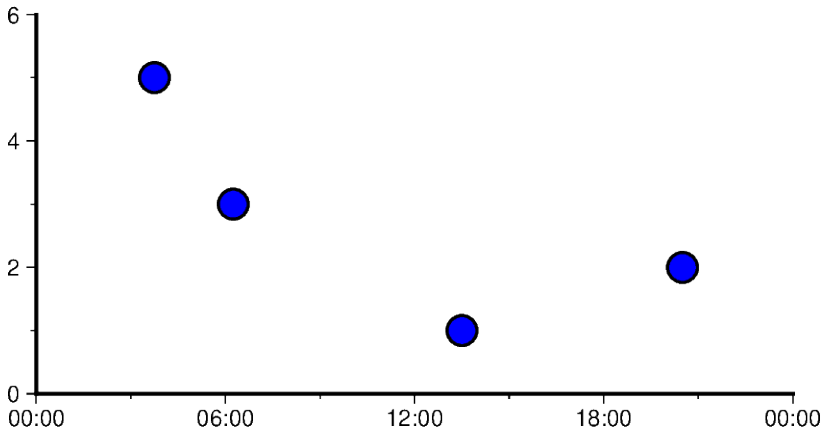
- Modifying `frame` to only include West (left) and South (bottom) borders, and removing grid lines
- Using circles to plot data points defined by `c` in the argument passed through the `style` parameter

```
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),
]
```

(continues on next page)

(continued from previous page)

```
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",
    fill="blue",
)
fig.show()
```



Using ISO Format

In addition to Python's `datetime` module, PyGMT also supports passing dates 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.

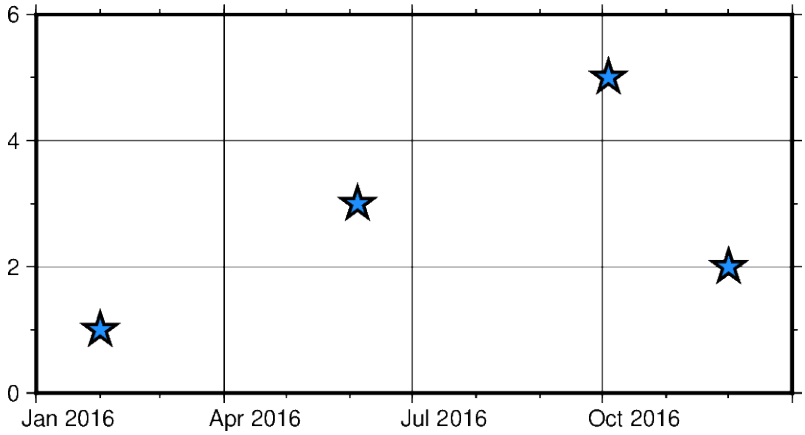
For including the time into an ISO string, the `T` character is used, as it can be seen in the following example. This character is immediately followed by a string formatted as `hh:mm:ss` where each `:` delineated section marking the two-digit hour value, two-digit minute value, and two-digit second value, respectively. The figure in the following example is plotted over a horizontal range of one year from 2016-01-01 to 2017-01-01.

```
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", "2017-01-01", 0, 6],
    frame=["WSen", "afg"],
```

(continues on next page)

(continued from previous page)

```
    x=x,
    y=y,
    style="a0.45c",
    pen="1p",
    fill="dodgerblue",
)
fig.show()
```



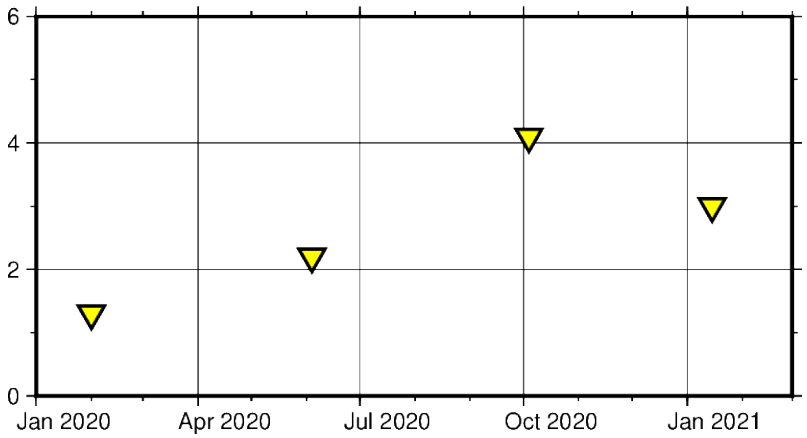
Note

PyGMT doesn't recognize non-ISO datetime strings like "Jun 05, 2018". If your data contain non-ISO datetime strings, you can convert them to a recognized format using `pandas.to_datetime` and then pass it to PyGMT.

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.

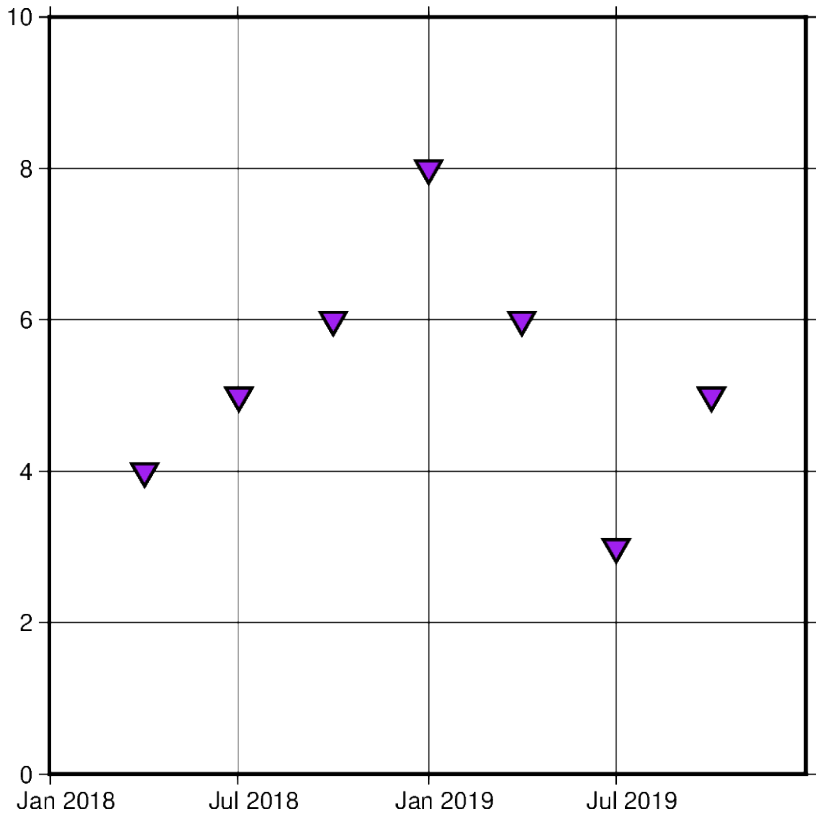
```
x = ["2020-02-01", "2020-06-04", "2020-10-04", datetime.datetime(2021, 1, 15)]
y = [1.3, 2.2, 4.1, 3]

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",
    fill="yellow",
)
fig.show()
```



Using `pandas.date_range`

In the following example, `pandas.date_range` produces a list of `pandas.DatetimeIndex` objects, which 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 `freq` 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.

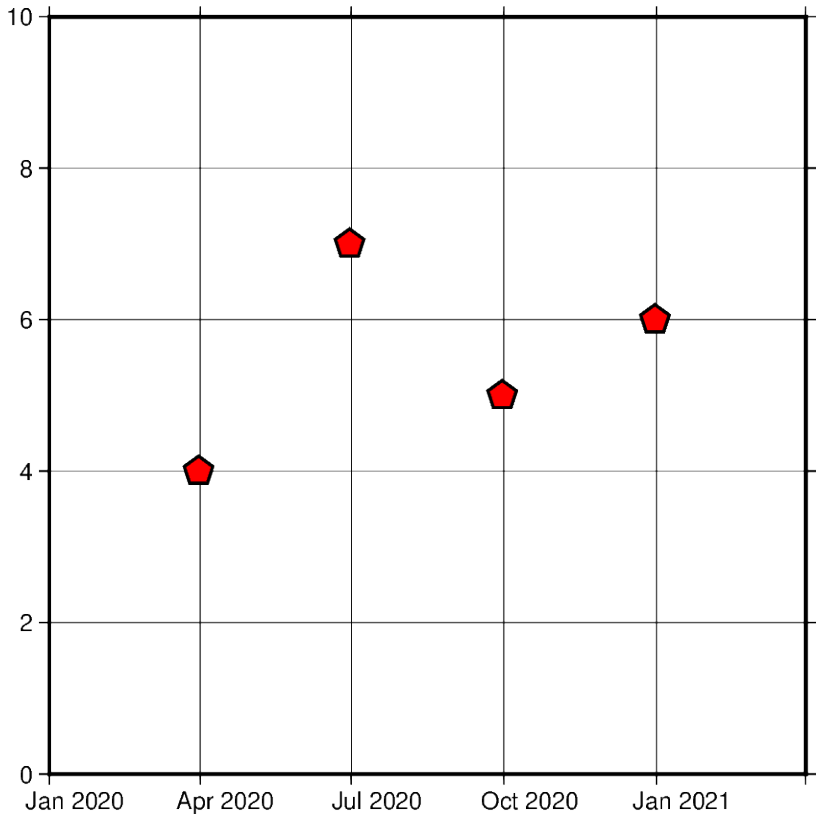
```
x = pd.date_range("2018-03-01", periods=7, freq="BQS")
y = [4, 5, 6, 8, 6, 3, 5]

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",
    fill="purple",
)
fig.show()
```



Using `xarray.DataArray`

In this example, instead of using a list of `pandas.DatetimeIndex` objects, `x` is initialized as an `xarray.DataArray` object. This object provides a wrapper around numpy ndarrays. 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` to fill the DataArray with data, but this is not essential for the creation of a valid DataArray.

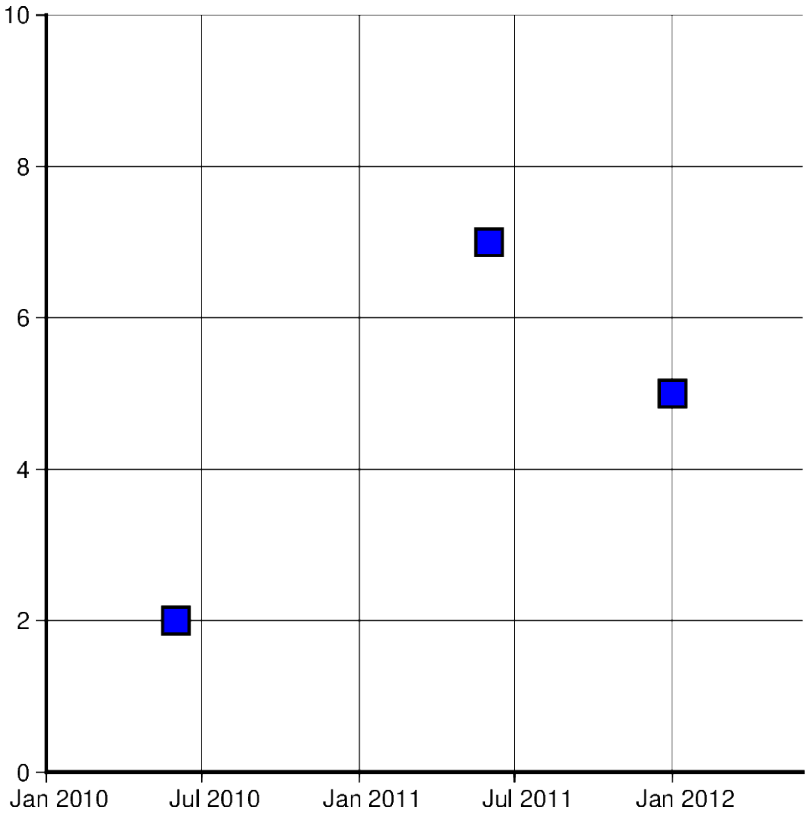
```
x = xr.DataArray(data=pd.date_range(start="2020-01-01", periods=4, freq="QE"))
y = [4, 7, 5, 6]

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",
    fill="red",
)
fig.show()
```



Using `numpy.datetime64`

In this example, instead of using `pd.date_range`, `x` is initialized as an `np.array` object. Similar to `xarray.DataArray` this wraps the dataset before passing it as an argument. However, `np.array` objects use less memory and allow developers to specify data types.

```
x = np.array(
    ["2010-06-01", "2011-06-01T12", "2012-01-01T12:34:56"], dtype=np.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",
    fill="blue",
)
fig.show()
```



Generating an automatic region

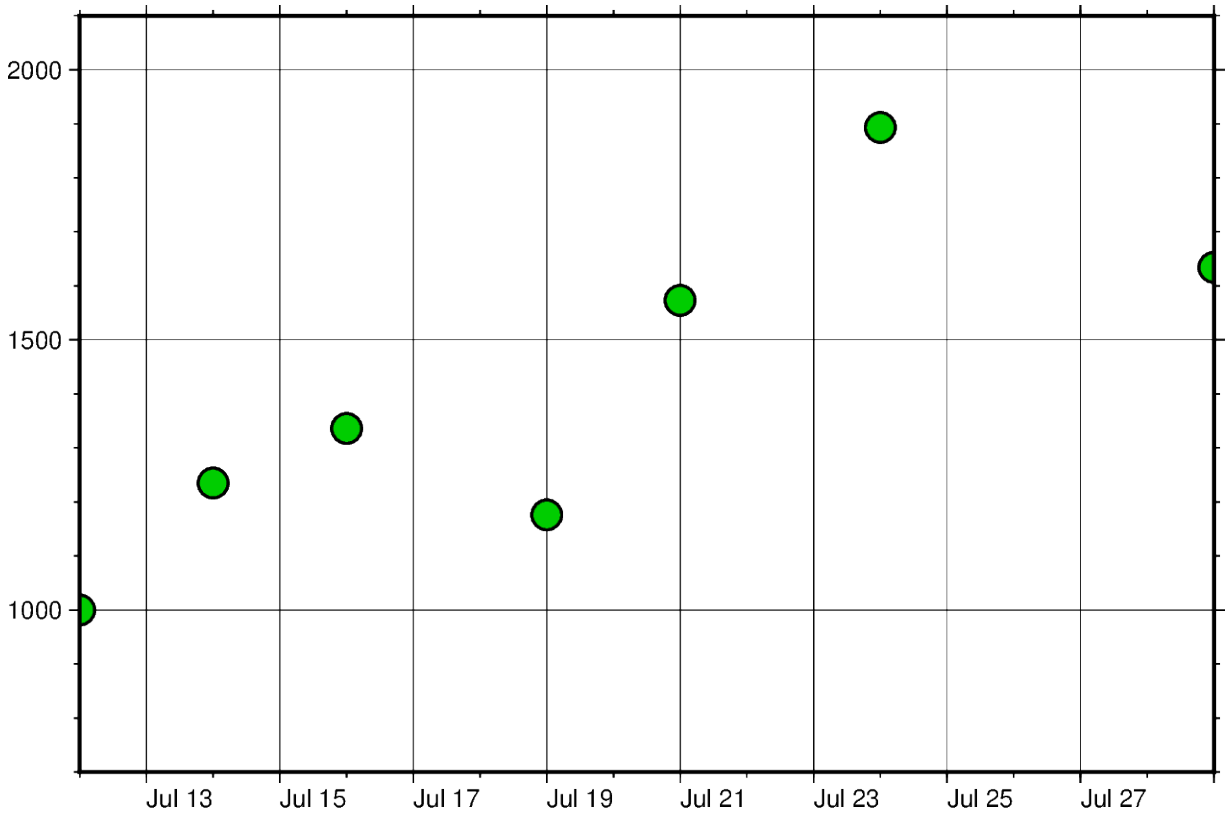
Another way of creating charts involving datetime data can be done by automatically generating the region of the plot. This can be done by passing the DataFrame to *pygmt.info*, which will find the maximum and minimum values for each column and create a list that could be passed as region. Additionally, the `spacing` parameter can be used 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")
region = pygmt.info(
    data=df[["Date", "Score"]], per_column=True, spacing=(700, 700), coltypes="T"
)

fig = pygmt.Figure()
fig.plot(
    region=region,
    projection="X15c/10c",
    frame=["WSen", "afg"],
```

(continues on next page)

(continued from previous page)

```
    x=df.Date,
    y=df.Score,
    style="c0.4c",
    pen="1p",
    fill="green3",
)
fig.show()
```



Setting Primary and Secondary Time Axes

This example focuses on annotating the axes and setting the interval in which the annotations should appear. All of these modifications are passed to the `frame` parameter and each item in that list modifies a specific aspect of the frame.

Adding `"WS"` means that only the Western/Left (**W**) and Southern/Bottom (**S**) borders of the plot are annotated. For more information on this, please refer to the *Frames, ticks, titles, and labels tutorial*.

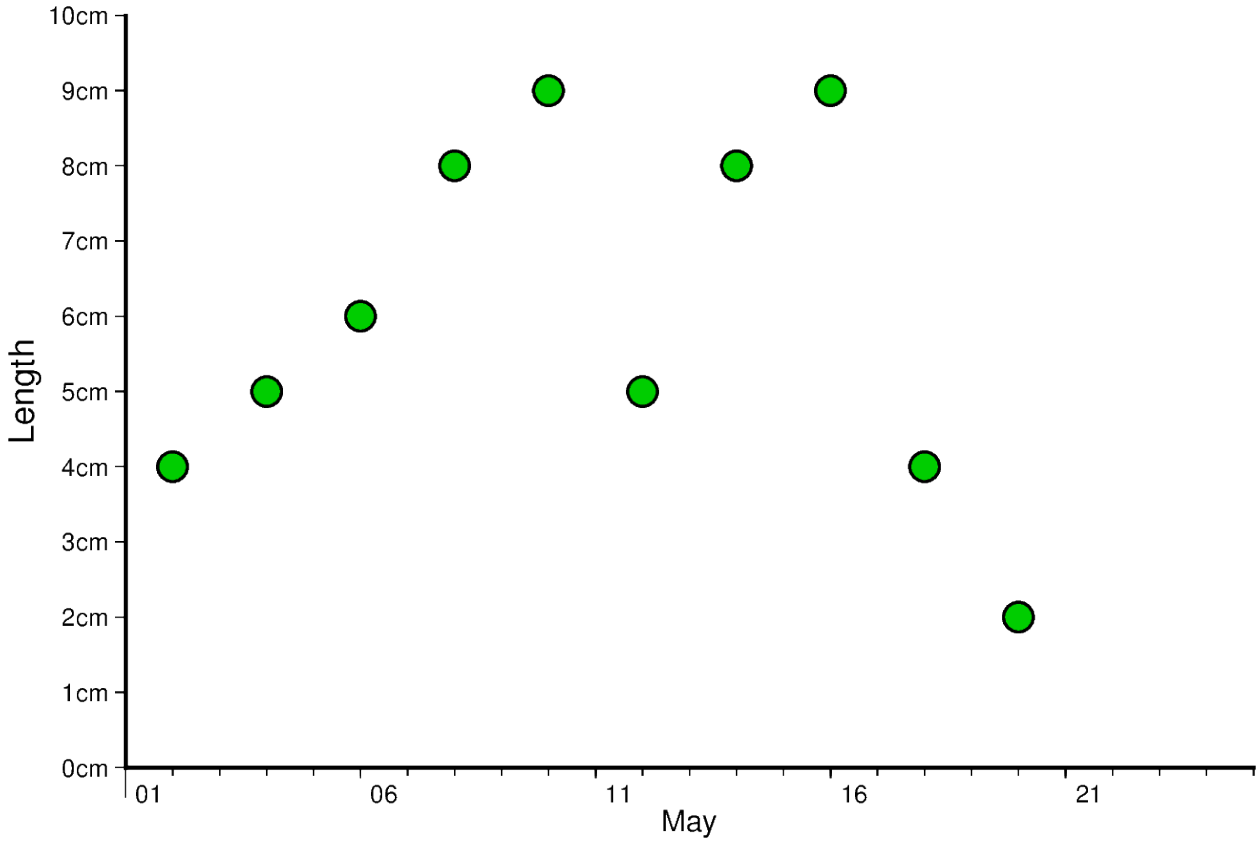
Another important item in the list passed to `frame` is `"sxa1Of1D"`. This string modifies the secondary annotation (**s**) of the x-axis (**x**). Specifically, it sets the main annotation and major tick spacing interval to one month (**a1O**) (capital letter O, not zero). Additionally, it sets the minor tick spacing interval to 1 day (**f1D**). To use the month name instead of its number set FORMAT_DATE_MAP to **o**. More information on configuring date formats can be found at FORMAT_DATE_MAP, FORMAT_DATE_IN, and FORMAT_DATE_OUT.

```
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"):
```

(continues on next page)

(continued from previous page)

```
    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"],
        x=x,
        y=y,
        style="c0.4c",
        pen="1p",
        fill="green3",
    )
fig.show()
```



The same concept shown above can be applied to smaller as well as larger intervals. In this example, data are plotted for different times throughout two days. The primary x-axis annotations are modified to repeat every 6 hours, and the secondary x-axis annotations repeat every day and show the day of the week.

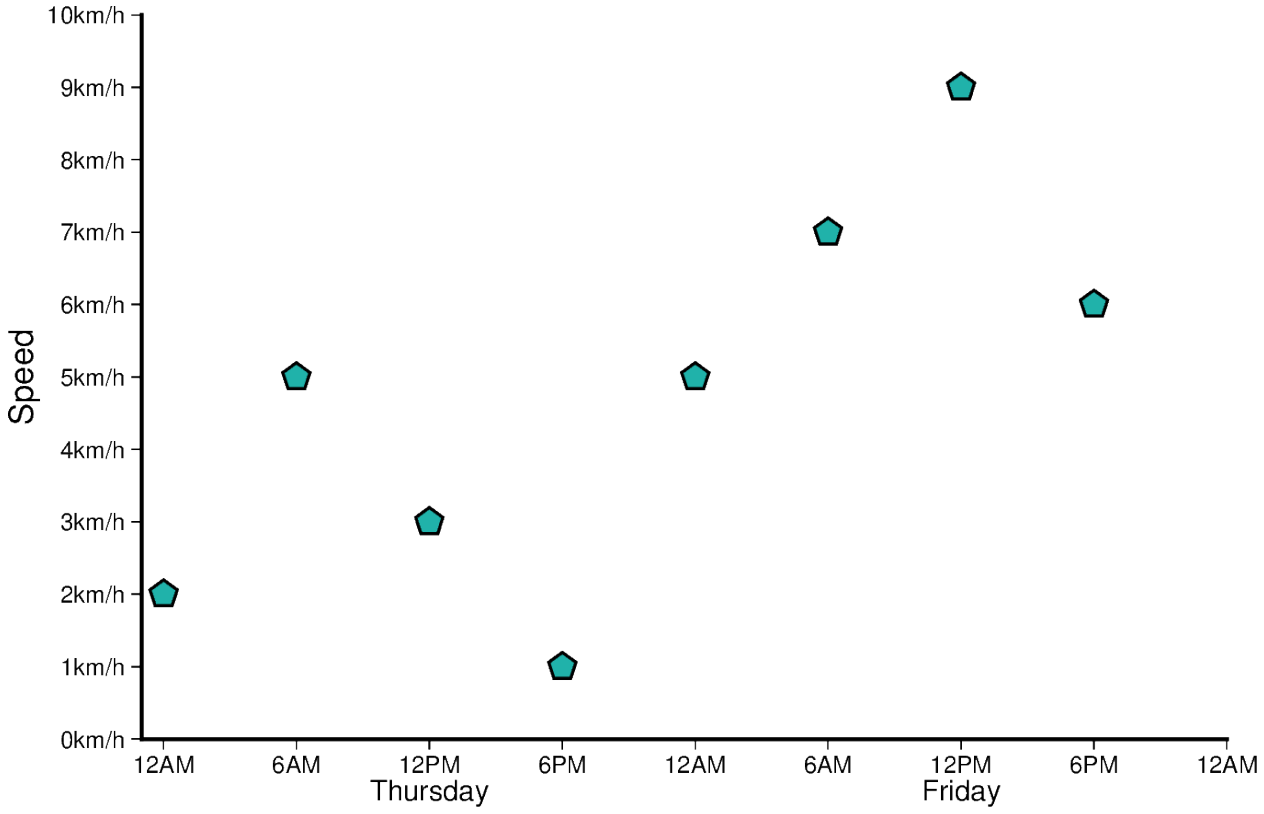
Another notable mention in this example is setting FORMAT_CLOCK_MAP to **-hhAM** which specifies the format used for time. In this case, leading zeros are removed using (**-**), and only hours are displayed. Additionally, an AM/PM system is used instead of a 24-hour system. More information on configuring time formats can be found at FORMAT_CLOCK_MAP, FORMAT_CLOCK_IN, and FORMAT_CLOCK_OUT.

```
x = pd.date_range("2021-04-15", periods=8, freq="6h")
y = [2, 5, 3, 1, 5, 7, 9, 6]

fig = pygmt.Figure()
with pygmt.config(FORMAT_CLOCK_MAP="-hhAM"):
```

(continues on next page)

(continued from previous page)

```
    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",
        fill="lightseagreen",
    )
fig.show()
```



Total running time of the script: (0 minutes 1.253 seconds)

4.2.13 Plotting focal mechanisms

Focal mechanisms can be plotted as beachballs with the *pygmt.Figure.meca* method.

The focal mechanism data or parameters can be provided as various input types: ASCII file, `numpy.array`, dictionary, or `pandas.Dataframe`. Different conventions to define the focal mechanism are supported: Aki and Richards (`"aki"`), global CMT (`"gcmt"`), moment tensor (`"mt"`), partial focal mechanism (`"partial"`), and, principal axis (`"principal_axis"`). Please refer to the table in the documentation of *pygmt.Figure.meca* regarding how to set

up the input data in respect to the chosen input type and convention (i.e., the expected column order, keys, or column names). In this tutorial we focus on how to adjust the display of the beachballs.

```
import pandas as pd
import pygmt

# Set up arguments for basemap
region = [-5, 5, -5, 5]
projection = "X10c/4c"
frame = ["af", "+ggray90"]
```

Setting up the focal mechanism data

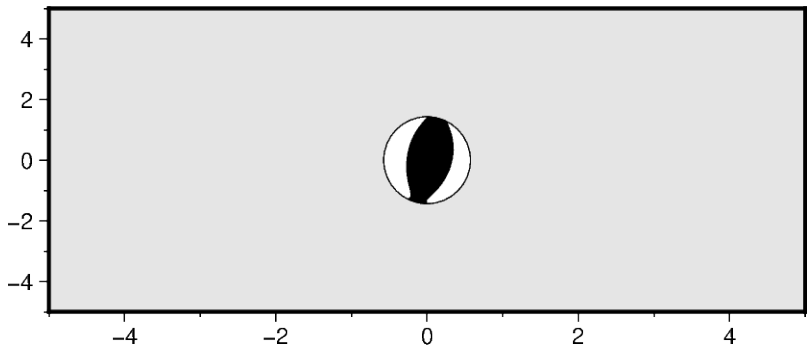
We store focal mechanism parameters for two single events in dictionaries using the moment tensor and Aki and Richards conventions:

```
# moment tensor convention
mt_single = {
    "mrr": 4.71,
    "mtt": 0.0381,
    "mff": -4.74,
    "mrt": 0.399,
    "mrf": -0.805,
    "mtf": -1.23,
    "exponent": 24,
}
# Aki and Richards convention
aki_single = {"strike": 318, "dip": 89, "rake": -179, "magnitude": 7.75}
```

Plotting a single beachball

Required parameters are `spec` and `scale` as well as `longitude`, `latitude` (event location), and depth (if these values are not included in the argument passed to `spec`). Additionally, the `convention` parameter is required if `spec` is an 1-D or 2-D numpy array; for the input types dictionary and `pandas.Dataframe`, the focal mechanism convention is automatically determined from dictionary keys or `pandas.DataFrame` column names. The `scale` parameter controls the radius of the beachball. By default, the value defines the size for a magnitude of 5 (i.e., a scalar seismic moment of $M_0 = 4.0 \times 10^{23}$ dyn cm) and the beachball size is proportional to the magnitude. Append `"+l"` to force the radius to be proportional to the seismic moment.

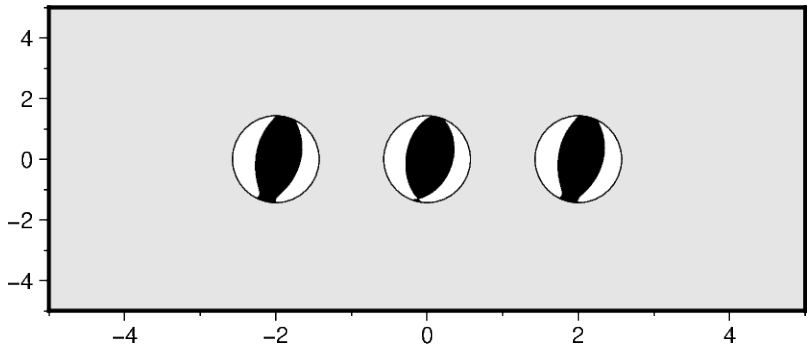
```
fig = pygmt.Figure()
fig.basemap(region=region, projection=projection, frame=frame)

fig.meca(spec=mt_single, scale="1c", longitude=0, latitude=0, depth=0)

fig.show()
```



Plotting the components of a seismic moment tensor

A moment tensor can be decomposed into isotropic and deviatoric parts. The deviatoric part can be further decomposed into multiple parts (e.g., a double couple (DC) and a compensated linear vector dipole (CLVD)). Use the `component` parameter to specify the component you want to plot.

```
fig = pygmt.Figure()
fig.basemap(region=region, projection=projection, frame=frame)

for component, longitude in zip(["full", "dc", "deviatoric"], [-2, 0, 2],␣
↪strict=True):
    fig.meca(
        spec=mt_single,
        scale="1c",
        longitude=longitude,
        latitude=0,
        depth=0,
        component=component,
    )

fig.show()
```



Filling the quadrants

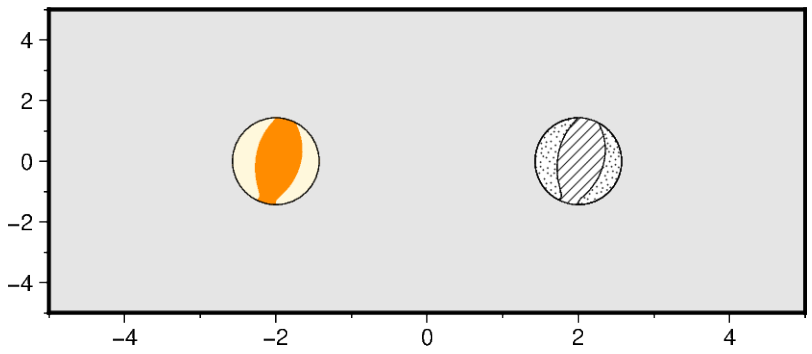
Use the parameters `compressionfill` and `extensionfill` to fill the quadrants with different colors or patterns. Regarding patterns see the gallery example *Bit and hachure patterns* and the Technical Reference *Bit and hachure patterns*.

```
fig = pygmt.Figure()
fig.basemap(region=region, projection=projection, frame=frame)

fig.meca(
```

(continues on next page)

(continued from previous page)

```
    spec=mt_single,
    scale="1c",
    longitude=-2,
    latitude=0,
    depth=0,
    compressionfill="darkorange",
    extensionfill="cornsilk",
)

fig.meca(
    spec=mt_single,
    scale="1c",
    longitude=2,
    latitude=0,
    depth=0,
    compressionfill="p8",
    extensionfill="p31",
    outline=True,
)

fig.show()
```



Adjusting the outlines

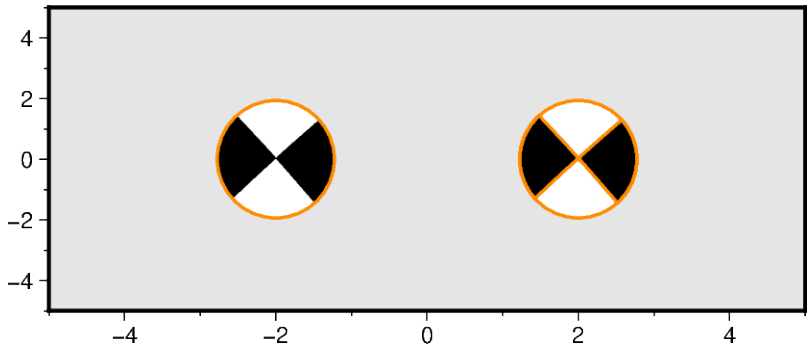
Use the parameters `pen` and `outline` for adjusting the circumference of the beachball or all lines (i.e, circumference and both nodal planes).

```
fig = pygmt.Figure()
fig.basemap(region=region, projection=projection, frame=frame)

fig.meca(
    spec=aki_single,
    scale="1c",
    longitude=-2,
    latitude=0,
    depth=0,
    # Use a 1-point thick, darkorange and solid line
    pen="1p,darkorange",
)

fig.meca(
```

(continues on next page)

(continued from previous page)

```
    spec=aki_single,
    scale="1c",
    longitude=2,
    latitude=0,
    depth=0,
    outline="1p,darkorange",
)

fig.show()
```



Highlighting the nodal planes

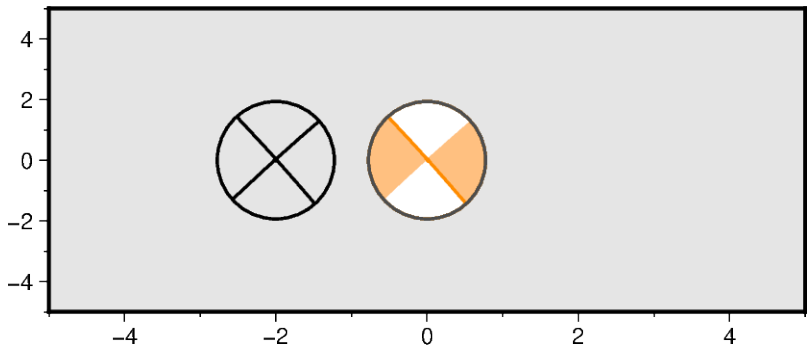
Use the parameter `nodal` to highlight specific nodal planes. `"0"` refers to both, `"1"` to the first, and `"2"` to the second nodal plane(s). Only the circumference and the specified nodal plane(s) are plotted, i.e. the quadrants remain unfilled (transparent). We can make use of the stacking concept of (Py)GMT, and use `nodal` in combination with the `outline`, `compressionfill` / `extensionfill` and `pen` parameters.

```
fig = pygmt.Figure()
fig.basemap(region=region, projection=projection, frame=frame)

fig.meca(
    spec=aki_single,
    scale="1c",
    longitude=-2,
    latitude=0,
    depth=0,
    nodal="0/1p,black",
)

# Plot the same beachball three times with different settings:
# (i) Fill the compressive quadrants
# (ii) Plot the first nodal plane and the circumference in darkorange
# (iii) Plot the circumfence in black on top; use "-" to not fill the quadrants
for kwargs in [
    {"compressionfill": "lightorange"},
    {"nodal": "1/1p,darkorange"},
    {"compressionfill": "-", "extensionfill": "-", "pen": "1p,gray30"},
]:
    fig.meca(
        spec=aki_single,
        scale="1c",
```

(continues on next page)

(continued from previous page)

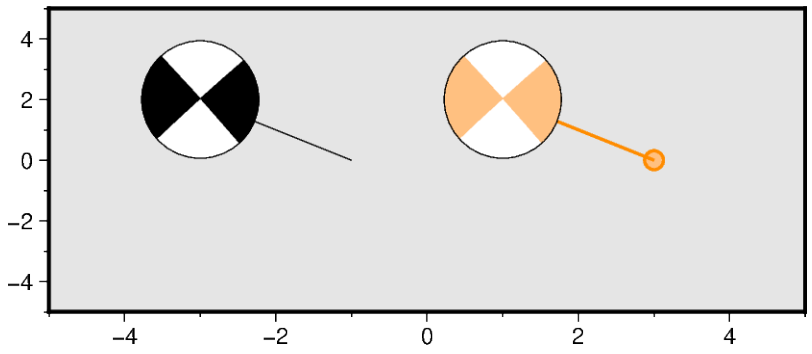
```
        longitude=0,
        latitude=0,
        depth=0,
        **kwargs,
    )
fig.show()
```



Adding offset from event location

Specify the optional parameters `plot_longitude` and `plot_latitude`. If `spec` is an ASCII file with columns for `plot_longitude` and `plot_latitude`, the `offset` parameter has to be set to `True`. Besides just drawing a line between the beachball and the event location, a small circle can be plotted at the event location by appending **+s** and the descired circle diameter. The connecting line as well as the outline of the circle are plotted with the setting of pen, or can be adjusted separately. The fill of the small circle corresponds to the fill of the compressive quadrantes.

```
fig = pygmt.Figure()
fig.basemap(region=region, projection=projection, frame=frame)

fig.meca(
    spec=aki_single,
    scale="1c",
    longitude=-1,
    latitude=0,
    depth=0,
    plot_longitude=-3,
    plot_latitude=2,
)

fig.meca(
    spec=aki_single,
    scale="1c",
    longitude=3,
    latitude=0,
    depth=0,
    plot_longitude=1,
    plot_latitude=2,
    offset="+p1p,darkorange+s0.25c",
    compressionfill="lightorange",
)

fig.show()
```



Plotting multiple beachballs

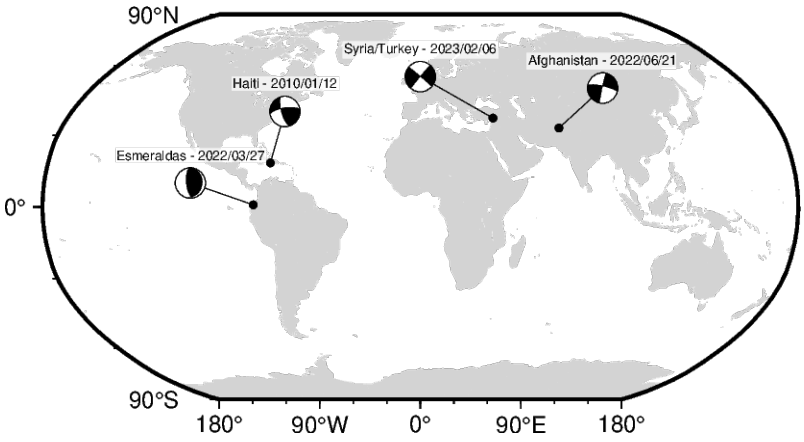
Now we want to plot multiple beachballs with one call of *pygmt.Figure.meca*. We use data of four earthquakes taken from USGS. For each focal mechanism parameter a list with a length corresponding to the number of events has to be given.

```
# Set up a pandas.DataFrame with multiple focal mechanism parameters.
aki_multiple = pd.DataFrame(
    {
        "strike": [255, 173, 295, 318],
        "dip": [70, 68, 79, 89],
        "rake": [20, 83, -177, -179],
        "magnitude": [7.0, 5.8, 6.0, 7.8],
        "longitude": [-72.53, -79.61, 69.46, 37.01],
        "latitude": [18.44, 0.90, 33.02, 37.23],
        "depth": [13, 19, 4, 10],
        "plot_longitude": [-70, -110, 100, 0],
        "plot_latitude": [40, 10, 50, 55],
        "event_name": [
            "Haiti - 2010/01/12",
            "Esmeraldas - 2022/03/27",
            "Afghanistan - 2022/06/21",
            "Syria/Turkey - 2023/02/06",
        ],
    }
)
```

Adding a label

Use the optional parameter `event_name` to add a label near the beachball, e.g., event name or event date and time. Change the font of the label text by appending **+f** and the desired font (size,name,color) to the argument passed to the `scale` parameter. Additionally, the location of the label relative to the beachball [Default is `"TC"`, i.e., Top Center] can be changed by appending **+j** and an offset can be applied by appending **+o** with values for *dx*/*dy*. Add a colored [Default is white] box behind the label via the label `labelbox`. Force a fixed size of the beachball by appending **+m** to the argument passed to the `scale` parameter.

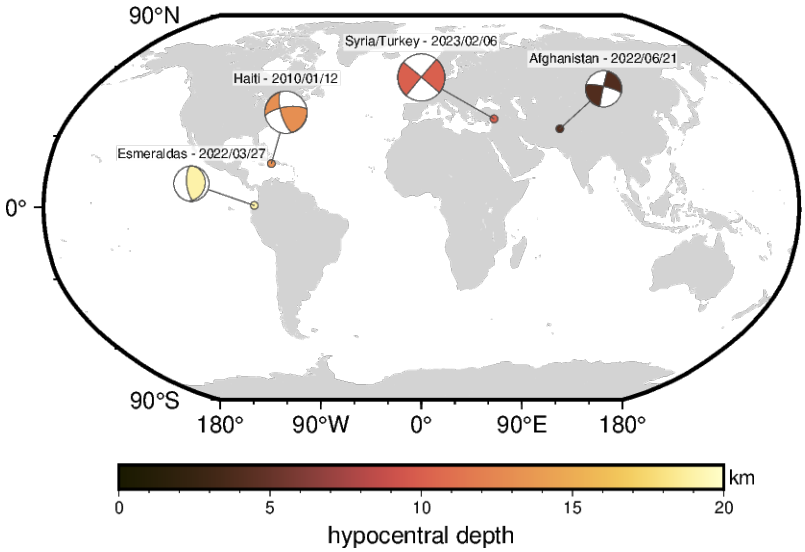
```
fig = pygmt.Figure()
fig.coast(region="d", projection="N10c", land="lightgray", frame=True)

fig.meca(spec=aki_multiple, scale="0.4c+m+f5p", labelbox="white@30", offset="+s0.1c")

fig.show()
```



Using size-coding and color-coding

The beachball can be sized and colored by the quantities given as `magnitude` and `depth`, e.g., by moment magnitude or hypocentral depth, respectively. Use the parameter `cmap` to pass the descired colormap. Now, the fills of the small circles indicating the event locations are given by the colormap.

```
fig = pygmt.Figure()
fig.coast(region="d", projection="N10c", land="lightgray", frame=True)

# Set up colormap and colorbar for hypocentral depth
pygmt.makecpt(cmap="lajolla", series=[0, 20])
fig.colorbar(frame=["x+lhypocentral depth", "y+lkm"])

fig.meca(
    spec=aki_multiple,
    scale="0.4c+f5p",
    offset="0.2p,gray30+s0.1c",
    labelbox="white@30",
    cmap=True,
    outline="0.2p,gray30",
)

fig.show()
```



Total running time of the script: (0 minutes 1.251 seconds)

4.2.14 Plotting vectors

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

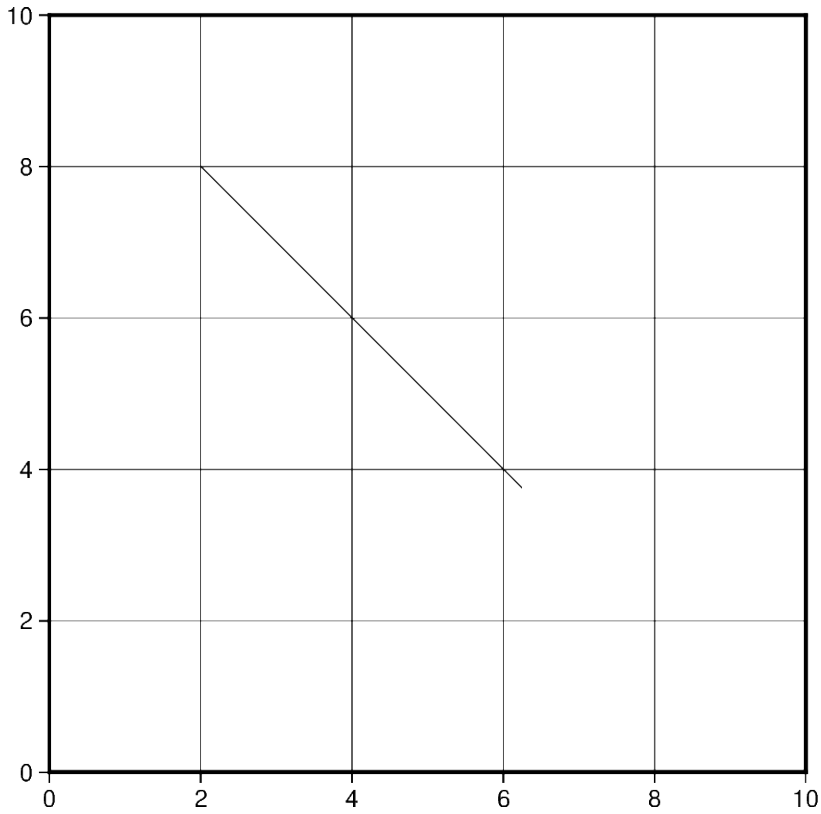
```
import numpy as np
import pygmt
```

Plot Cartesian Vectors

Create a simple Cartesian vector using a start 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 1-D 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.

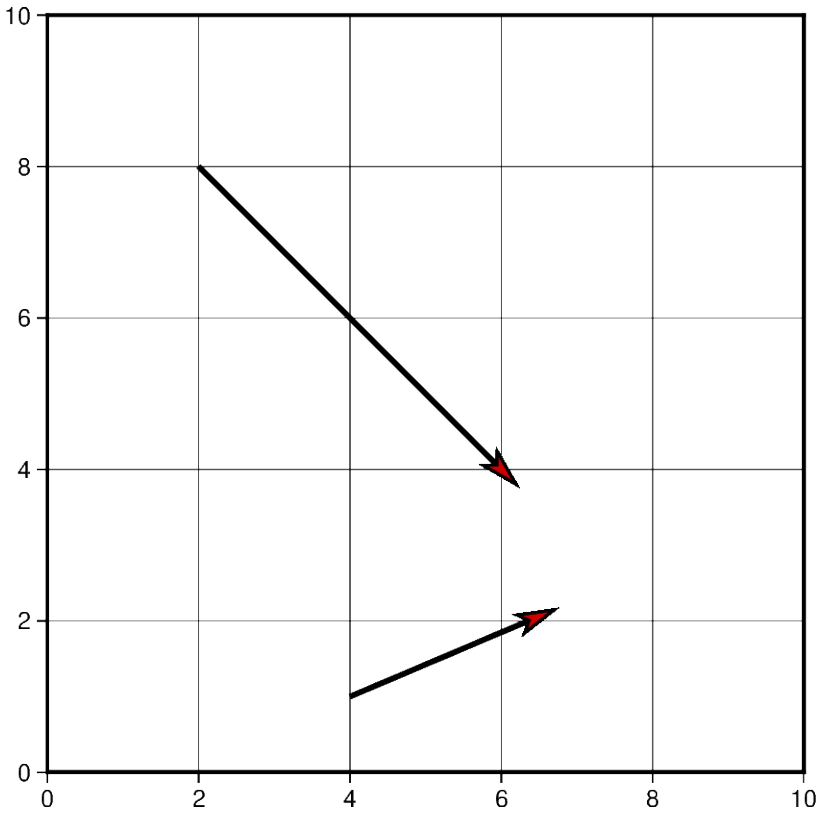
```
fig = pygmt.Figure()
fig.plot(
    region=[0, 10, 0, 10],
    projection="X10c/10c",
    frame="ag",
    x=2,
    y=8,
    style="v0c",
    direction=[[-45], [6]],
)
fig.show()
```



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



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.

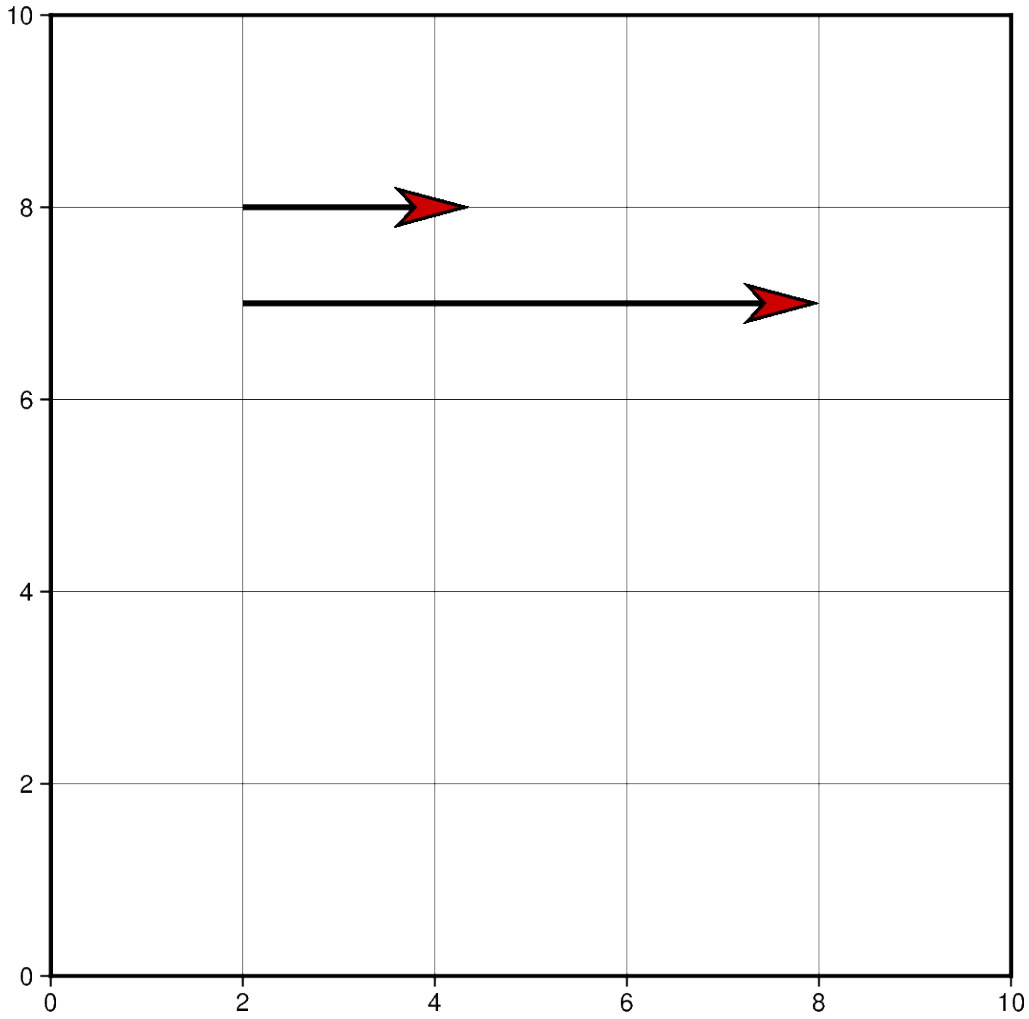
In this example, the graphed region is 5in X 5in, but the length of the first vector is still graphed in centimeters. Using `pygmt.config(PROJ_LENGTH_UNIT="i")`, the default unit can be changed to inches in the second plotted vector.

```
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,
    style="v1c+e",
    direction=[[0], [3]],
    pen="2p",
    fill="red3",
)
# Vector 2 after changing default unit to inches
with pygmt.config(PROJ_LENGTH_UNIT="i"):
    fig.plot(
        x=2,
        y=7,
        direction=[[0], [3]],
        style="v1c+e",
        pen="2p",
        fill="red3",
```

(continues on next page)

(continued from previous page)

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



Vectors can also be plotted by including all the information about a vector in a single list. However, this requires creating a 2-D list or numpy array containing all vectors. Each vector list contains the information structured as: `[x_start, y_start, direction_degrees, length]`.

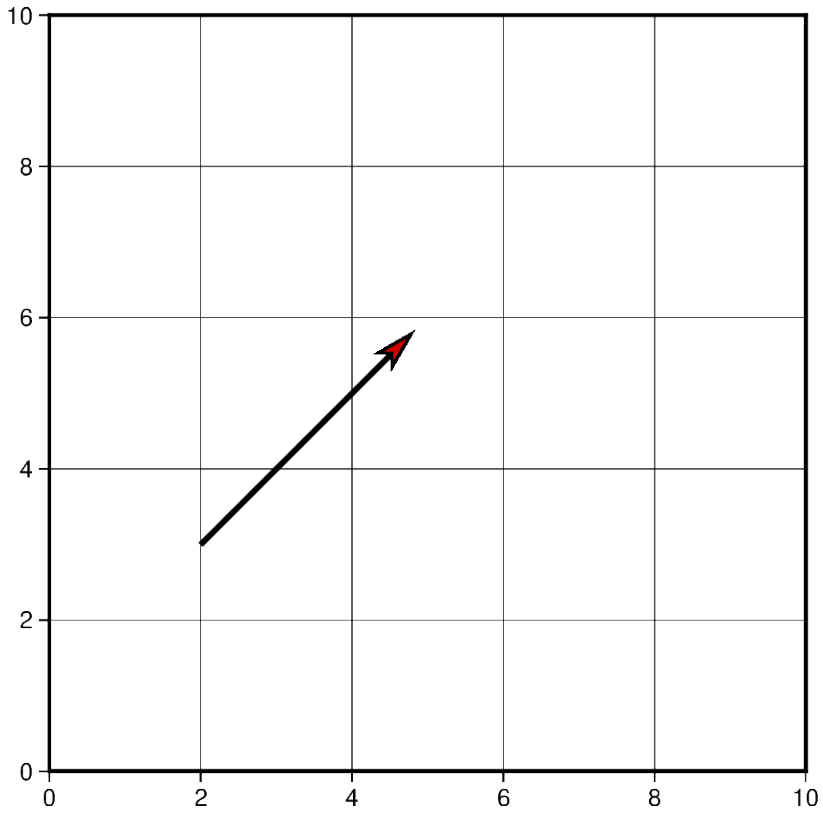
If this approach is chosen, the `data` parameter must be used instead of `x`, `y`, and `direction`.

```
# 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",
    fill="red3",
```

(continues on next page)

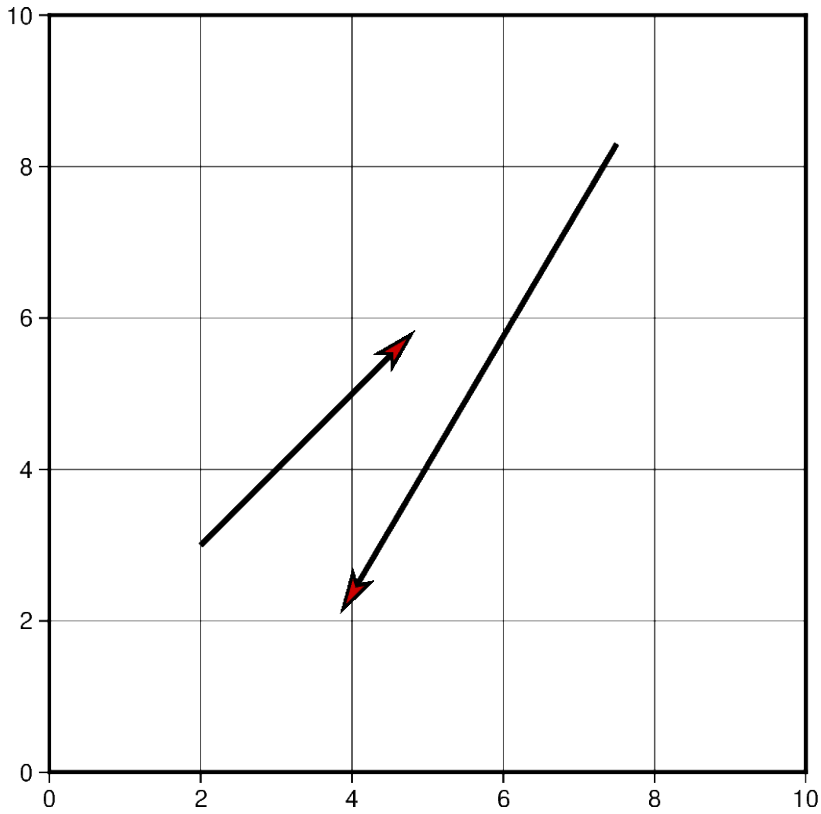
(continued from previous page)

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



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 2-D 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",
    fill="red3",
)
fig.show()
```



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.

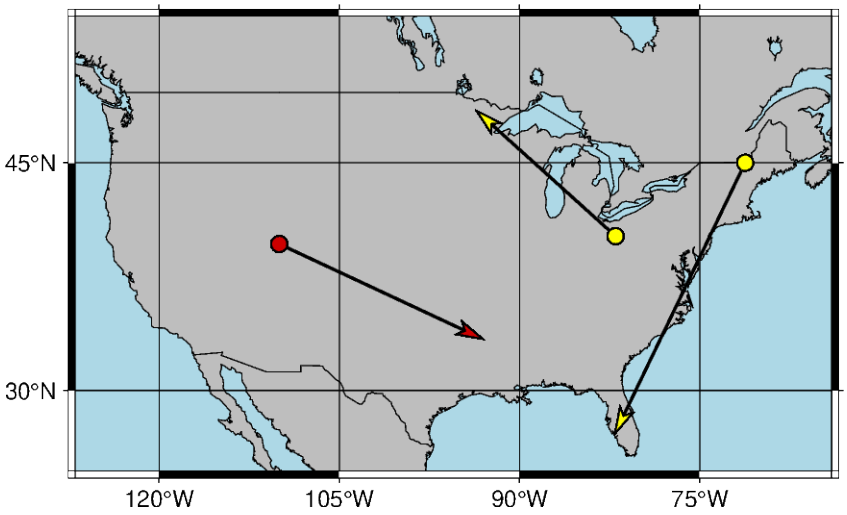
This example also shows some of the styles a vector supports. The beginning point of the vector (**+b**) should take the shape of a circle (**c**). Similarly, the end point of the vector (**+e**) should have an arrow shape (**a**) (to draw a plain arrow, use **A** instead). Lastly, the **+a** specifies 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",
```

(continues on next page)

(continued from previous page)

```
    fill="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",
    fill="yellow",
)
fig.show()
```



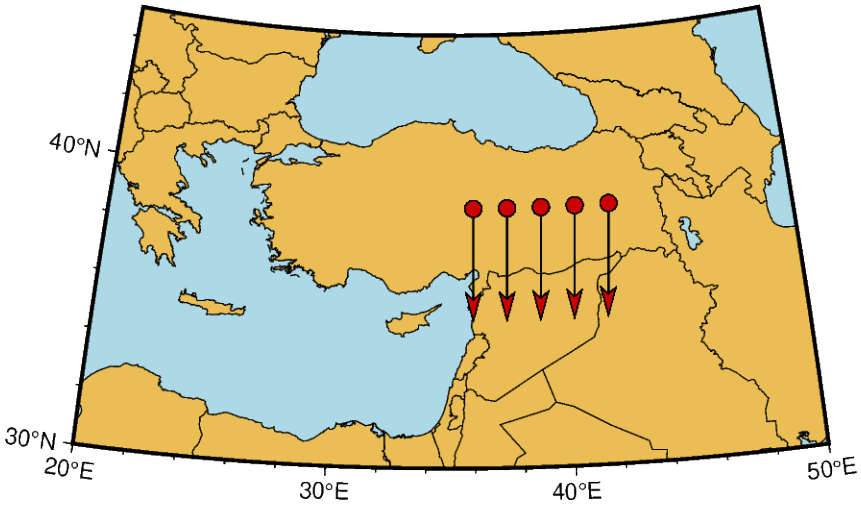
Another example of plotting Cartesian vectors over a coast plot. This time a Transverse Mercator projection is used. Additionally, `numpy.linspace` is used to create 5 vectors with equal stops.

```
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,
# Transverse Mercator projection (T) over Turkey and Syria
fig = pygmt.Figure()
fig.coast(
    region=[20, 50, 30, 45],
    projection="T35/10c",
    frame=True,
    borders=1,
    shorelines="0.25p,black",
```

(continues on next page)

(continued from previous page)

```
    area_thresh=4000,
    land="lightbrown",
    water="lightblue",
)

fig.plot(
    x=x,
    y=y,
    style="v0.4c+ea+bc",
    direction=[direction, length],
    pen="0.6p",
    fill="red3",
)

fig.show()
```



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.

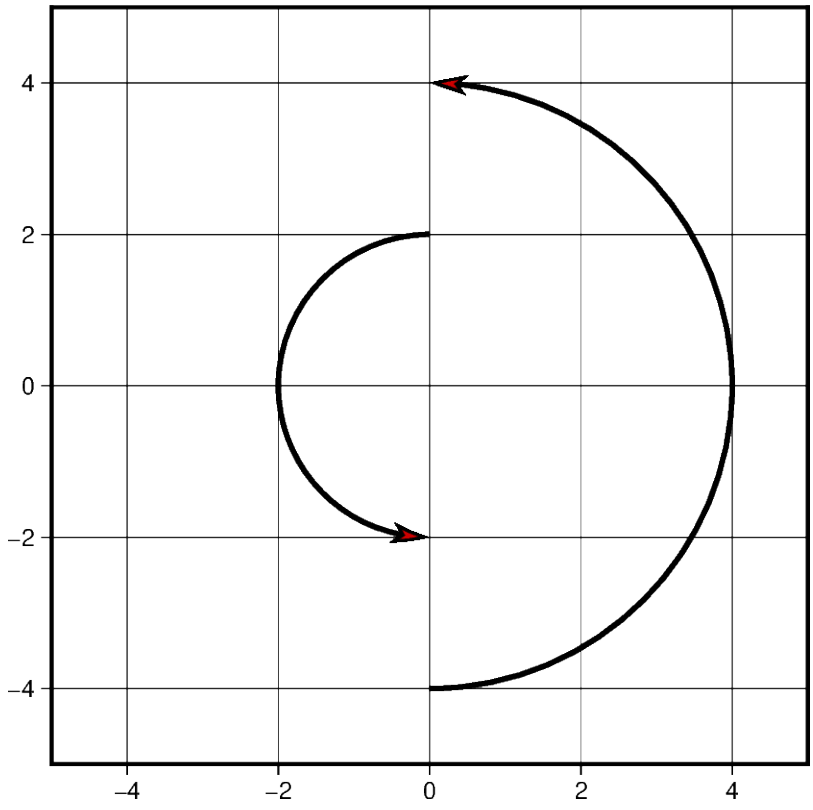
The last two values in the vector list represent the degree at which the plot will start and stop. These values are measured counter-clockwise from the horizontal axis. In this example, the result shown is the left half of a circle as the plot starts at 90 degrees and goes until 270. Notice that the `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,
```

(continues on next page)

(continued from previous page)

```
    style="m0.5c+ea",
    pen="2p",
    fill="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",
    pen="2p",
    fill="red3",
)
fig.show()
```



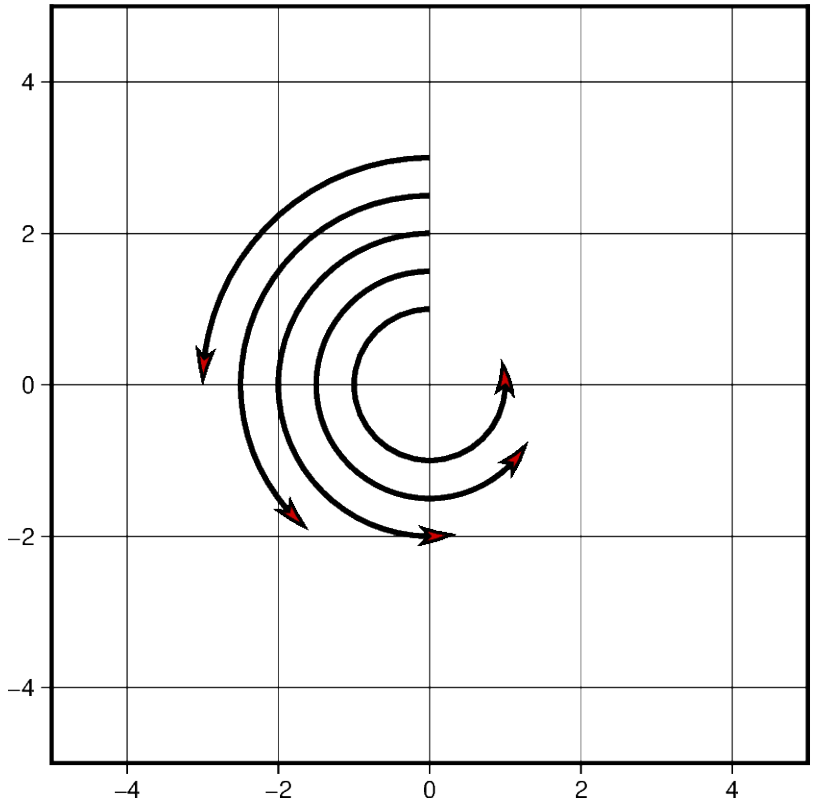
When plotting multiple circular vectors, a two dimensional array or numpy array object should be passed as the `data` parameter. In this example, `numpy.column_stack` is used to generate this two dimensional array. Other numpy objects are used to generate linear values for the `radius` parameter and 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(
```

(continues on next page)

(continued from previous page)

```
    [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",
    data=data,
    style="m0.5c+ea",
    pen="2p",
    fill="red3",
)
fig.show()
```



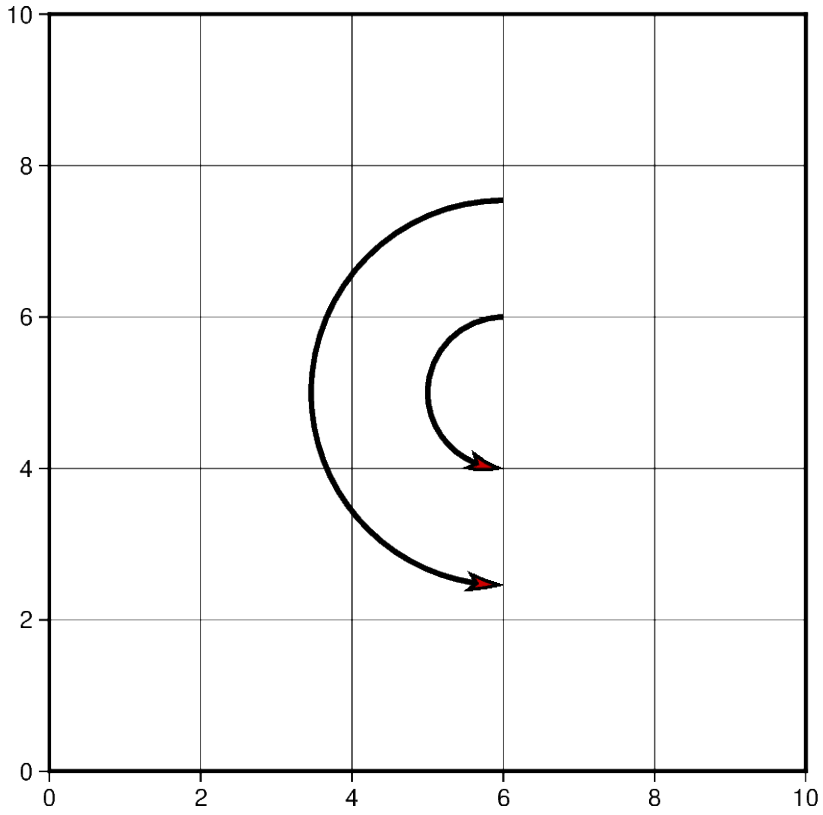
Much like when plotting Cartesian vectors, the default unit used is centimeters. When this is changed to inches, the size of the plot appears larger when the projection units do not change. Below is an example of two circular vectors. One is plotted using the default unit, and the second is plotted using 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",
```

(continues on next page)

(continued from previous page)

```
    data=[circular_vector],
    style="m0.5c+ea",
    pen="2p",
    fill="red3",
)

with pygmt.config(PROJ_LENGTH_UNIT="i"):
    fig.plot(
        data=[circular_vector],
        style="m0.5c+ea",
        pen="2p",
        fill="red3",
    )
fig.show()
```



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 `fill` parameters.

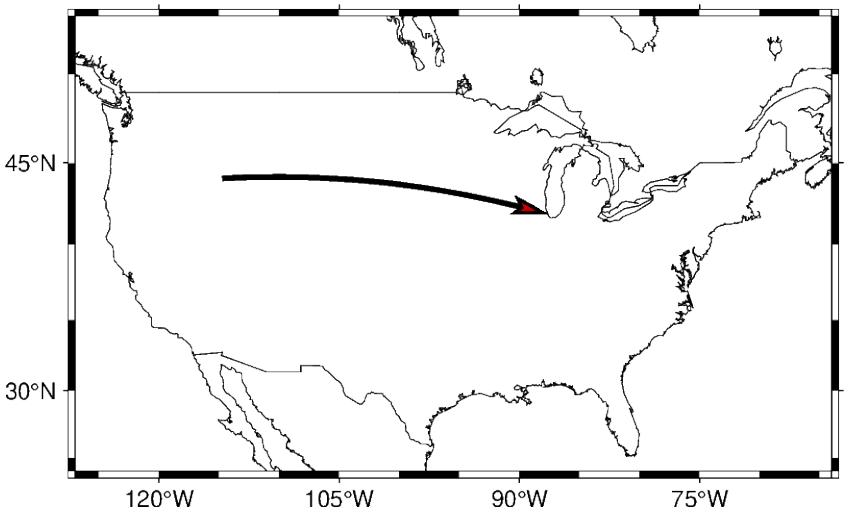
Note that the **+s** is added to use a start point and an end point to represent the vector instead of input angle and length.

```
point_1 = [-114.7420, 44.0682]
point_2 = [-87.6298, 41.8781]
data = np.array([point_1 + point_2])
```

(continues on next page)

(continued from previous page)

```
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",
    fill="red3",
)
fig.show()
```



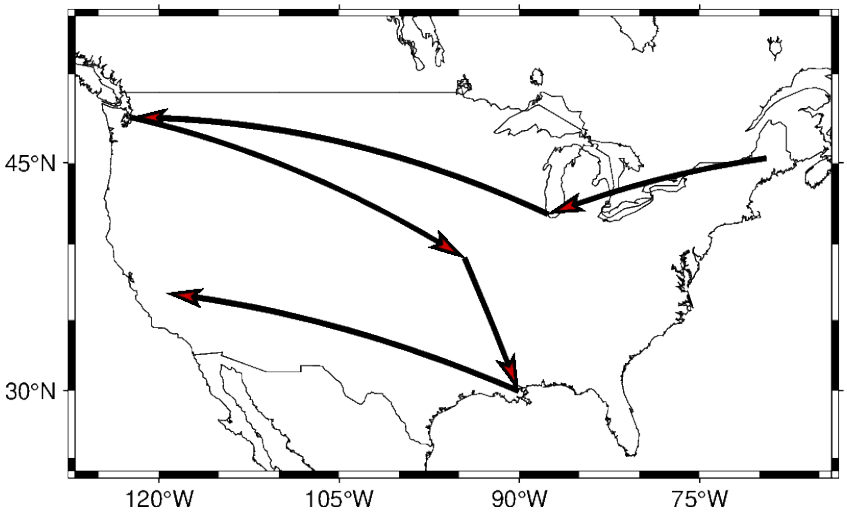
Using the same technique shown in the previous example, multiple vectors can be plotted in a chain where the end point of one is the start point of another. This can be done by adding the coordinate lists together to create this structure: `[[start_latitude, start_longitude, end_latitude, end_longitude]]`. Each list within the 2-D 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]

# Add array to piece together the vectors
data = [ME + CHI, CHI + SEA, SEA + KC, KC + NO, NO + CA]

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

(continues on next page)

(continued from previous page)

```
    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",
    fill="red3",
)
fig.show()
```



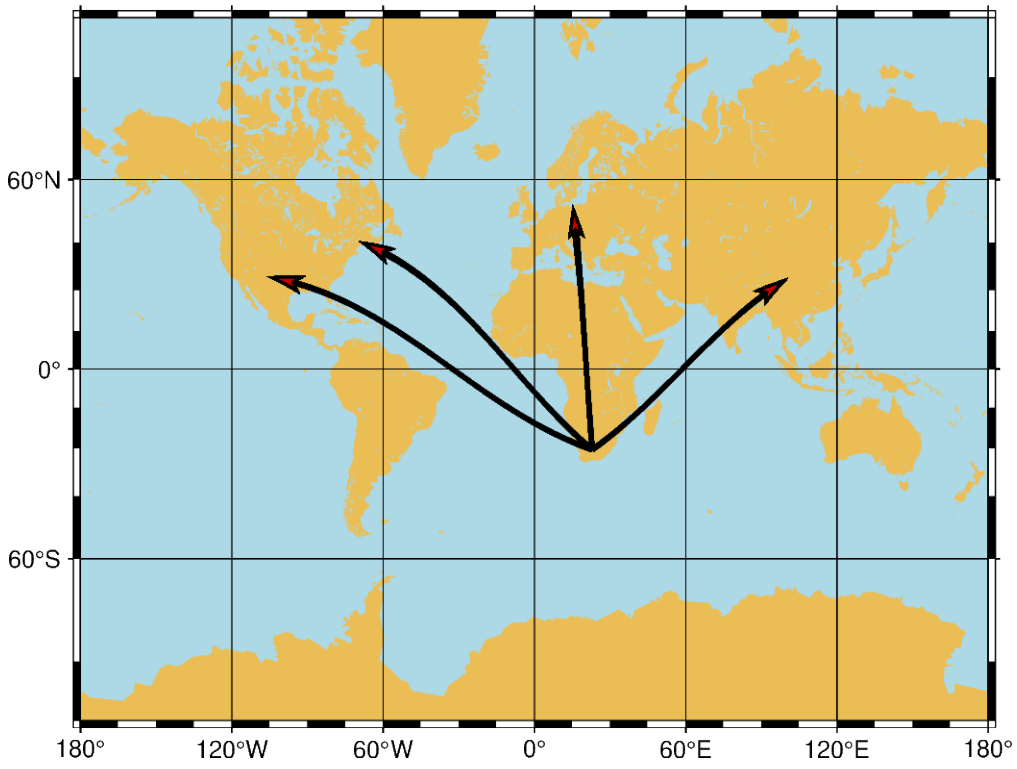
This example plots vectors over a Mercator projection. The start points are located at SA which is South Africa and going to four different locations.

```
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(
    region=[-180, 180, -80, 80],
    projection="M0/0/12c",
    frame="afg",
    land="lightbrown",
    water="lightblue",
)
fig.plot(
    data=data,
    style="=0.5c+ea+s",
```

(continues on next page)

(continued from previous page)

```
    pen="2p",
    fill="red3",
)
fig.show()
```



Total running time of the script: (0 minutes 2.484 seconds)

4.2.15 Typesetting non-ASCII text

In addition to ASCII printable characters, sometimes you may also want to typeset non-ASCII characters on the plot, such as Greek letters, mathematical symbols, or special characters.

Due to the limitations of the underlying PostScript language, PyGMT doesn't support all characters in the Unicode standard. Instead, PyGMT supports a limited set of characters in the "Adobe Symbol", "Adobe ZapfDingbats", "Adobe ISOLatin1+", and "ISO-8859-*x*" (*x* can be 1-11, 13-16) encodings. Refer to *Supported Encodings and Non-ASCII Characters* for the complete list of supported characters.

In PyGMT, the supported (ASCII and non-ASCII) characters can be directly used in the `text` parameter of the *`pygmt.Figure.text`* method for typesetting text strings. They can also be used in the arguments of other plotting functions (e.g., in the `frame` parameter to set the labels or title).

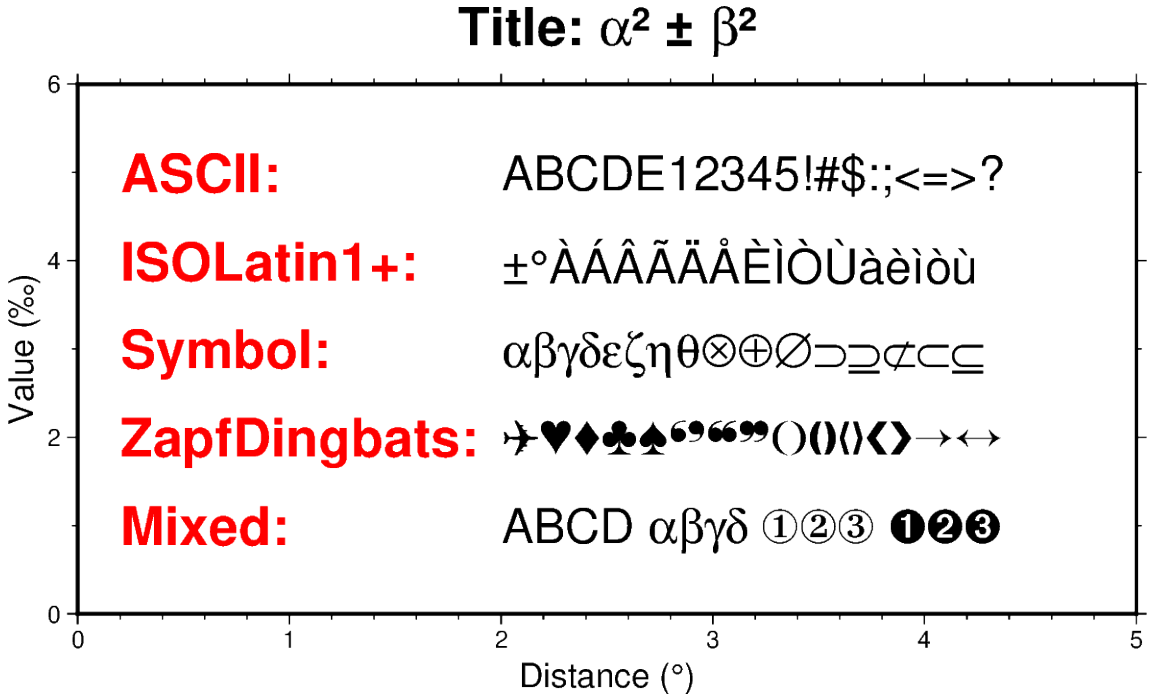
In this example, we demonstrate how to typeset non-ASCII characters in PyGMT.

```
import pygmt

fig = pygmt.Figure()
fig.basemap(
    region=[0, 5, 0, 6],
    projection="X14c/7c",
    frame=["xaf+lDistance (°)", "yaf+lValue (‰)", "WSen+tTitle: α² ± β²"],
```

(continues on next page)

(continued from previous page)

```
)

fig.text(
    x=[0.2] * 5,
    y=[5, 4, 3, 2, 1],
    text=["ASCII:", "ISOLatin1+:", "Symbol:", "ZapfDingbats:", "Mixed:"],
    font="20p,Helvetica-Bold,red",
    justify="LM",
)
fig.text(
    x=[2] * 5,
    y=[5, 4, 3, 2, 1],
    text=[
        "ABCDE12345!#$:;<=>?",  # ASCII only
        "±°ÀÁÂÃÄÅÈÌÒÙàèìòù",  # Non-ASCII characters from Adobe ISOLatin1+
        "αβγδεζηθ⊗⊕∅⊃⊇⊄⊂⊆",  # Non-ASCII characters from Adobe Symbol
        "?♥♦♣♠???????????→↔",  # Non-ASCII characters from Adobe ZapfDingbats
        "ABCD αβγδ ①②③ ???",  # Mix characters from ISOLatin1+, Symbol and␣
↪ZapfDingbats
    ],
    font="18p,Helvetica",
    justify="LM",
)

fig.show()
```



Here are some important tips when using non-ASCII characters:

- **Similar-looking characters**: Be cautious when using characters that appear visually similar but are distinct. For example, Ω (OHM SIGN) and Ω (GREEK CAPITAL LETTER OMEGA) may look alike, but PyGMT only supports the latter. Using the incorrect character can lead to unexpected results. To avoid this, it's recommended to

copy and paste characters from the *Supported Encodings and Non-ASCII Characters* documentation.

- **Mix characters from different encodings**: As shown in the example above, you can mix characters from different encodings in the same text string. However, due to the limitations of the underlying PostScript language, you cannot mix characters from the "Adobe ISOLatin1+" and "ISO-8859-*x*" encodings in the same text string. For example, you cannot mix characters from "Adobe ISOLatin1+" and "ISO-8859-2". If you need to use characters from different encodings, you can use them in different PyGMT function/method calls.
- **Non-ASCII characters in text files**: Non-ASCII characters are not supported if you have them in a text file and pass it to *pygmt.Figure.text*. In this case, you may want to load the text file into `pandas.DataFrame` and then pass it to the `text` parameter.

Total running time of the script: (0 minutes 0.182 seconds)

CHAPTER

FIVE

GALLERY

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

5.1 Maps and map elements

5.1.1 Choropleth map

The *pygmt.Figure.plot* method allows us to plot geographical data such as polygons 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. You can also use a full URL pointing to your desired data source. Then, pass the geopandas.GeoDataFrame as an argument to the data parameter of *pygmt.Figure.plot*, and style the geometry using the pen parameter. To fill the polygons based on a corresponding column you need to set fill="+z" as well as select the appropriate column using the aspatial parameter as shown in the example below.

```
import geodatasets
import geopandas as gpd
import pygmt

# Read the example dataset provided by geodatasets.
gdf = gpd.read_file(geodatasets.get_path("geoda airbnb"))
print(gdf.head())
```

```
Downloading file 'airbnb.zip' from 'https://geodacenter.github.io/data-and-lab//data/
↪airbnb.zip' to '/home/runner/.cache/geodatasets'.
         community  ...                                           geometry
0          DOUGLAS  ...  POLYGON ((-87.60914 41.84469, -87.60915 41.844...
1          OAKLAND  ...  POLYGON ((-87.59215 41.81693, -87.59231 41.816...
2      FULLER PARK  ...  POLYGON ((-87.6288 41.80189, -87.62879 41.8017...
3  GRAND BOULEVARD  ...  POLYGON ((-87.60671 41.81681, -87.6067 41.8165...
4          KENWOOD  ...  POLYGON ((-87.59215 41.81693, -87.59215 41.816...

[5 rows x 21 columns]
```

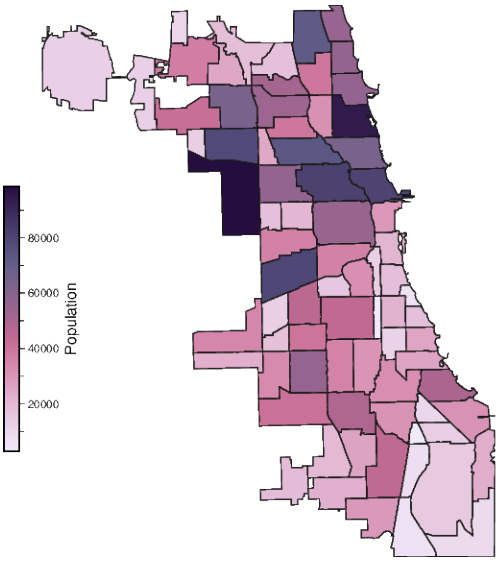
```
fig = pygmt.Figure()

fig.basemap(
    region=gdf.total_bounds[[0, 2, 1, 3]],
    projection="M6c",
    frame="+tPopulation of Chicago",
)
```

(continues on next page)

(continued from previous page)

```
# The dataset contains different attributes, here we select the "population" column to
# plot.

# First, we define the colormap to fill the polygons based on the "population" column.
pygmt.makecpt(
    cmap="acton",
    series=[gdf["population"].min(), gdf["population"].max(), 10],
    continuous=True,
    reverse=True,
)

# Next, we plot the polygons and fill them using the defined colormap. The target column
# is defined by the aspatial parameter.
fig.plot(
    data=gdf,
    pen="0.3p,gray10",
    fill="+z",
    cmap=True,
    aspatial="Z=population",
)

# Add colorbar legend.
fig.colorbar(frame="x+lPopulation", position="jML+o-0.5c+w3.5c/0.2c")

fig.show()
```

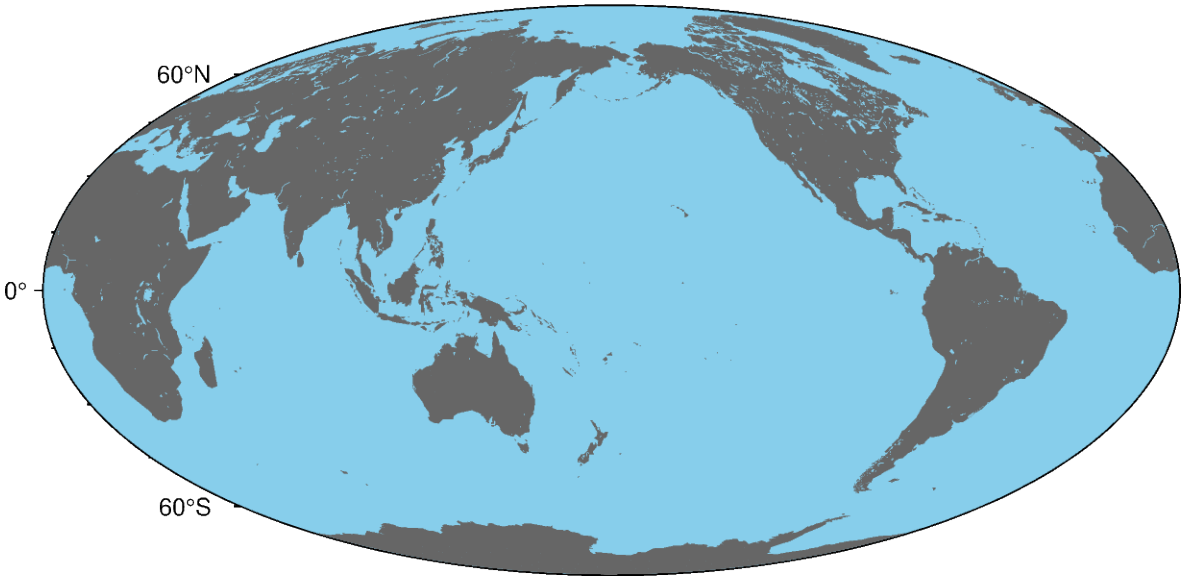
Population of Chicago



Total running time of the script: (0 minutes 0.666 seconds)

5.1.2 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 gray level (like `"gray50"`).




```
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 0.229 seconds)

5.1.3 Highlight country, continent and state polygons

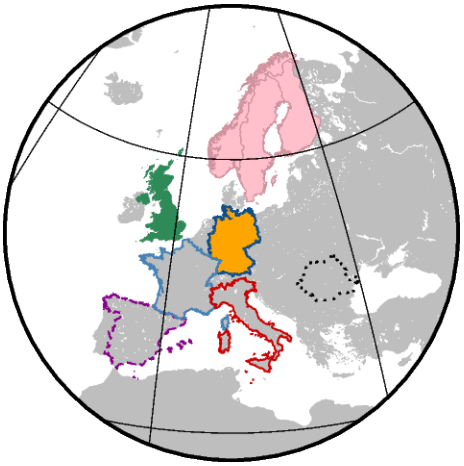
The *`pygmt.Figure.coast`* method can highlight country polygons via the `dcw` parameter. It accepts the country code or full country name and can draw its borders and add a color to its landmass. It's also possible to define multiple countries at once by separating the individual names with commas.

```
import pygmt

fig = pygmt.Figure()

fig.coast(
    region=[-12, 32, 34, 72],
    # Lambert Azimuthal Equal Area lon0/lat0/horizon/width
    projection="A10/52/25/6c",
    land="gray",
    water="white",
    frame="afg",
    dcw=[
```

(continues on next page)

(continued from previous page)

```
        # Great Britain (country code) with seagreen land
        "GB+gseagreen",
        # Italy with a red border
        "IT+p0.5p,red3",
        # Spain with a magenta dashed border
        "ES+p0.5p,magenta4,-",
        # Romania with a black dotted border
        "RO+p0.75p,black,.",
        # Germany with orange land and a blue border
        "DE+gorange+p0.5p,dodgerblue4",
        # France (full country name) with a steelblue border
        "France+p0.5p,steelblue",
        # Norway, Sweden and Finland (multiple countries) with pink
        # land and pink3 borders
        "Norway,Sweden,Finland+gpink+p0.2p,pink3",
    ],
)

fig.show()
```



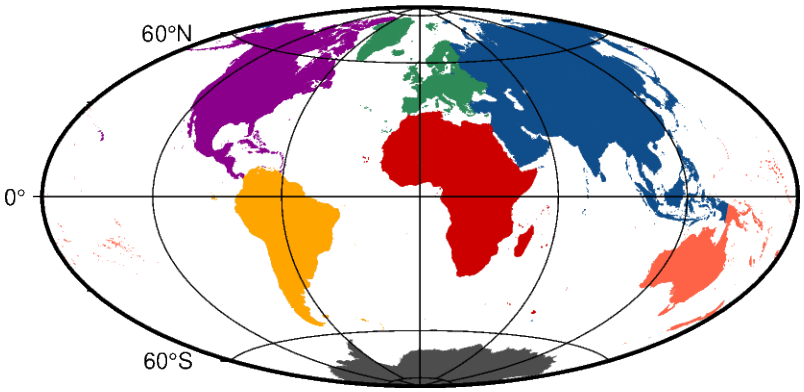
Entire continents can also be highlighted by adding `"="` in front of the continent code to differentiate it from a country code.

```
fig = pygmt.Figure()

fig.coast(
    region="d",
    projection="H10c",
    land="gray",
    water="white",
    frame="afg",
    dcw=[
        # Europe
        "=EU+gseagreen",
        # Africa
        "=AF+gred3",
        # North America
```

(continues on next page)

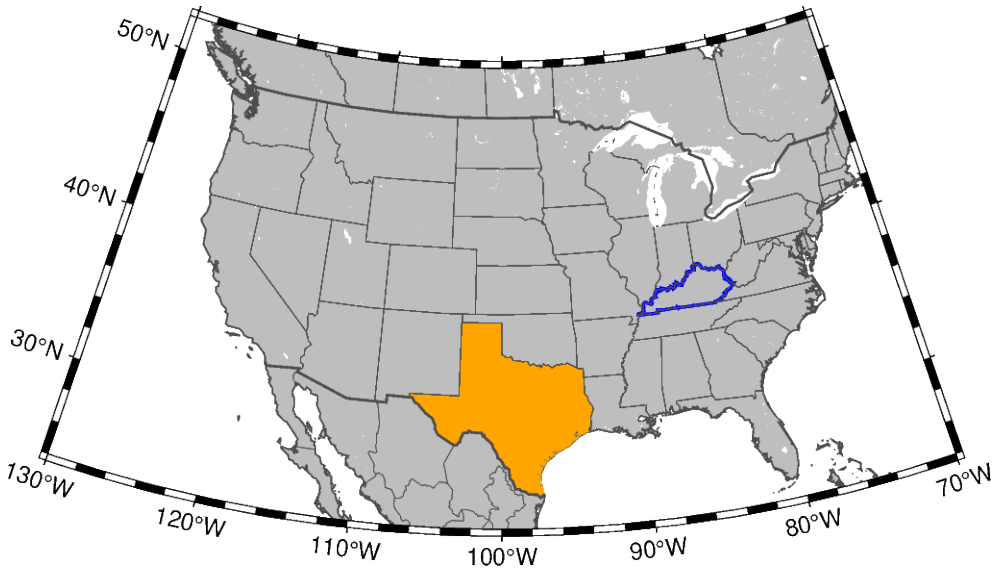
(continued from previous page)

```
        "=NA+gmagenta4",
        # South America
        "=SA+gorange",
        # Asia
        "=AS+gdodgerblue4",
        # Oceania
        "=OC+gtomato",
        # Antarctica
        "=AN+ggray30",
    ],
)

fig.show()
```



If available, states/territories of a country can be highlighted, too.

```
fig = pygmt.Figure()

fig.coast(
    region=[-130, -70, 24, 52],
    projection="L-100/35/33/45/12c",
    land="gray",
    shorelines="1/0.5p,gray30",
    borders=["1/0.8p,gray30", "2/0.2p,gray30"],
    frame=True,
    dcw=[
        # Texas with orange fill
        "US.TX+gorange",
        # Kentucky with blue outline
        "US.KY+p1p,blue",
    ],
)

fig.show()
```



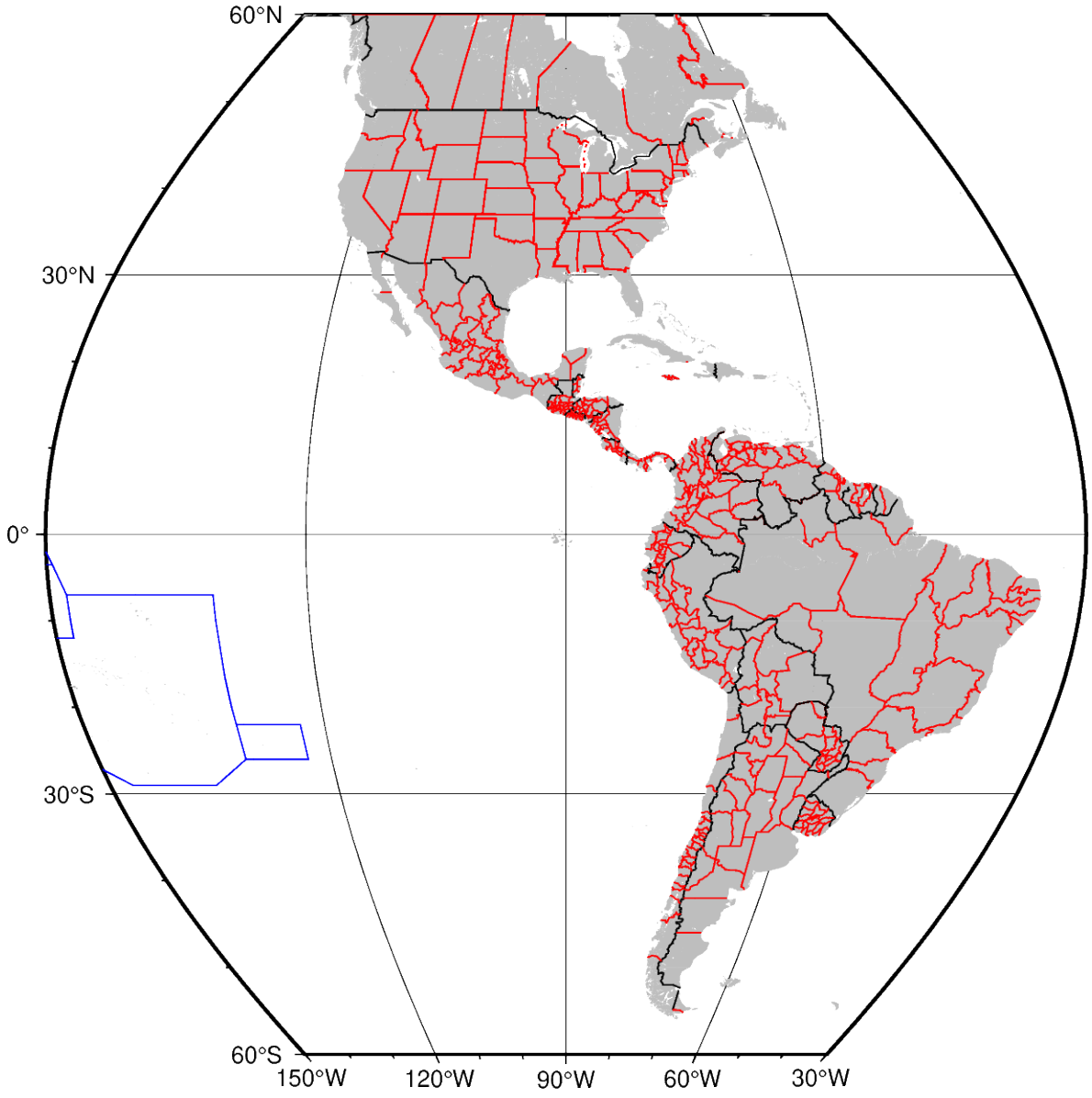
Total running time of the script: (0 minutes 7.174 seconds)

5.1.4 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 1 point and black line color use `borders="1/1p,black"`. You can draw multiple boundaries by passing in a list to `borders`.

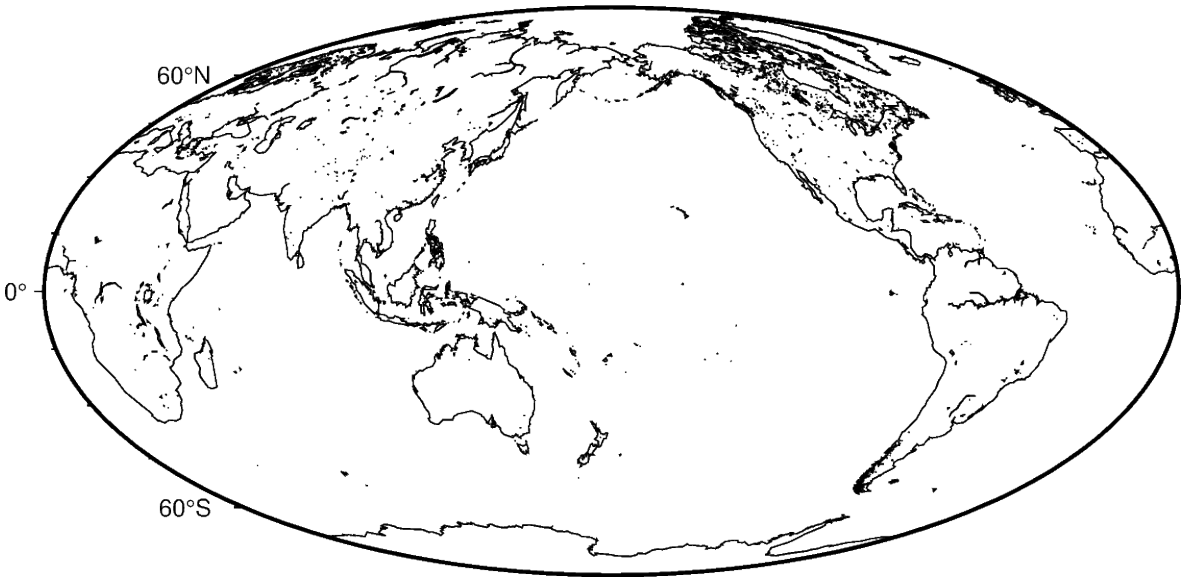



```
import pygmt

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 0.362 seconds)

5.1.5 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 0.232 seconds)

5.1.6 Tile maps

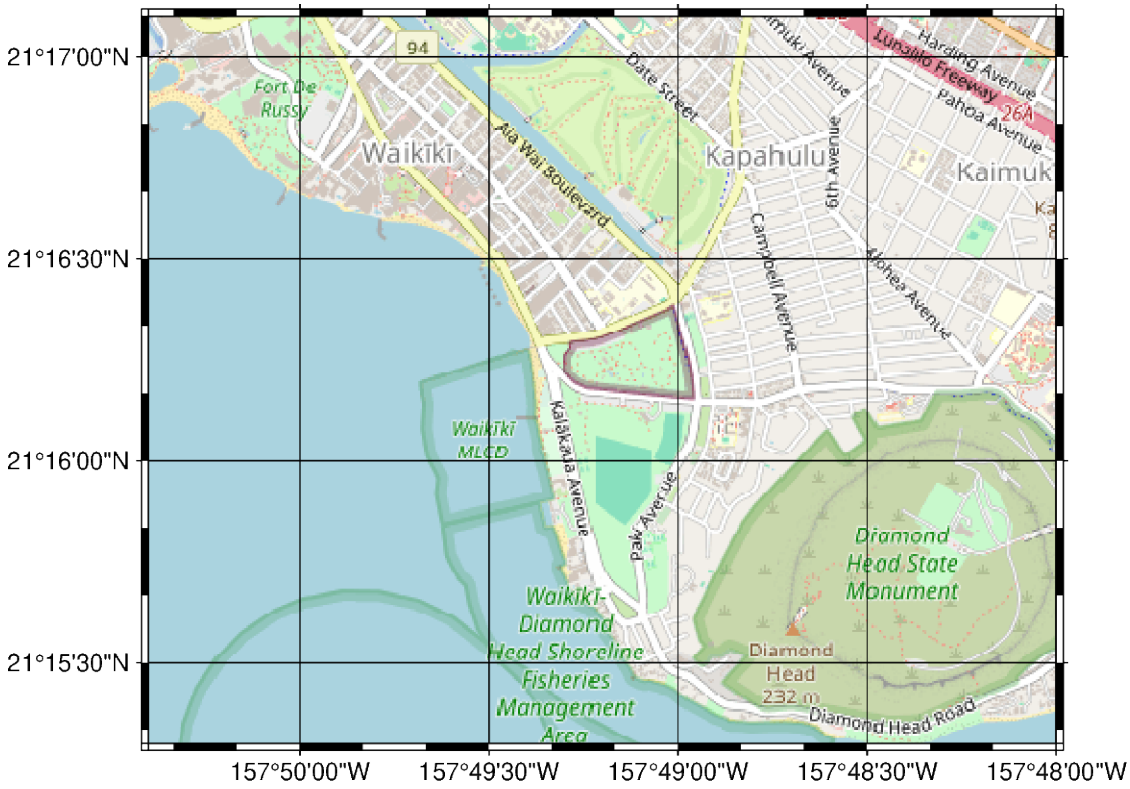
The *pygmt.Figure.tilemap* method allows to plot tiles from a tile server or local file as a basemap or overlay.

```
import contextily
import pygmt

fig = pygmt.Figure()
fig.tilemap(
    region=[-157.84, -157.8, 21.255, 21.285],
    projection="M12c",
    # Set level of details (0-22)
    # Higher levels mean a zoom level closer to the Earth's
    # surface with more tiles covering a smaller
    # geographic area and thus more details and vice versa
    # Please note, not all zoom levels are always available
    zoom=14,
    # Use tiles from OpenStreetMap tile server
    source="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png",
    frame="afg",
```

(continues on next page)

(continued from previous page)

```
)

fig.show()
```




```
grdimage [WARNING]: (w - x_min) must equal (NX + eps) * x_inc), where NX is an␣
↪integer and |eps| <= 0.0001.
grdimage [WARNING]: w reset from -157.84 to -157.840070056
grdimage [WARNING]: (e - x_min) must equal (NX + eps) * x_inc), where NX is an␣
↪integer and |eps| <= 0.0001.
grdimage [WARNING]: e reset from -157.8 to -157.799990801
grdimage [WARNING]: (s - y_min) must equal (NY + eps) * y_inc), where NY is an␣
↪integer and |eps| <= 0.0001.
grdimage [WARNING]: s reset from 21.255 to 21.2549765742
grdimage [WARNING]: (n - y_min) must equal (NY + eps) * y_inc), where NY is an␣
↪integer and |eps| <= 0.0001.
grdimage [WARNING]: n reset from 21.285 to 21.2850360149
```

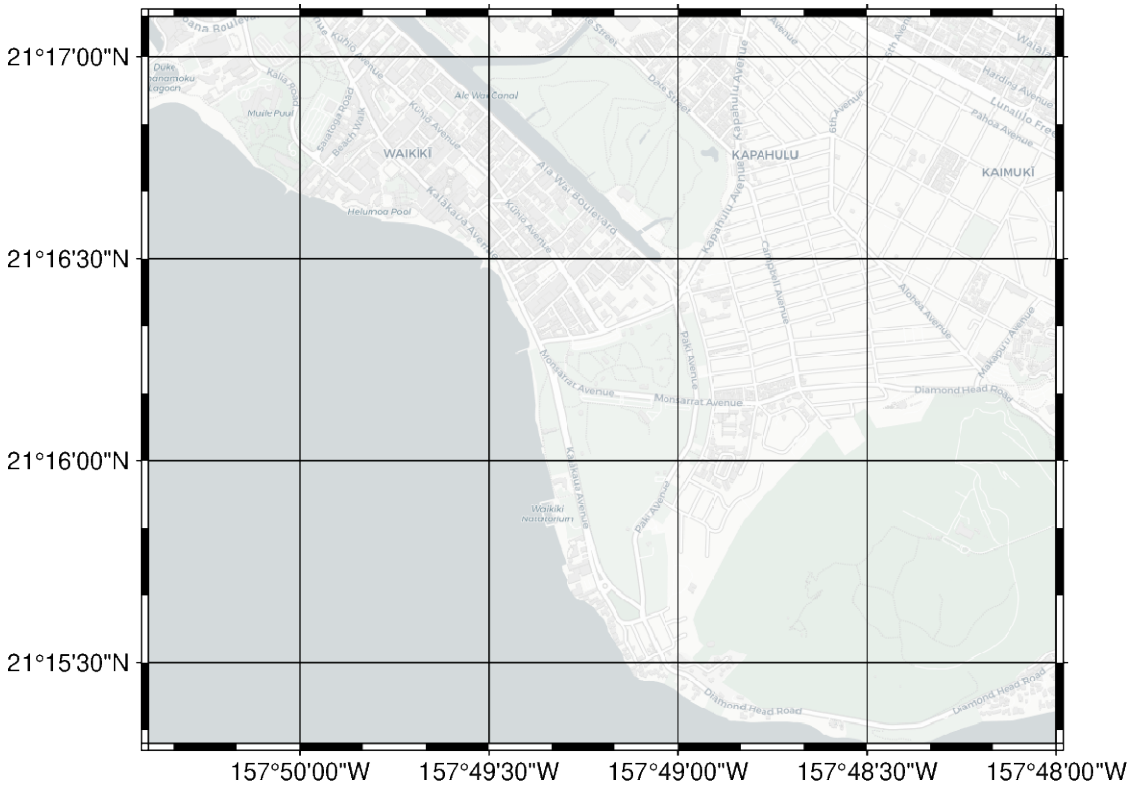
It's also possible to use tiles provided via the contextily library. See Contextily providers for a list of possible tilemap options.

```
fig = pygmt.Figure()
fig.tilemap(
    region=[-157.84, -157.8, 21.255, 21.285],
    projection="M12c",
    # Use the CartoDB Positron option from contextily
    source=contextily.providers.CartoDB.Positron,
    frame="afg",
```

(continues on next page)

(continued from previous page)

```
)

fig.show()
```



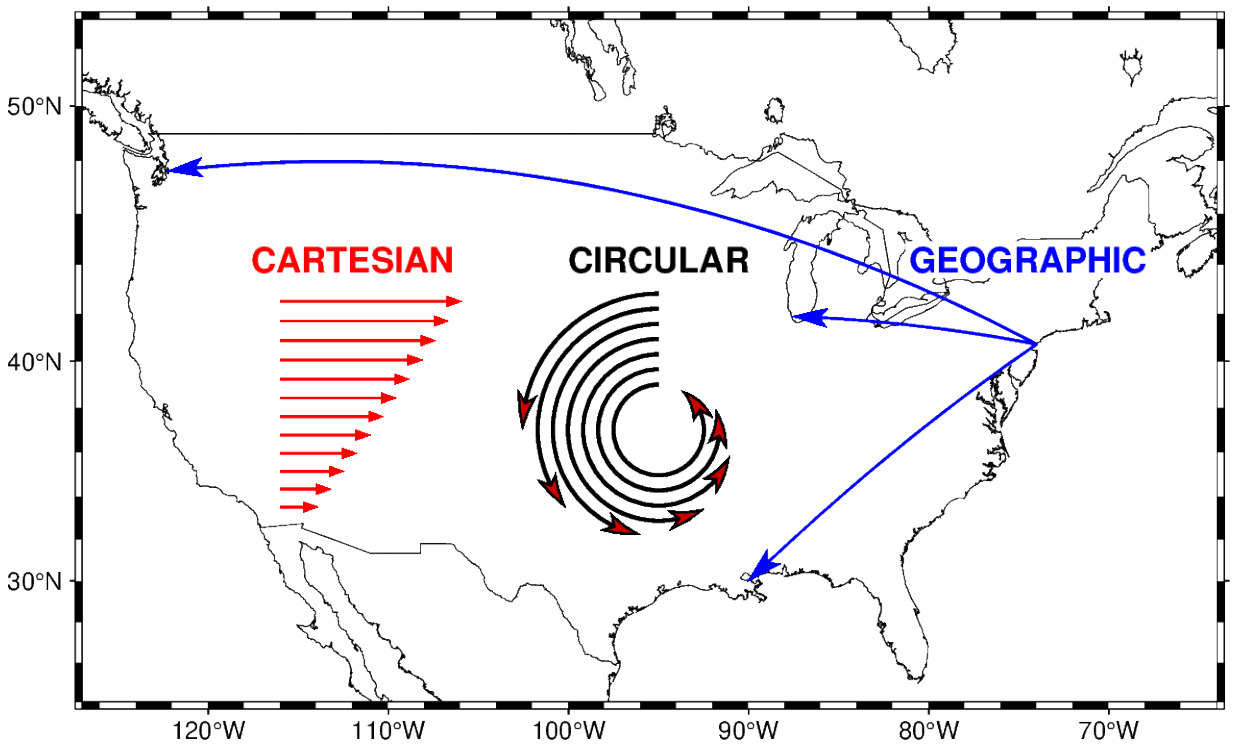

```
grdimage [WARNING]: (w - x_min) must equal (NX + eps) * x_inc), where NX is an␣
↪integer and |eps| <= 0.0001.
grdimage [WARNING]: w reset from -157.84 to -157.840016369
grdimage [WARNING]: (e - x_min) must equal (NX + eps) * x_inc), where NX is an␣
↪integer and |eps| <= 0.0001.
grdimage [WARNING]: e reset from -157.8 to -157.799990966
grdimage [WARNING]: (s - y_min) must equal (NY + eps) * y_inc), where NY is an␣
↪integer and |eps| <= 0.0001.
grdimage [WARNING]: s reset from 21.255 to 21.2549741829
grdimage [WARNING]: (n - y_min) must equal (NY + eps) * y_inc), where NY is an␣
↪integer and |eps| <= 0.0001.
grdimage [WARNING]: n reset from 21.285 to 21.2850347554
```

Total running time of the script: (0 minutes 1.254 seconds)

5.2 Lines and vectors

5.2.1 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 the *Vector attributes example*.




```
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 fill and pen (+g, +p), vector head at the end (+e),␣
↪and
# 40 degree angle (+a) with no indentation for the 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
↪")
```

(continues on next page)

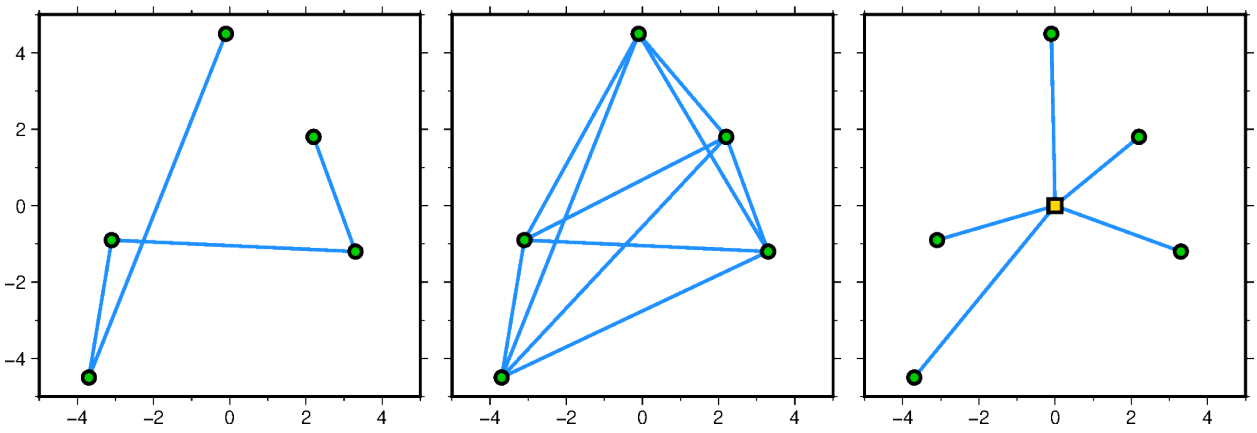
(continued from previous page)

```
# 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 the end
fig.plot(data=data, style=arcstyle, fill="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 0.299 seconds)

5.2.2 Connection lines

The `connection` parameter of the *pygmt.Figure.plot* method allows to plot connection lines between a set of data points. Width, color, and style of the lines can be adjusted via the `pen` parameter. The data points must be plotted separately using the `style` parameter, with adjustments for the symbol fill and outline via the `fill` and `pen` parameters, respectively.



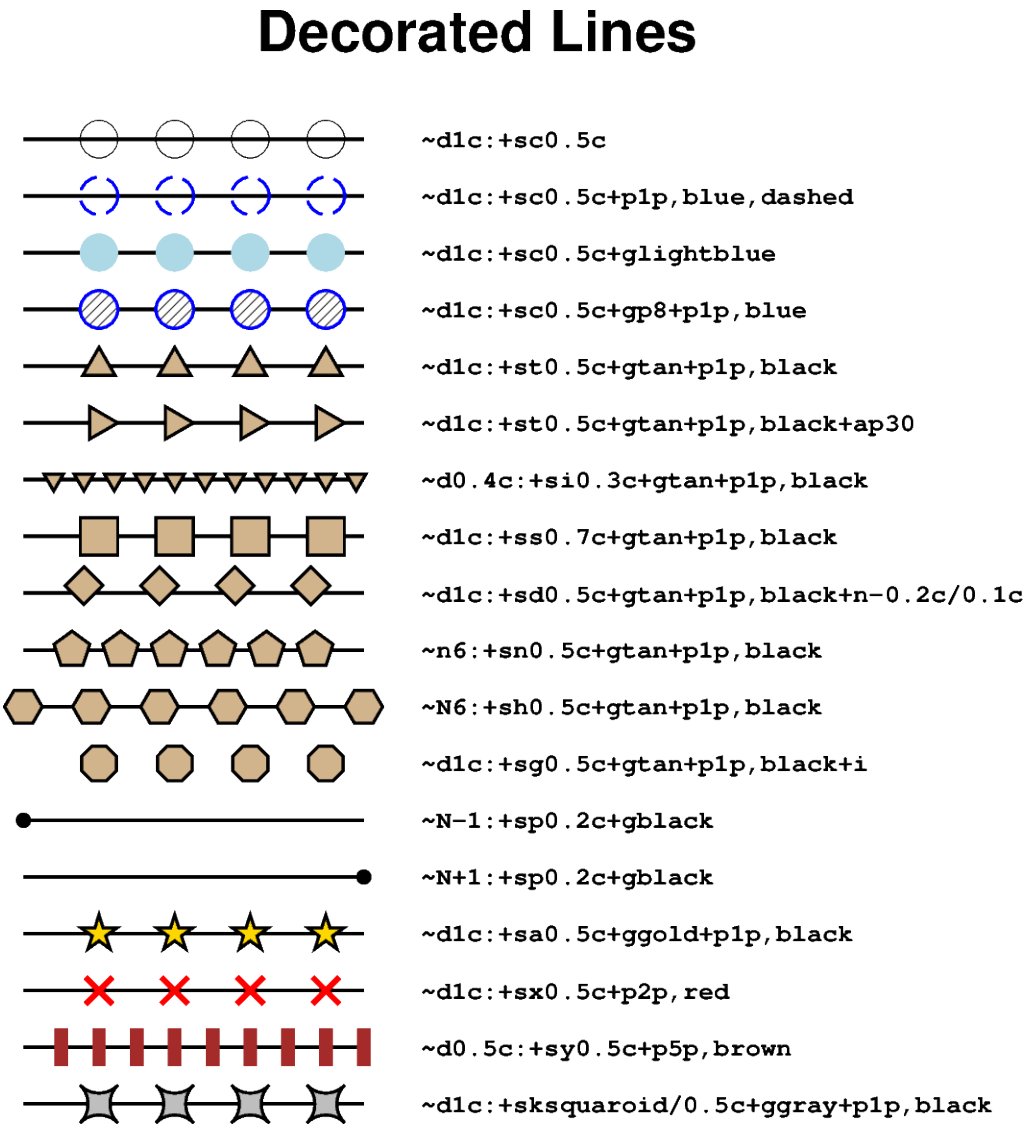
```
import pygmt

# Set up same sample data
x = [2.2, 3.3, -3.1, -3.7, -0.1]
y = [1.8, -1.2, -0.9, -4.5, 4.5]

# Create new Figure instance
fig = pygmt.Figure()

# ------------------------------------------------------------------------------
# Left: record order
fig.basemap(region=[-5, 5, -5, 5], projection="X6c", frame=["WSne", "af"])

# Connect data points based on the record order [Default connection=None]
fig.plot(x=x, y=y, pen="1.5p,dodgerblue")
# Plot data points
fig.plot(x=x, y=y, style="c0.2c", fill="green3", pen="1.5p")

fig.shift_origin(xshift="w+0.5c")

# ------------------------------------------------------------------------------
# Middle: network
fig.basemap(region=[-5, 5, -5, 5], projection="X6c", frame=["wSne", "af"])

# Connect data points as network
fig.plot(x=x, y=y, pen="1.5p,dodgerblue", connection="n")
# Plot data points
fig.plot(x=x, y=y, style="c0.2c", fill="green3", pen="1.5p")

fig.shift_origin(xshift="w+0.5c")

# ------------------------------------------------------------------------------
# Right: reference point
fig.basemap(region=[-5, 5, -5, 5], projection="X6c", frame=["wSne", "af"])

# Connect data points with the reference point (0,0)
fig.plot(x=x, y=y, pen="1.5p,dodgerblue", connection="p0/0")
# Plot data points
fig.plot(x=x, y=y, style="c0.2c", fill="green3", pen="1.5p")
# Plot reference point
fig.plot(x=0, y=0, style="s0.3c", fill="gold", pen="1.5p")

fig.show()
```

Total running time of the script: (0 minutes 0.160 seconds)

5.2.3 Decorated lines

To draw a so-called *decorated line*, i.e., symbols along a line or curve, use the `style` parameter of the *pygmt.Figure.plot* method with the argument `"~"` and the desired modifiers. A colon (`":"`) is used to separate the algorithm settings from the symbol information. This example shows how to adjust the symbols. Beside the built-in symbols also custom symbols can be used. For modifying the main decorated line via the `pen` parameter, see the *Line styles example*. For details on the input data see the upstream GMT documentation at https://docs.generic-mapping-tools.org/6.5/plot.html#s.

Furthermore, there are so-called *line fronts*, which are often used to plot fault lines, subduction zones, or weather fronts; for details see the *Line fronts example*.




```
import numpy as np
import pygmt

# Generate a two-point line for plotting
x = np.array([1, 4])
y = np.array([24, 24])

fig = pygmt.Figure()
fig.basemap(region=[0, 10, 0, 24], projection="X15c", frame="+tDecorated Lines")

# Plot different decorated lines
for decoline in [
    # Line with circles ("c") of 0.5 centimeters radius in distance of 1 centimeter
    "~d1c:+sc0.5c",
    # Adjust thickness, color, and style of the outline via "+p"
```

(continues on next page)

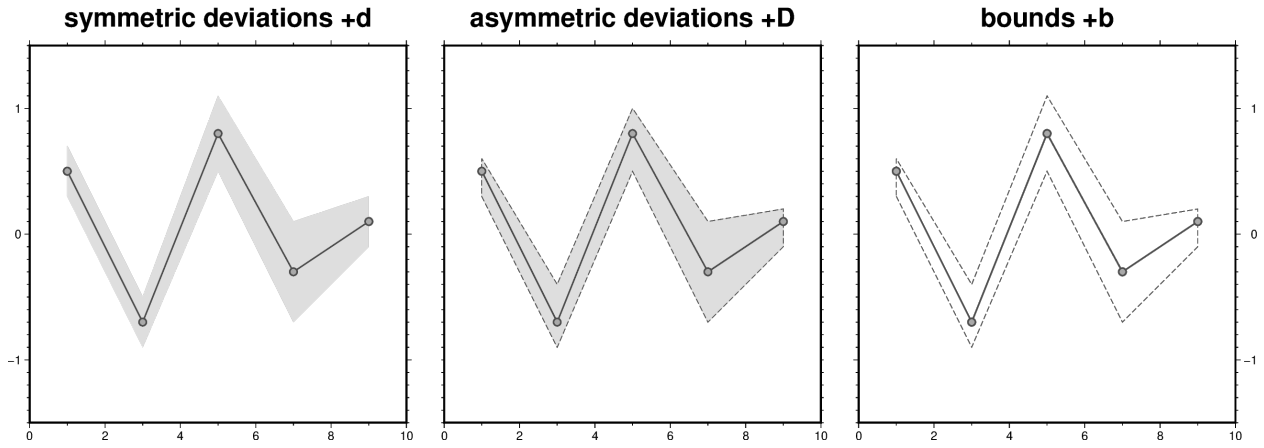
(continued from previous page)

```
    # Here, we plot a 1-point thick, blue, dashed outline
    "~d1c:+sc0.5c+p1p,blue,dashed",
    # Add a fill color using "+g" with the desired color
    "~d1c:+sc0.5c+glightblue",
    # To use a pattern as fill append "p" and give the pattern number
    "~d1c:+sc0.5c+gp8+p1p,blue",
    # Line with triangles ("t")
    "~d1c:+st0.5c+gtan+p1p,black",
    # Rotate counter-clockwise from line-parallel ("+ap") by 30 degrees
    "~d1c:+st0.5c+gtan+p1p,black+ap30",
    # Line with inverse triangles with a size of 0.3 centimeters in a
    # distance of 0.4 centimeters
    "~d0.4c:+si0.3c+gtan+p1p,black",
    # Line with squares ("s") with a size of 0.7 centimeters in a distance of
    # 1 centimeter
    "~d1c:+ss0.7c+gtan+p1p,black",
    # Shift symbols using "+n" in x and y directions relative to the main decorated␣
↪line
    "~d1c:+sd0.5c+gtan+p1p,black+n-0.2c/0.1c",
    # Give the number of equally spaced symbols by using "n" instead of "d"
    "~n6:+sn0.5c+gtan+p1p,black",
    # Use uppercase "N" to have symbols at the start and end of the line
    "~N6:+sh0.5c+gtan+p1p,black",
    # Suppress the main decorated line by appending "+i"
    "~d1c:+sg0.5c+gtan+p1p,black+i",
    # To only plot a symbol at the start of the line use "N-1"
    "~N-1:+sp0.2c+gblack",
    # To only plot a symbol at the end of the line use "N+1"
    "~N+1:+sp0.2c+gblack",
    # Line with stars ("a")
    "~d1c:+sa0.5c+ggold+p1p,black",
    # Line with crosses ("x")
    "~d1c:+sx0.5c+p2p,red",
    # Line with (vertical) lines or bars ("y")
    "~d0.5c:+sy0.5c+p5p,brown",
    # Use custom symbol ("k") "squaroid" with a size of 0.5 centimeters
    "~d1c:+sksquaroid/0.5c+ggray+p1p,black",
]:
    y = y - 1.2  # Move current line down
    fig.plot(x=x, y=y, style=decoline, pen="1.25p,black")
    fig.text(
        x=x[-1],
        y=y[-1],
        text=decoline,
        font="Courier-Bold",
        justify="ML",
        offset="0.75c/0c",
    )

fig.show()
```

Total running time of the script: (0 minutes 0.310 seconds)

5.2.4 Envelope

The `close` parameter of the *pygmt.Figure.plot* method can be used to build a symmetrical or an asymmetrical envelope. The user can give either the deviations or the bounds in y-direction. For the first case append `"+d"` or `"+D"` and for the latter case `"+b"`.




```
import pandas as pd
import pygmt

# Define a pandas.DataFrame with columns for x and y as well as the lower and upper
# deviations
df_devi = pd.DataFrame(
    data={
        "x": [1, 3, 5, 7, 9],
        "y": [0.5, -0.7, 0.8, -0.3, 0.1],
        "y_deviation_low": [0.2, 0.2, 0.3, 0.4, 0.2],
        "y_deviation_upp": [0.1, 0.3, 0.2, 0.4, 0.1],
    }
)

# Define the same pandas.DataFrame but with lower and upper bounds
df_bound = pd.DataFrame(
    data={
        "x": [1, 3, 5, 7, 9],
        "y": [0.5, -0.7, 0.8, -0.3, 0.1],
        "y_bound_low": [0.3, -0.9, 0.5, -0.7, -0.1],
        "y_bound_upp": [0.6, -0.4, 1.1, 0.1, 0.2],
    }
)


fig = pygmt.Figure()

# --------------------------------------------------------------------------------
# Left
fig.basemap(
    region=[0, 10, -1.5, 1.5],
    projection="X10c",
    frame=["WSne+tsymmetric deviations +d", "xa2f1", "ya1f0.1"],
)
```

(continues on next page)

(continued from previous page)

```
# Plot a symmetrical envelope based on the deviations ("+d")
fig.plot(
    data=df_devi,
    close="+d",
    # Fill the envelope in gray color with a transparency of 50 %
    fill="gray@50",
    pen="1p,gray30",
)

# Plot the data points on top
fig.plot(data=df_devi, style="c0.2c", pen="1p,gray30", fill="darkgray")

# Shift plot origin by the figure width ("w") plus 1 centimeter in x direction
fig.shift_origin(xshift="w+1c")

# ------------------------------------------------------------------------------
# Middle
fig.basemap(
    region=[0, 10, -1.5, 1.5],
    projection="X10c",
    frame=["WSne+tasymmetric deviations +D", "xa2f1", "yf0.1"],
)

# Plot an asymmetrical envelope based on the deviations ("+D")
fig.plot(
    data=df_devi,
    fill="gray@50",
    # Add an outline around the envelope. Here, a dashed pen ("+p") with 0.5-points
    # thickness and "gray30" color is used
    close="+D+p0.5p,gray30,dashed",
    pen="1p,gray30",
)

fig.plot(data=df_devi, style="c0.2c", pen="1p,gray30", fill="darkgray")

fig.shift_origin(xshift="w+1c")

# ------------------------------------------------------------------------------
# Right
fig.basemap(
    region=[0, 10, -1.5, 1.5],
    projection="X10c",
    # Use "\\053" to handle "+b" as a string not as a modifier
    frame=["wSnE+tbounds \\053b", "xa2f1", "ya1f0.1"],
)

# Plot an envelope based on the bounds ("+b")
fig.plot(data=df_bound, close="+b+p0.5p,gray30,dashed", pen="1p,gray30")

fig.plot(data=df_bound, style="c0.2c", pen="1p,gray30", fill="darkgray")
```

(continues on next page)

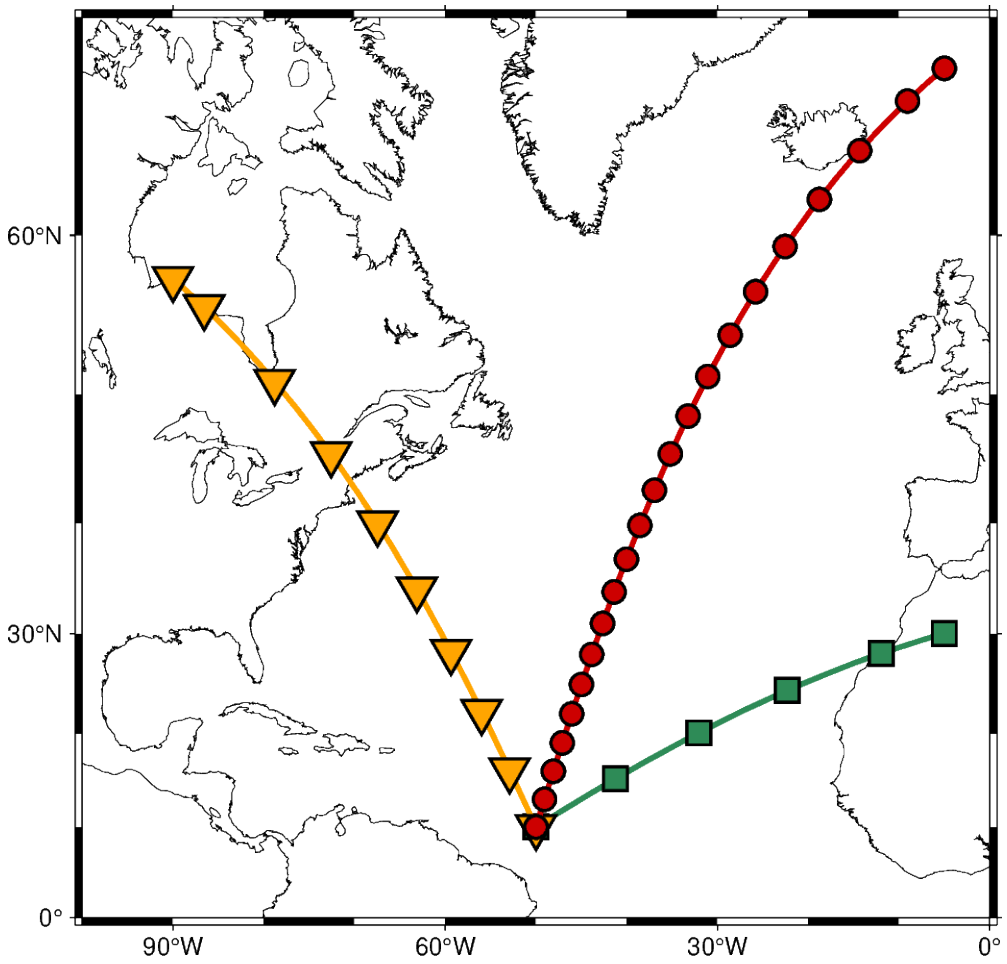
(continued from previous page)

```
fig.show()
```

Total running time of the script: (0 minutes 0.312 seconds)

5.2.5 Generate points along great circles

The *pygmt.project* function can generate points along a great circle whose center and end points can be defined via the `center` and `endpoint` parameters, respectively. Using the `generate` parameter allows to generate (*r*, *s*, *p*) points every *dist* units of *p* along a profile as output. By default, all units (*r*, *s* and *p*) are set to degrees while `unit=True` allows to set the unit for *p* to km.




```
import pygmt

fig = pygmt.Figure()

# generate points every 10 degrees along a great circle from 10N,50W to 30N,5W
points1 = pygmt.project(center=[-50, 10], endpoint=[-5, 30], generate=10)
# generate points every 750 km along a great circle from 10N,50W to 57.5N,90W
points2 = pygmt.project(center=[-50, 10], endpoint=[-90, 57.5], generate=750,␣
↪unit=True)
# generate points every 350 km along a great circle from 10N,50W to 68N,5W
points3 = pygmt.project(center=[-50, 10], endpoint=[-5, 68], generate=350, unit=True)
```

(continues on next page)

(continued from previous page)

```
# create a plot with coast and Mercator projection (M)
fig.basemap(region=[-100, 0, 0, 70], projection="M12c", frame=True)
fig.coast(shorelines=True, area_thresh=5000)

# plot individual points of first great circle as seagreen line
fig.plot(x=points1.r, y=points1.s, pen="2p,seagreen")
# plot individual points as seagreen squares atop
fig.plot(x=points1.r, y=points1.s, style="s.45c", fill="seagreen", pen="1p")

# plot individual points of second great circle as orange line
fig.plot(x=points2.r, y=points2.s, pen="2p,orange")
# plot individual points as orange inverted triangles atop
fig.plot(x=points2.r, y=points2.s, style="i.6c", fill="orange", pen="1p")

# plot individual points of third great circle as red3 line
fig.plot(x=points3.r, y=points3.s, pen="2p,red3")
# plot individual points as red3 circles atop
fig.plot(x=points3.r, y=points3.s, style="c.3c", fill="red3", pen="1p")

fig.show()
```

Total running time of the script: (0 minutes 0.214 seconds)

5.2.6 GeoPandas: Plotting lines with LineString or MultiLineString geometry

The *pygmt.Figure.plot* method allows us to plot geographical data such as lines with LineString or MultiLineString geometry types stored in a `geopandas.GeoDataFrame` object or any object that implements the __geo_interface__ property.

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` object as an argument to the `data` parameter of *pygmt.Figure.plot*, and style the lines using the `pen` parameter.

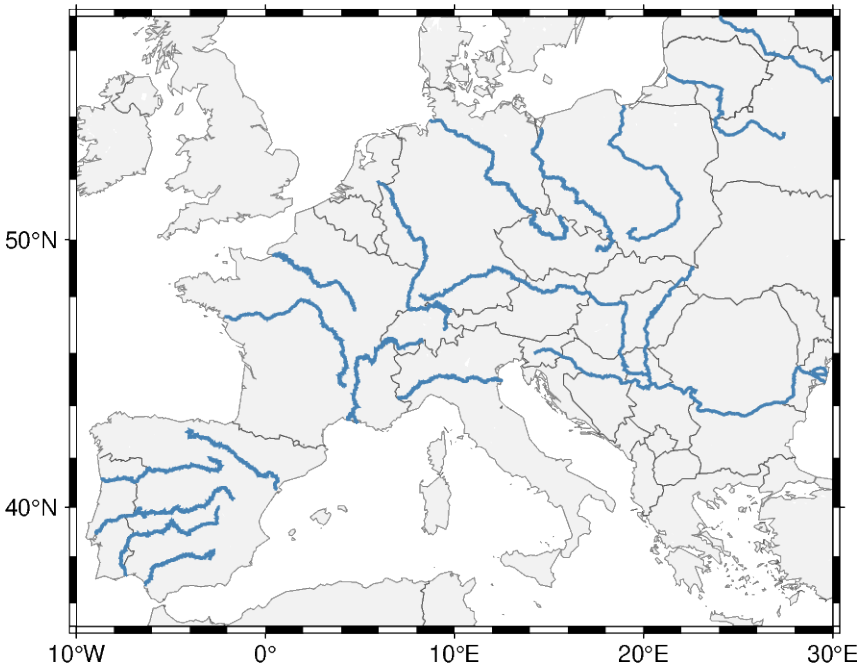
```
import geodatasets
import geopandas as gpd
import pygmt

# Read a sample dataset provided by the geodatasets package.
# The dataset contains large rivers in Europe, stored as LineString/MultiLineString
# geometry types.
gdf = gpd.read_file(geodatasets.get_path("eea large_rivers"))

# Convert object to EPSG 4326 coordinate system
gdf = gdf.to_crs("EPSG:4326")
gdf.head()
```

```
fig = pygmt.Figure()

fig.coast(
    projection="M10c",
    region=[-10, 30, 35, 57],
```

(continues on next page)

(continued from previous page)

```
    resolution="l",
    land="gray95",
    shorelines="1/0.1p,gray50",
    borders="1/0.1,gray30",
    frame=True,
)

# Add rivers to map
fig.plot(data=gdf, pen="1p,steelblue")

fig.show()
```



Total running time of the script: (0 minutes 2.870 seconds)

5.2.7 Horizontal and vertical lines

The *pygmt.Figure.hlines* and *pygmt.Figure.vlines* methods allow to plot horizontal and vertical lines in Cartesian, geographic and polar coordinate systems.

Cartesian coordinate system

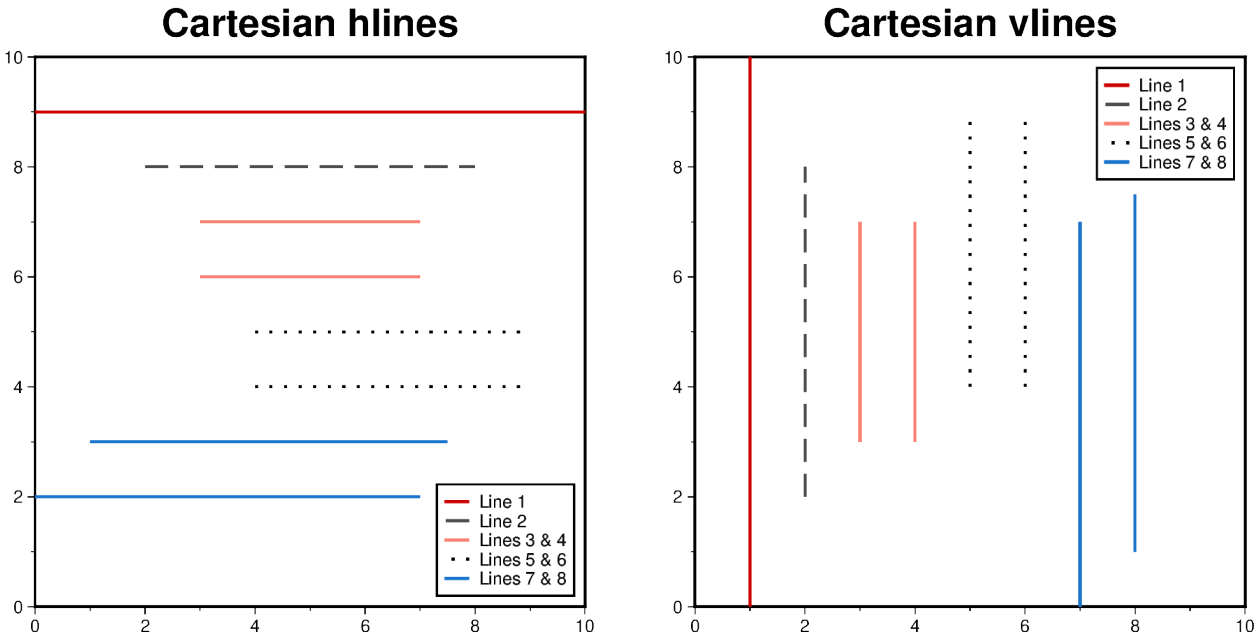
In Cartesian coordinate systems lines are plotted as straight lines.

```
import pygmt

fig = pygmt.Figure()

fig.basemap(
    region=[0, 10, 0, 10], projection="X10c/10c", frame=["+tCartesian hlines", "af"]
)
```

(continues on next page)

(continued from previous page)

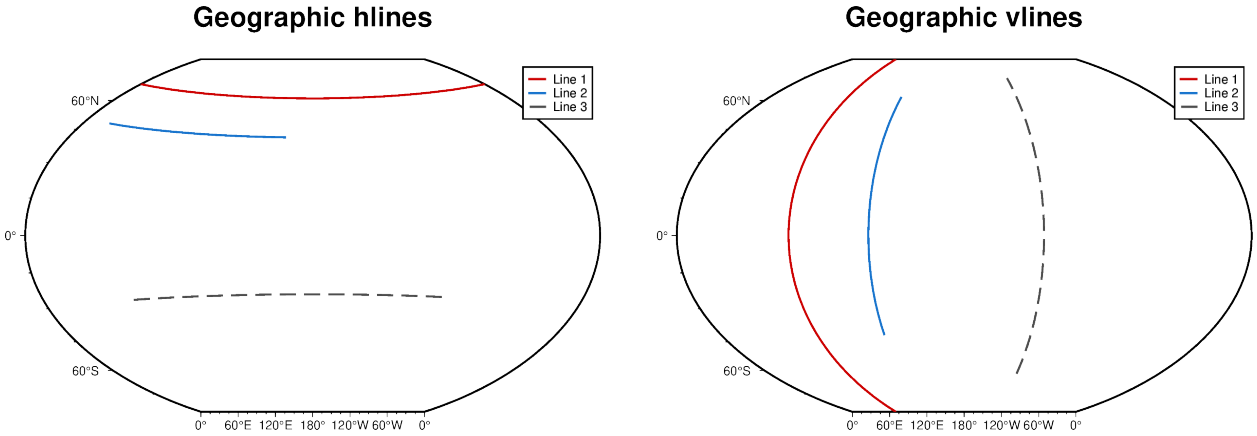
```
# Add a horizontal line at y=9
fig.hlines(y=9, pen="1.5p,red3", label="Line 1")
# Add a horizontal line at y=8 with x from 2 to 8
fig.hlines(y=8, xmin=2, xmax=8, pen="1.5p,gray30,-", label="Line 2")
# Add two horizontal lines at y=6 and y=7 both with x from 3 to 7
fig.hlines(y=[6, 7], xmin=3, xmax=7, pen="1.5p,salmon", label="Lines 3 & 4")
# Add two horizontal lines at y=4 and y=5 both with x from 4 to 9
fig.hlines(y=[4, 5], xmin=4, xmax=9, pen="1.5p,black,.", label="Lines 5 & 6")
# Add two horizontal lines at y=2 and y=3 with different x limits
fig.hlines(
    y=[2, 3], xmin=[0, 1], xmax=[7, 7.5], pen="1.5p,dodgerblue3", label="Lines 7 & 8"
)
fig.legend(position="JBR+jBR+o0.2c", box="+gwhite+p1p")

fig.shift_origin(xshift="w+2c")

fig.basemap(
    region=[0, 10, 0, 10], projection="X10c/10c", frame=["+tCartesian vlines", "af"]
)
# Add a vertical line at x=1
fig.vlines(x=1, pen="1.5p,red3", label="Line 1")
# Add a vertical line at x=2 with y from 2 to 8
fig.vlines(x=2, ymin=2, ymax=8, pen="1.5p,gray30,-", label="Line 2")
# Add two vertical lines at x=3 and x=4 both with y from 3 to 7
fig.vlines(x=[3, 4], ymin=3, ymax=7, pen="1.5p,salmon", label="Lines 3 & 4")
# Add two vertical lines at x=5 and x=6 both with y from 4 to 9
fig.vlines(x=[5, 6], ymin=4, ymax=9, pen="1.5p,black,.", label="Lines 5 & 6")
# Add two vertical lines at x=7 and x=8 with different y limits
fig.vlines(
    x=[7, 8], ymin=[0, 1], ymax=[7, 7.5], pen="1.5p,dodgerblue3", label="Lines 7 & 8"
)
fig.legend()

fig.show()
```



Geographic coordinate system

The same can be done in geographic coordinate systems where "horizontal" means lines are plotted along parallels (constant latitude) while "vertical" means lines are plotted along meridians (constant longitude).

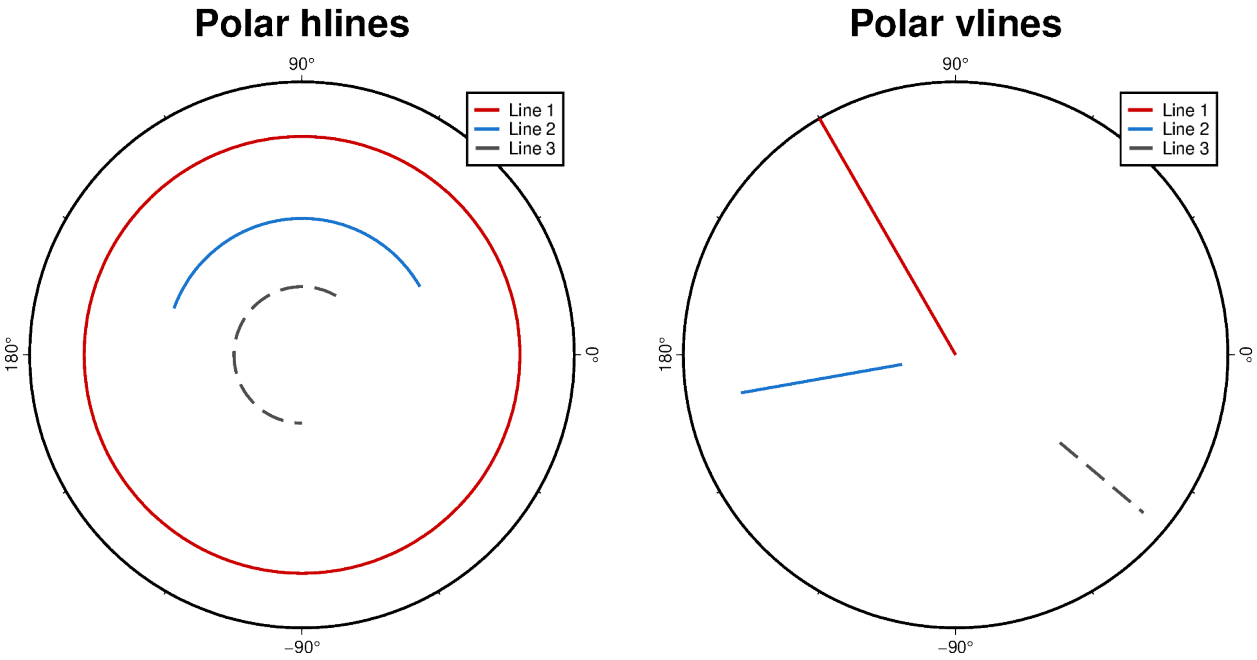
```
fig = pygmt.Figure()

fig.basemap(region="g", projection="R15c", frame=["+tGeographic hlines", "af"])
# Add a line at 70°N
fig.hlines(y=70, pen="1.5p,red3", label="Line 1")
# Add a line at 50°N with longitude limits at 20°E and 160°E
fig.hlines(y=50, xmin=20, xmax=160, pen="1.5p,dodgerblue3", label="Line 2")
# Add a line at 30°S with longitude limits at 60°E and 270°E
fig.hlines(y=-30, xmin=60, xmax=270, pen="1.5p,gray30,-", label="Line 3")
fig.legend()

fig.shift_origin(xshift="w+2c")

fig.basemap(region="g", projection="R15c", frame=["+tGeographic vlines", "af"])
# Add a line at 70°E
fig.vlines(x=70, pen="1.5p,red3", label="Line 1")
# Add a line at 20°E with latitude limits at 50°S and 70°N
fig.vlines(x=120, ymin=-50, ymax=70, pen="1.5p,dodgerblue3", label="Line 2")
# Add a line at 230°E with latitude limits at 70°S and 80°N
fig.vlines(x=230, ymin=-70, ymax=80, pen="1.5p,gray30,-", label="Line 3")
fig.legend()

fig.show()
```



Polar coordinate system

When using polar coordinate systems "horizontal" means lines are plotted as arcs along a constant radius while "vertical" means lines are plotted as straight lines along radius at a specified azimuth.

```
fig = pygmt.Figure()

fig.basemap(region=[0, 360, 0, 1], projection="P10c", frame=["+tPolar hlines", "af"])
# Add a line along radius=0.8
fig.hlines(y=0.8, pen="1.5p,red3", label="Line 1")
# Add a line along radius=0.5 with azimuth limits at 30° and 160°
fig.hlines(y=0.5, xmin=30, xmax=160, pen="1.5p,dodgerblue3", label="Line 2")
# Add a line along radius=0.25 with azimuth limits at 60° and 270°
fig.hlines(y=0.25, xmin=60, xmax=270, pen="1.5p,gray30,-", label="Line 3")
fig.legend()

fig.shift_origin(xshift="w+2c")

fig.basemap(region=[0, 360, 0, 1], projection="P10c", frame=["+tPolar vlines", "af"])
# Add a line along azimuth=120°
fig.vlines(x=120, pen="1.5p,red3", label="Line 1")
# Add a line along azimuth=190° with radius limits at 0.2 and 0.8
fig.vlines(x=190, ymin=0.2, ymax=0.8, pen="1.5p,dodgerblue3", label="Line 2")
# Add a line along azimuth=320 with radius limits at 0.5 and 0.9
fig.vlines(x=320, ymin=0.5, ymax=0.9, pen="1.5p,gray30,-", label="Line 3")
fig.legend()

fig.show()
```

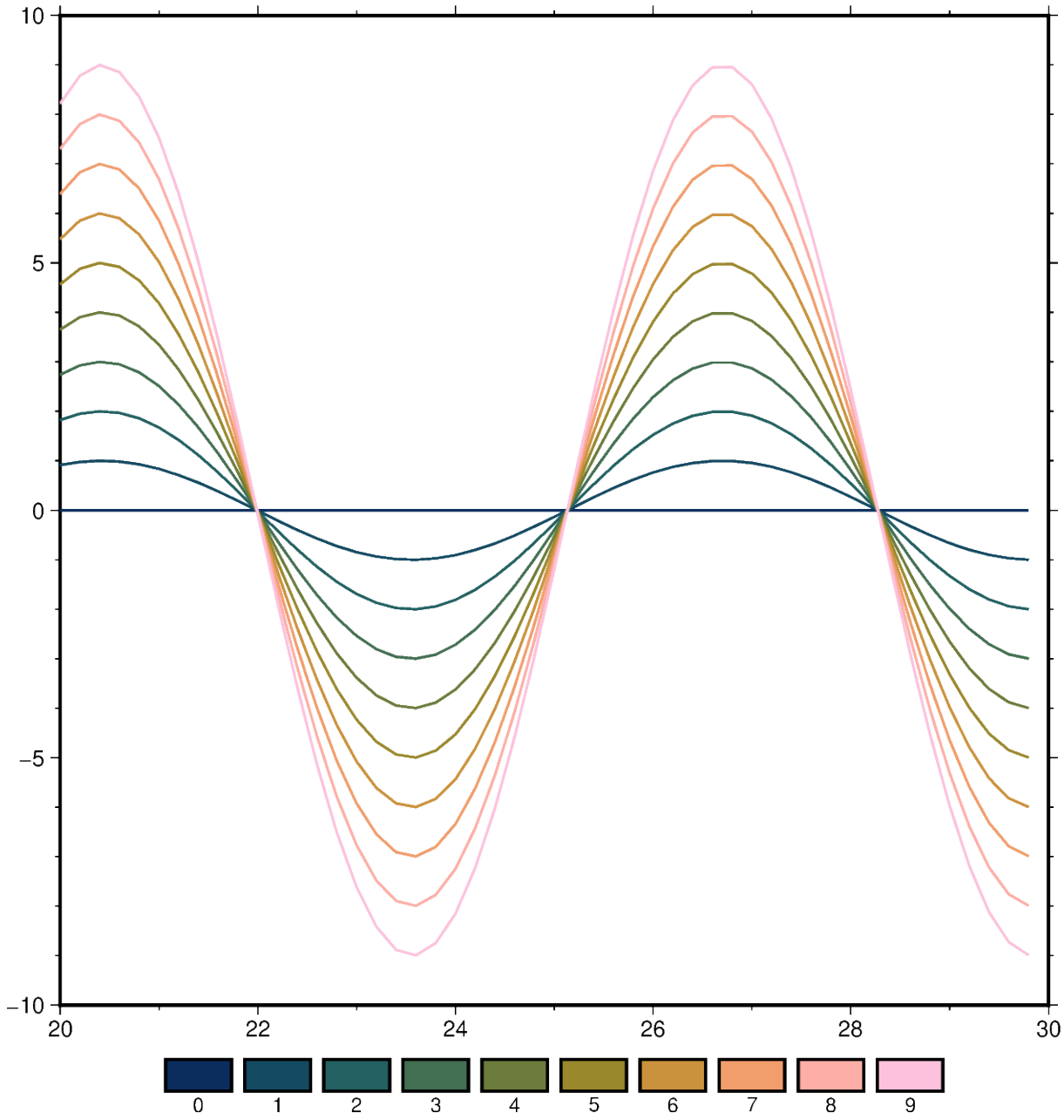


Total running time of the script: (0 minutes 0.808 seconds)

5.2.8 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.




```
import numpy as np
import pygmt

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

(continues on next page)

(continued from previous page)

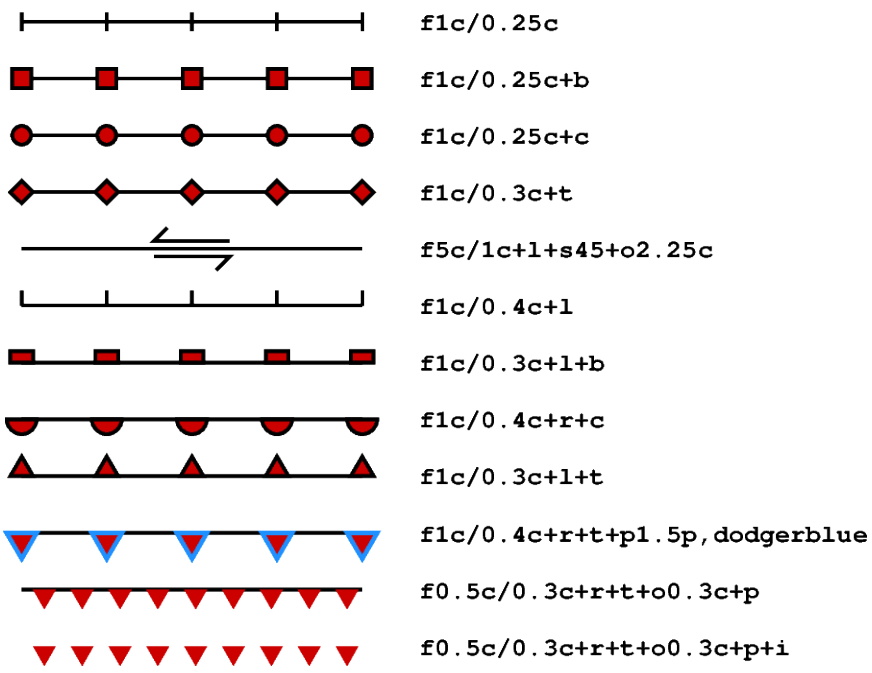
```
pygmt.makecpt(cmap="batlow", series=[0, 9, 1], color_model="+c0-9")

# Plot 10 lines and set a different z-value for each line
for zvalue in range(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 0.226 seconds)

5.2.9 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. To draw general symbols along a line or curve, i.e., a so-called *decorated line*, see the *Decorated lines example*.

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 degrees]. 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**. For modifying the main front line via the `pen` parameter, see the *Line styles example*.




```
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="+tLine 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",
    # line with left-lateral ("+l") slip ("+s") front style, angle is set to
    # 45 degrees 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
```

(continues on next page)

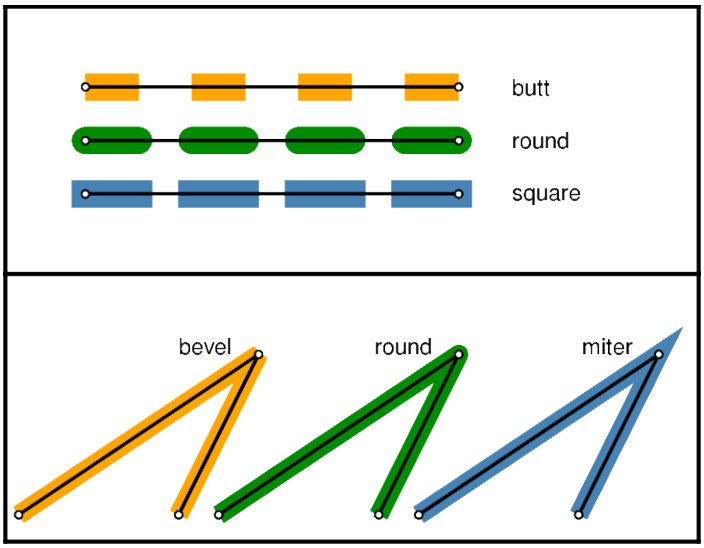
(continued from previous page)

```
    "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, fill="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 0.205 seconds)

5.2.10 Line segment caps and joints

PyGMT offers different appearances of line segment caps and joints. The desired appearance can be set via the GMT default parameters PS_LINE_CAP (`"butt"`, `"round"`, or `"square"` [Default]) as well as PS_LINE_JOIN (`"bevel"`, `"round"`, and `"miter"` [Default]) and PS_MITER_LIMIT (limit on the angle at the mitered joint below which a bevel is applied).




```
import numpy as np
import pygmt

# Set up dictionary for colors
dict_col = {
    "round": "green4",
    "square": "steelblue",
    "butt": "orange",
    "miter": "steelblue",
    "bevel": "orange",
}

# Create new Figure instance
fig = pygmt.Figure()

# ------------------------------------------------------------------------------
# Top: PS_LINE_CAP

# Create sample data
x = np.array([30, 170])
y = np.array([70, 70])

fig.basemap(region=[0, 260, 0, 100], projection="x1p", frame="rltb")

for line_cap in ["butt", "round", "square"]:
    # Change GMT default locally
    with pygmt.config(PS_LINE_CAP=line_cap):
        color = dict_col[line_cap]
        # Draw a 10-point thick line with 20-point long segments and gaps
        # Use the local PS_LINE_CAP setting
        fig.plot(x=x, y=y, pen=f"10p,{color},20_20")

    # Draw a 1-point thick black solid line to highlight segment cap appearance
    fig.plot(x=x, y=y, pen="1p,black,solid")
    # Plot data points as circles
```

(continues on next page)

(continued from previous page)

```
    fig.plot(x=x, y=y, style="c0.1c", fill="white", pen="0.5p,")
    # Add label for PS_LINE_CAP setting
    fig.text(text=line_cap, x=x[-1] + 20, y=y[-1], justify="LM")

    y = y - 20

fig.shift_origin(yshift="-h")

# ----------------------------------------------------------------------------
# Bottom: PS_LINE_JOIN and PS_MITER_LIMIT

x = np.array([5, 95, 65])
y = np.array([10, 70, 10])

fig.basemap(region=[0, 260, 0, 100], projection="x1p", frame="rltb")

for line_join in ["bevel", "round", "miter"]:
    with pygmt.config(PS_LINE_JOIN=line_join, PS_MITER_LIMIT=1):
        color = dict_col[line_join]
        # Draw a 7-point thick solid line
        # Use the local PS_LINE_JOIN and PS_MITER_LIMIT settings
        fig.plot(x=x, y=y, pen=f"7p,{color},solid")

    fig.plot(x=x, y=y, pen="1p,black,solid")
    fig.plot(x=x, y=y, style="c0.1c", fill="white", pen="0.5p")
    fig.text(text=line_join, x=x[1] - 10, y=y[1], justify="RB")

    x = x + 75

fig.show()
```

Total running time of the script: (0 minutes 0.203 seconds)

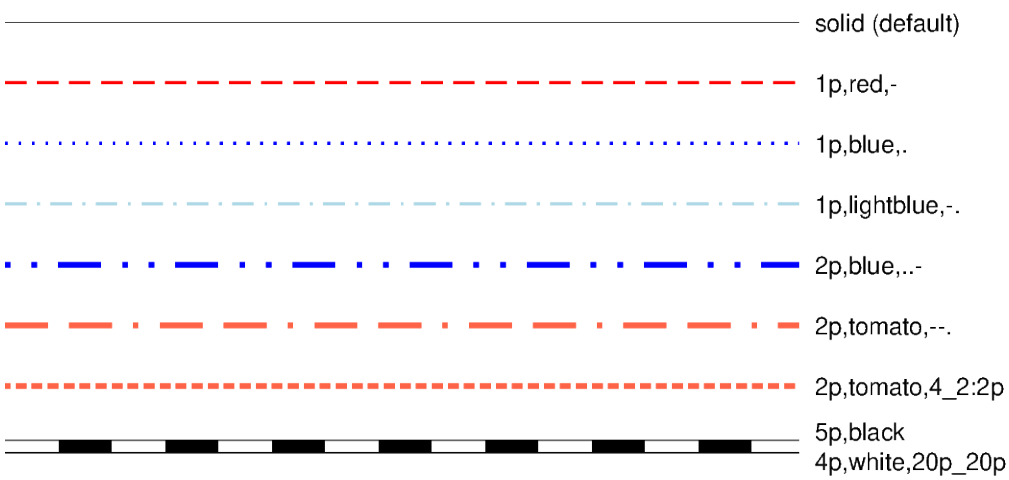
5.2.11 Line styles

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 Technical Reference https://docs.generic-mapping-tools.org/6.5/reference/features.html#wpen-attrib.

Line Styles




```
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="+tLine 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
    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")
```

(continues on next page)

(continued from previous page)

```
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 0.180 seconds)

5.2.12 Quoted lines

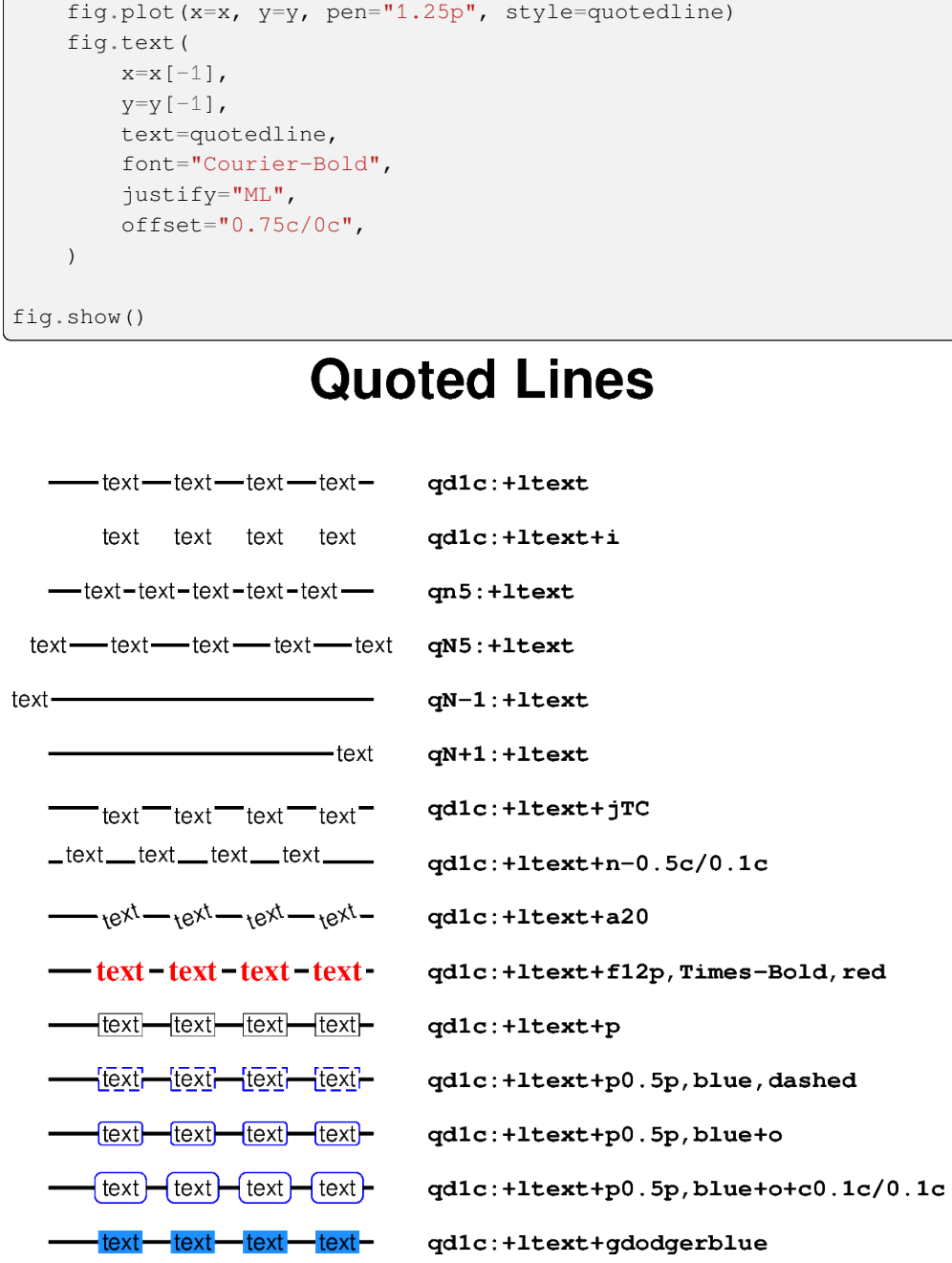
To plot a so-called *quoted line*, i.e., labels along a line or curve, use the `style` parameter of the *pygmt.Figure.plot* method with the argument `"q"` and the desired modifiers. A colon (`":"`) is used to separate the algorithm settings from the label information. This example shows how to adjust the labels. For modifying the main quoted line via the `pen` parameter, see the *Line styles example*. For details on the input data see the upstream GMT documentation at https://docs.generic-mapping-tools.org/6.5/plot.html#s.

```
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="+tQuoted Lines")

# Plot different quoted lines
for quotedline in [
    # Line with labels ("+l") "text" in distance ("d") of 1 centimeter
    "qd1c:+ltext",
    # Suppress the main quoted line by appending "+i"
    "qd1c:+ltext+i",
    # Give the number of equally spaced labels by using "n" instead of "d"
    "qn5:+ltext",
    # Use uppercase "N" to have labels at the start and end of the line
    "qN5:+ltext",
    # To only plot a label at the start of the line use "N-1"
    "qN-1:+ltext",
    # To only plot a label at the end of the line use "N+1"
    "qN+1:+ltext",
    # Adjust the justification of the labels via "+j", here Top Center
    "qd1c:+ltext+jTC",
    # Shift labels using "+n" in x and y directions relative to the main
    # quoted line
    "qd1c:+ltext+n-0.5c/0.1c",
    # Rotate labels via "+a" (counter-clockwise from horizontal)
    "qd1c:+ltext+a20",
    # Adjust size, type, and color of the font via "+f"
    "qd1c:+ltext+f12p,Times-Bold,red",
    # Add a box around the label via "+p"
    "qd1c:+ltext+p",
    # Adjust thickness, color, and style of the outline
    "qd1c:+ltext+p0.5p,blue,dashed",
```

(continues on next page)

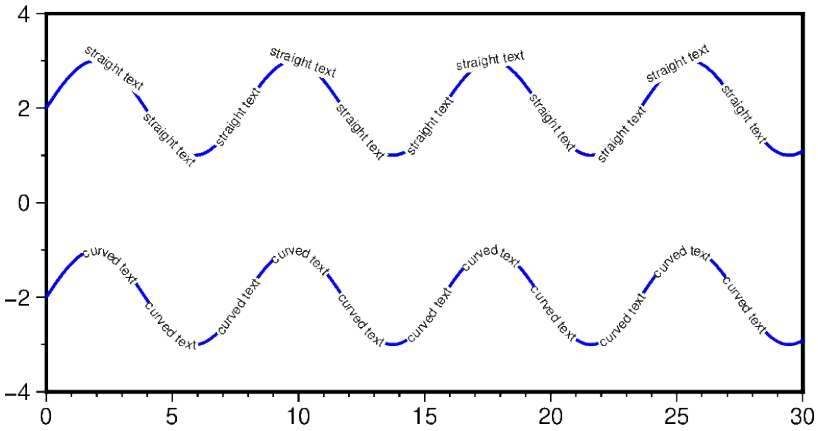
(continued from previous page)

```
    # Append "+o" to get a box with rounded edges
    "qd1c:+ltext+p0.5p,blue+o",
    # Adjust the space between label and box in x and y directions via "+c"
    "qd1c:+ltext+p0.5p,blue+o+c0.1c/0.1c",
    # Give a fill of the box via "+g" together with the desired color
    "qd1c:+ltext+gdodgerblue",
]:
    y -= 1  # Move current line down
    fig.plot(x=x, y=y, pen="1.25p", style=quotedline)
    fig.text(
        x=x[-1],
        y=y[-1],
        text=quotedline,
        font="Courier-Bold",
        justify="ML",
        offset="0.75c/0c",
    )

fig.show()
```



For curved labels following the line, append `"+v"` to the argument passed to the `style` parameter.

```
# Generate sine curve
x = np.arange(0, 10 * np.pi, 0.1)
y = np.sin(0.8 * x)

fig = pygmt.Figure()

fig.basemap(region=[0, 30, -4, 4], projection="X10c/5c", frame=True)

fig.plot(x=x, y=y + 2, style="qd1.2c:+lstraight text+f5p", pen="1p,blue")

fig.plot(
    x=x,
    y=y - 2,
    # Append "+v" to force curved labels
    style="qd1.2c:+lcurved text+f5p+v",
    pen="1p,blue",
)

fig.show()
```



Total running time of the script: (0 minutes 1.166 seconds)

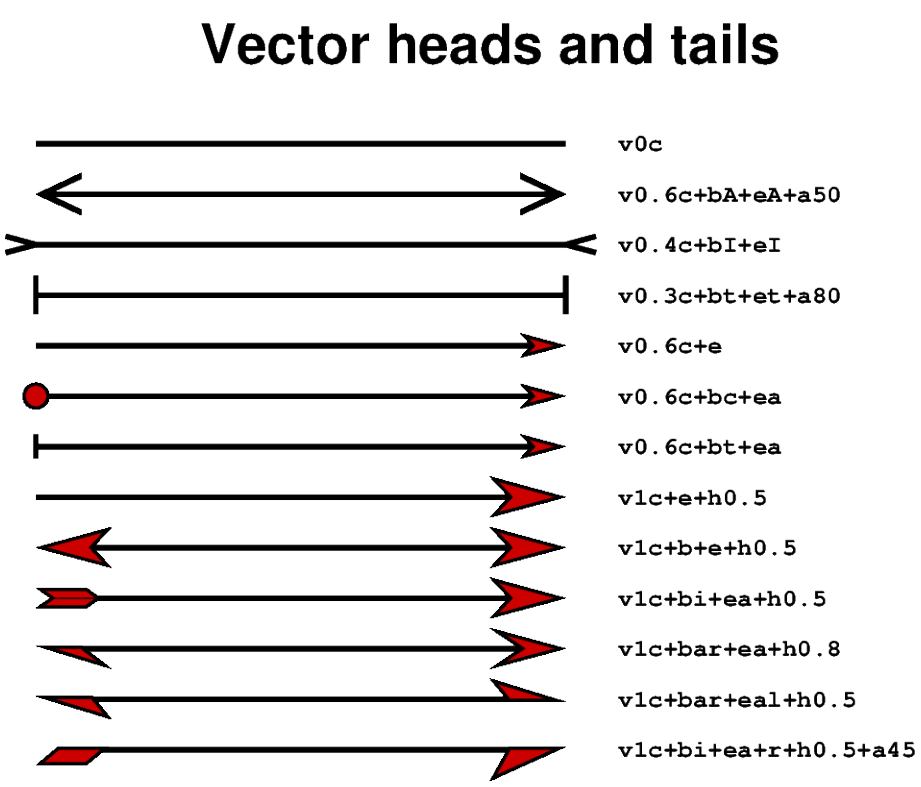
5.2.13 Vector heads and tails

Many methods 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 beginning and end. 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 https://docs.generic-mapping-tools.org/6.5/plot.html#vector-attributes.

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"` for Cartesian vector, see also the *Vector types example*) by passing the corresponding shortcuts to the `style` parameter.




```
import pygmt

fig = pygmt.Figure()
fig.basemap(
    region=[0, 10, 0, 15], projection="X15c/10c", frame="+tVector heads and tails"
)

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

for vecstyle in [
    # 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
```

(continues on next page)

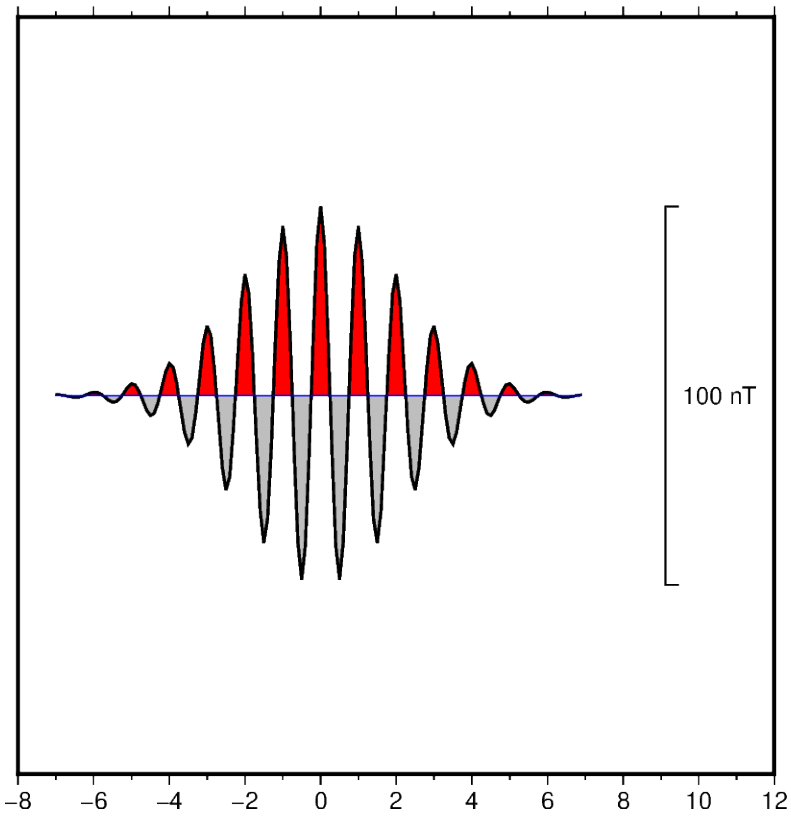
(continued from previous page)

```
    "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", fill="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 0.260 seconds)

5.2.14 Wiggle along tracks

The *pygmt.Figure.wiggle* method can plot z = f(x,y) anomalies along tracks. `x`, `y`, `z` can be specified as 1-D 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 `fillpositive` and/or `fillnegative` parameters.




```
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,
    z=z,
    # Set anomaly scale to 20 centimeters
    scale="20c",
    # Fill positive areas red
    fillpositive="red",
    # Fill negative areas gray
    fillnegative="gray",
    # Set the outline width to 1.0 point
    pen="1.0p",
    # Draw a blue track with a width of 0.5 points
    track="0.5p,blue",
    # Plot a vertical scale bar at Middle Right (MR). The bar length (+w)
    # is 100 in data (z) units. Set the z unit label (+l) to "nT".
    position="jMR+w100+lnT",
```

(continues on next page)

(continued from previous page)

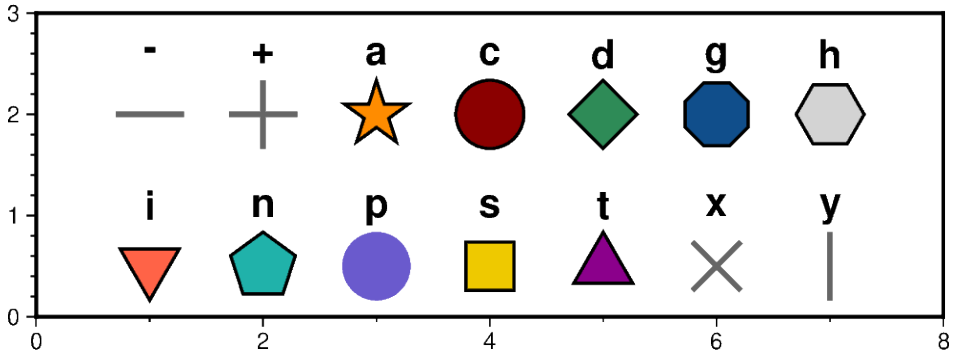
```
)
fig.show()
```

Total running time of the script: (0 minutes 0.125 seconds)

5.3 Symbols and markers

5.3.1 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 `fill` parameter. For the remaining symbols we may define a linewidth as well as a color fill.




```
import pygmt

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

# 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", fill="darkorange")
```

(continues on next page)

(continued from previous page)

```
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", fill="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", fill="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", fill="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", fill="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
# color fill to "tomato"
fig.plot(x=1, y=y, style="i0.9c", pen="1p,black", fill="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", fill="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", fill="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", fill="gold2")
fig.text(x=4, y=y + 0.6, text="s", font=font)
```

(continues on next page)

(continued from previous page)

```
# 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", fill="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 0.185 seconds)

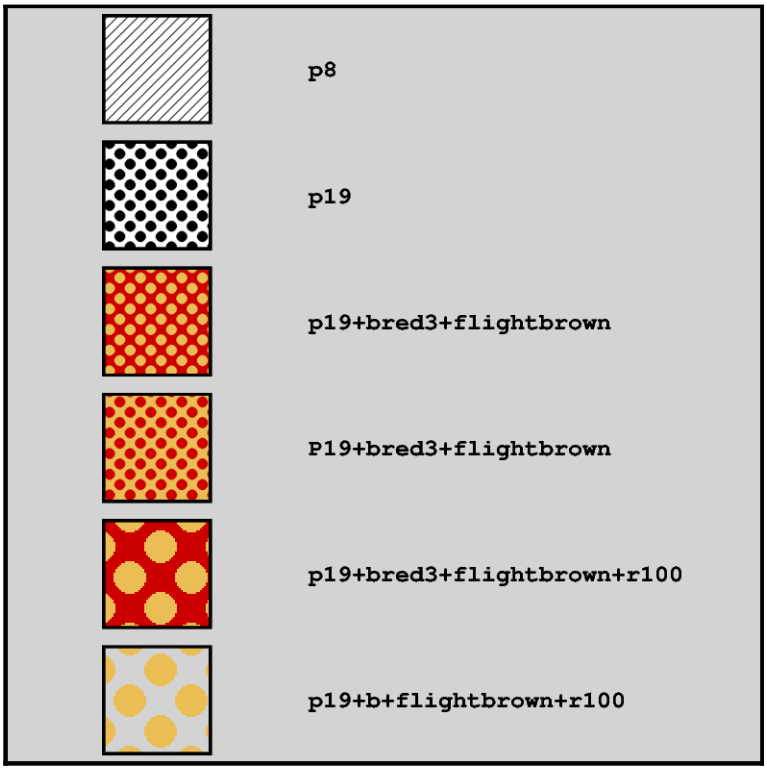
5.3.2 Bit and hachure patterns

In addition to colors, PyGMT also allows using bit and hachure patterns to fill symbols, polygons, and other areas, via the `fill` parameter or similar parameters.

Example method parameters that support bit and hachure patterns include:

- *pygmt.Figure.coast*: Land and water masses via `land` and `water`
- *pygmt.Figure.histogram*: Histogram bars via `fill`
- *pygmt.Figure.meca*: Focal mechanisms via `compressionfill` and `extensionfill`
- *pygmt.Figure.plot*: Symbols and polygons via `fill`
- *pygmt.Figure.rose*: Histogram sectors via `fill`
- *pygmt.Figure.solar*: Day-light terminators via `fill`
- *pygmt.Figure.ternary*: Symbols via `fill`
- *pygmt.Figure.velo*: Uncertainty wedges and velocity error ellipses via `uncertaintyfill`
- *pygmt.Figure.wiggle*: Anomalies via `fillpositive` and `fillnegative`

GMT provides 90 predefined patterns that can be used in PyGMT. The patterns are numbered from 1 to 90, and can be colored and inverted. The resolution of the pattern can be changed, and the background and foreground colors can be set. For a complete list of available patterns and the full syntax to specify a pattern, refer to the *Bit and Hachure Patterns*.




```
import pygmt

# A list of patterns that will be demonstrated.
# To use a pattern as fill append "p" and the number of the desired pattern.
# By default, the pattern is plotted in black and white with a resolution of 300 dpi.
patterns = [
    # Plot a hachted pattern via pattern number 8
    "p8",
    # Plot a dotted pattern via pattern number 19
    "p19",
    # Set the background color ("+b") to "red3" and the foreground color ("+f") to
    # "lightgray"
    "p19+bred3+flightbrown",
    # Invert the pattern by using a capitalized "P"
    "P19+bred3+flightbrown",
    # Change the resolution ("+r") to 100 dpi
    "p19+bred3+flightbrown+r100",
    # Make the background transparent by not giving a color after "+b";
    # works analogous for the foreground
    "p19+b+flightbrown+r100",
]

fig = pygmt.Figure()
fig.basemap(
    region=[0, 10, 0, 12],
    projection="X10c",
```

(continues on next page)

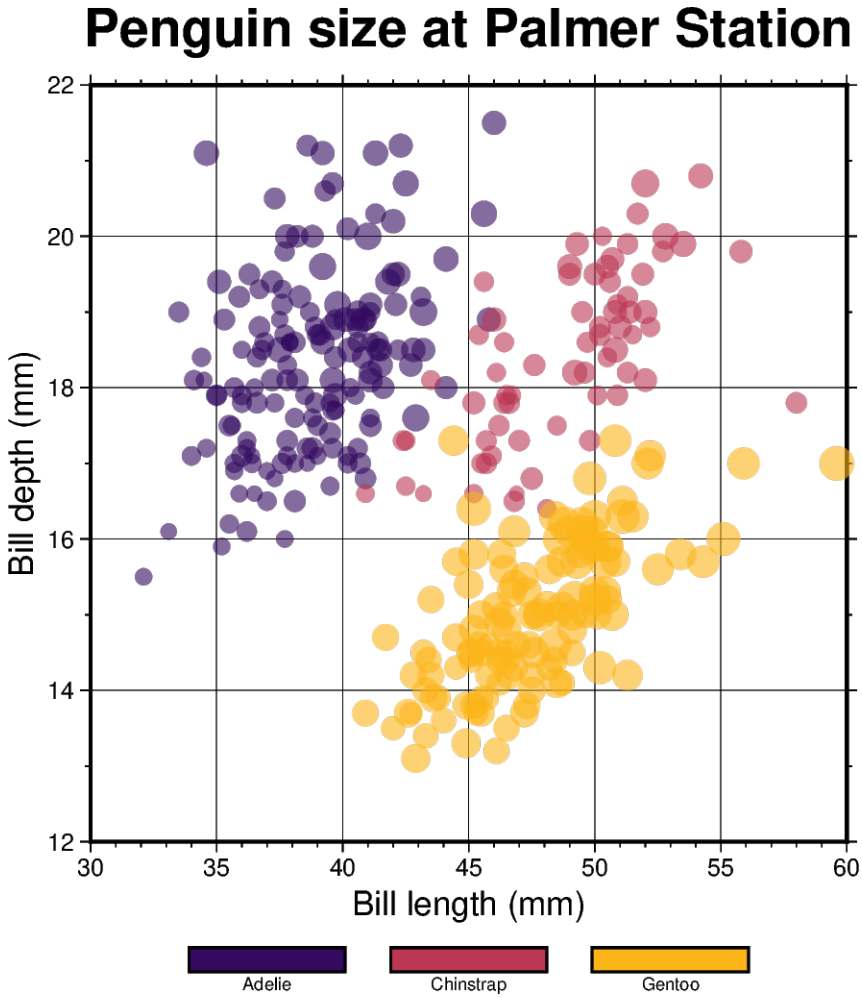
(continued from previous page)

```
    frame="rlbt+glightgray+tBit and Hachure Patterns",
)

y = 11
for pattern in patterns:
    # Plot a square with the pattern as fill.
    # The square has a size of 2 centimeters with a 1 point thick, black outline.
    fig.plot(x=2, y=y, style="s2c", pen="1p,black", fill=pattern)
    # Add a description of the pattern.
    fig.text(x=4, y=y, text=pattern, font="Courier-Bold", justify="ML")
    y -= 2
fig.show()
```

Total running time of the script: (0 minutes 0.183 seconds)

5.3.3 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 `fill` parameter via `fill=df.species.cat.codes.astype(int)`. Additionally, we have to set `cmap=True`. A desired colormap can be selected via the *pygmt.makecpt* function.



```
import pandas as pd
import pygmt

# Load sample penguins data
df = pd.read_csv("https://github.com/mwaskom/seaborn-data/raw/master/penguins.csv")

# Convert 'species' column to categorical dtype
# By default, pandas sorts the individual categories in an alphabetical order.
# For a non-alphabetical order, you have to manually adjust the list of
# categories. For handling and manipulating categorical data in pandas,
# have a look at:
# https://pandas.pydata.org/docs/user_guide/categorical.html
df.species = df.species.astype(dtype="category")

# Make a list of the individual categories of the 'species' column
# ['Adelie', 'Chinstrap', 'Gentoo']
# They are (corresponding to the categorical number code) by default in
# alphabetical order and later used for the colorbar annotations
cb_annots = list(df.species.cat.categories)

# 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 2-D 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+lBill length (mm)",
        "yafg+lBill depth (mm)",
        "WSen+tPenguin size at Palmer Station",
    ],
)

# Define a colormap 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",
    # Use the minimum and maximum of the categorical number code
```

(continues on next page)

(continued from previous page)

```
    # to set the lowest_value and the highest_value of the CPT
    series=(df.species.cat.codes.min(), df.species.cat.codes.max(), 1),
    # Convert ['Adelie', 'Chinstrap', 'Gentoo'] to 'Adelie,Chinstrap,Gentoo'
    color_model="+c" + ",".join(cb_annots),
)

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 symbol size according to the body mass, scaled by 7.5e-5
    size=df.body_mass_g * 7.5e-5,
    # Points colored by categorical number code (refers to the species)
    fill=df.species.cat.codes.astype(int),
    # Use colormap created by makecpt
    cmap=True,
    # Do not clip symbols that fall close to the plot bounds
    no_clip=True,
    # Use circles as symbols (the first "c") with diameter in
    # centimeters (the second "c")
    style="cc",
    # Set transparency level for all symbols to deal with overplotting
    transparency=40,
)

# Add colorbar legend
fig.colorbar()

fig.show()
```

Total running time of the script: (0 minutes 0.374 seconds)

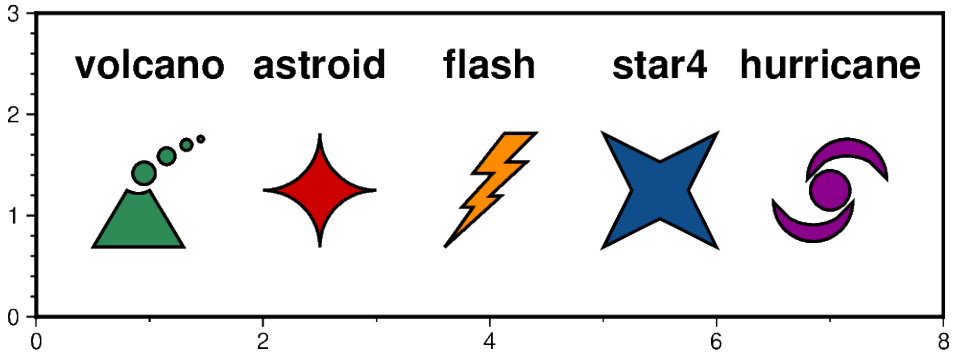
5.3.4 Custom symbols

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 Technical Reference https://docs.generic-mapping-tools.org/6.5/reference/custom-symbols.html.

Beside these built-in custom symbols GMT allows users to define their own custom symbols. For this, a specific macro language is used. A detailed introduction can be found at https://docs.generic-mapping-tools.org/6.5/reference/custom-symbols.html#the-macro-language. After defining such a symbol it can be used in the same way as a built-in custom symbol.

Please note: Custom symbols can not be used in auto-legends yet.




```
import pygmt

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

# Define pen and fontstyle 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, fill="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, fill="red3")
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, fill="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, fill="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, fill="magenta4")
fig.text(x=7, y=2.5, text="hurricane", font=font)

fig.show()
```

Total running time of the script: (0 minutes 0.144 seconds)

5.3.5 Datetime inputs

Datetime inputs of the following types are supported in PyGMT:

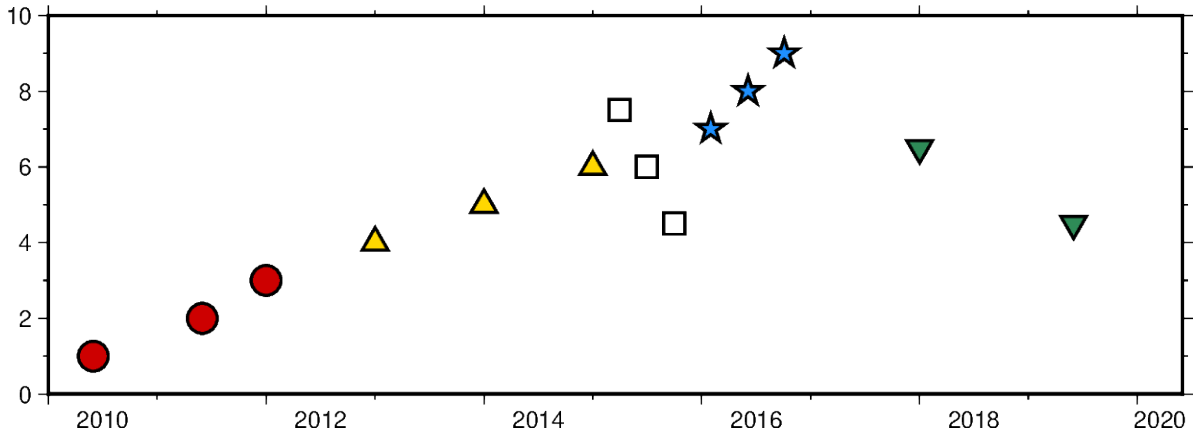
- `numpy.datetime64`
- `pandas.DatetimeIndex`
- `xarray.DataArray`: datetimes included in an *xarray.DataArray*
- raw datetime strings in ISO 8601 format (e.g. `"YYYY-MM-DD"`, `"YYYY-MM-DDTHH"`, and

"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.




```
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=np.datetime64
)
y = [1, 2, 3]
fig.plot(x=x, y=y, style="c0.4c", pen="1p", fill="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", fill="gold")

# xarray.DataArray
x = xr.DataArray(data=pd.date_range(start="2015-03", periods=3, freq="QS"))
```

(continues on next page)

(continued from previous page)

```
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", fill="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", fill="seagreen")

fig.show()
```

Total running time of the script: (0 minutes 0.126 seconds)

5.3.6 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 1-D 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, `diameter/startdir/stopdir`, the last two arguments are directions given in degrees counter-clockwise from horizontal. Append **+i** and the desired value to apply an inner diameter.

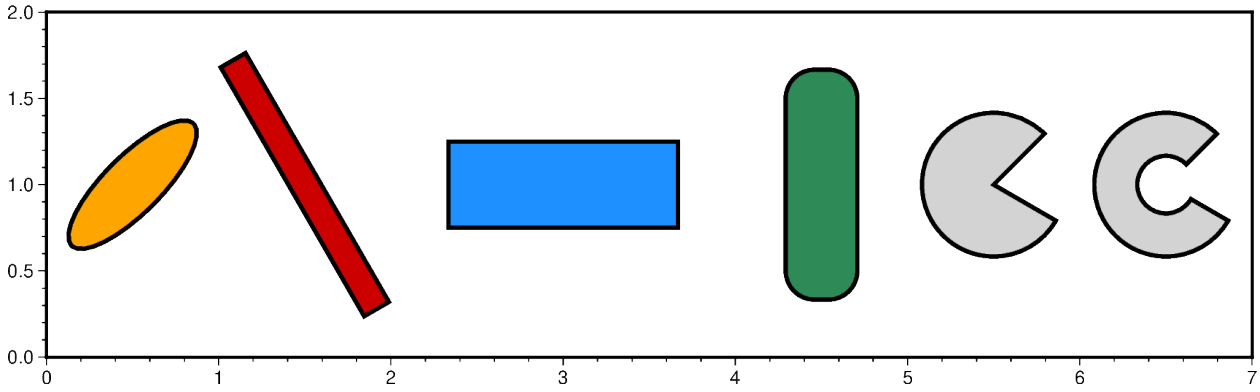
Uppercase versions **E**, **J**, and **W** are similar to **e**, **j**, and **w** but expect geographic azimuths and distances.

```
fig = pygmt.Figure()
fig.basemap(region=[0, 7, 0, 2], projection="x3c", frame=True)

# Ellipse
fig.plot(x=0.5, y=1, style="e45/3/1", fill="orange", pen="2p,black")
```

(continues on next page)

(continued from previous page)

```
# Rotated rectangle
fig.plot(x=1.5, y=1, style="j120/5/0.5", fill="red3", pen="2p,black")
# Rectangle
fig.plot(x=3, y=1, style="r4/1.5", fill="dodgerblue", pen="2p,black")
# Rounded rectangle
fig.plot(x=4.5, y=1, style="R1.25/4/0.5", fill="seagreen", pen="2p,black")
# Pie wedge
fig.plot(x=5.5, y=1, style="w2.5/45/330", fill="lightgray", pen="2p,black")
# Ring sector
fig.plot(x=6.5, y=1, style="w2.5/45/330+i1", fill="lightgray", pen="2p,black")

fig.show()
```



We can also plot symbols with varying parameters via defining those values in a 2-D 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 2-D list or numpy array are defined as:

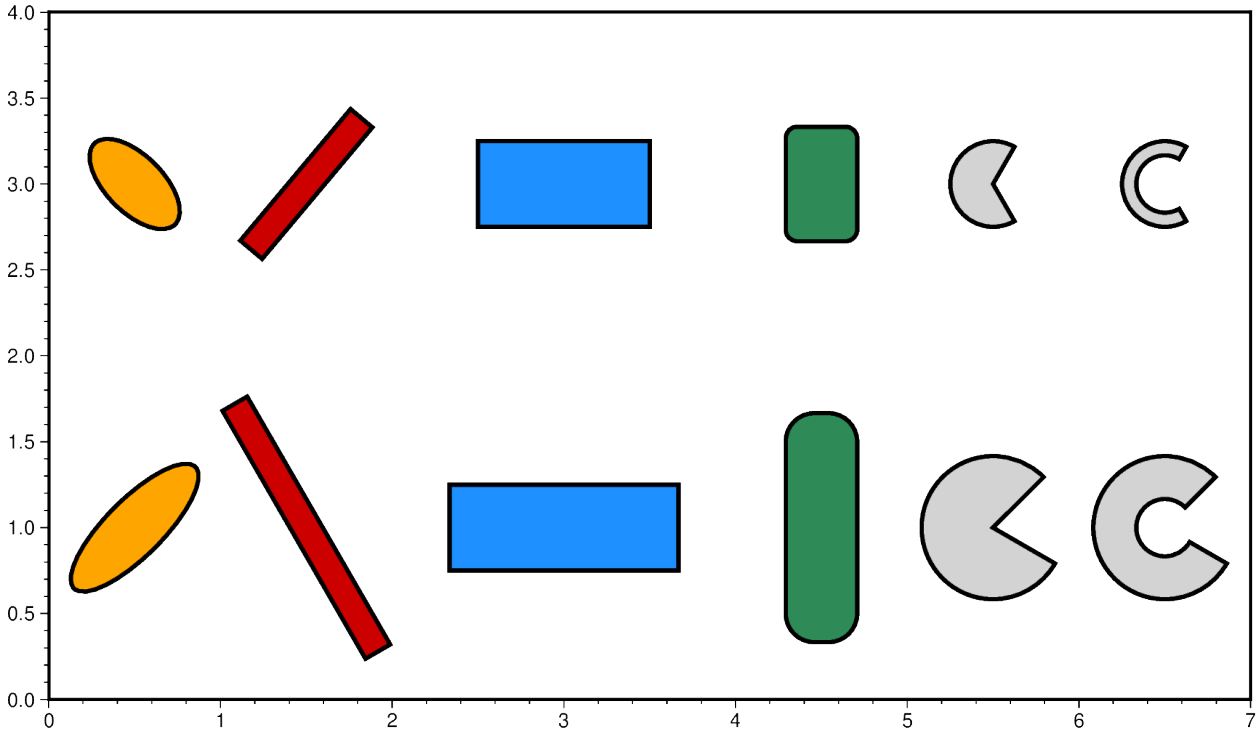
- **e**: ellipse, `[[lon, lat, direction, major_axis, minor_axis]]`
- **j**: rotated rectangle, `[[lon, lat, direction, width, height]]`
- **r**: rectangle, `[[lon, lat, width, height]]`
- **R**: rounded rectangle, `[[lon, lat, width, height, radius]]`
- **w**: pie wedge, `[[lon, lat, diameter, startdir,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

```
fig = pygmt.Figure()
fig.basemap(region=[0, 7, 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", fill="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", fill="red3", pen="2p,black")
# Rectangle
data = [[3, 1, 4, 1.5], [3, 3, 3, 1.5]]
fig.plot(data=data, style="r", fill="dodgerblue", pen="2p,black")
# Rounded rectangle
```

(continues on next page)

(continued from previous page)

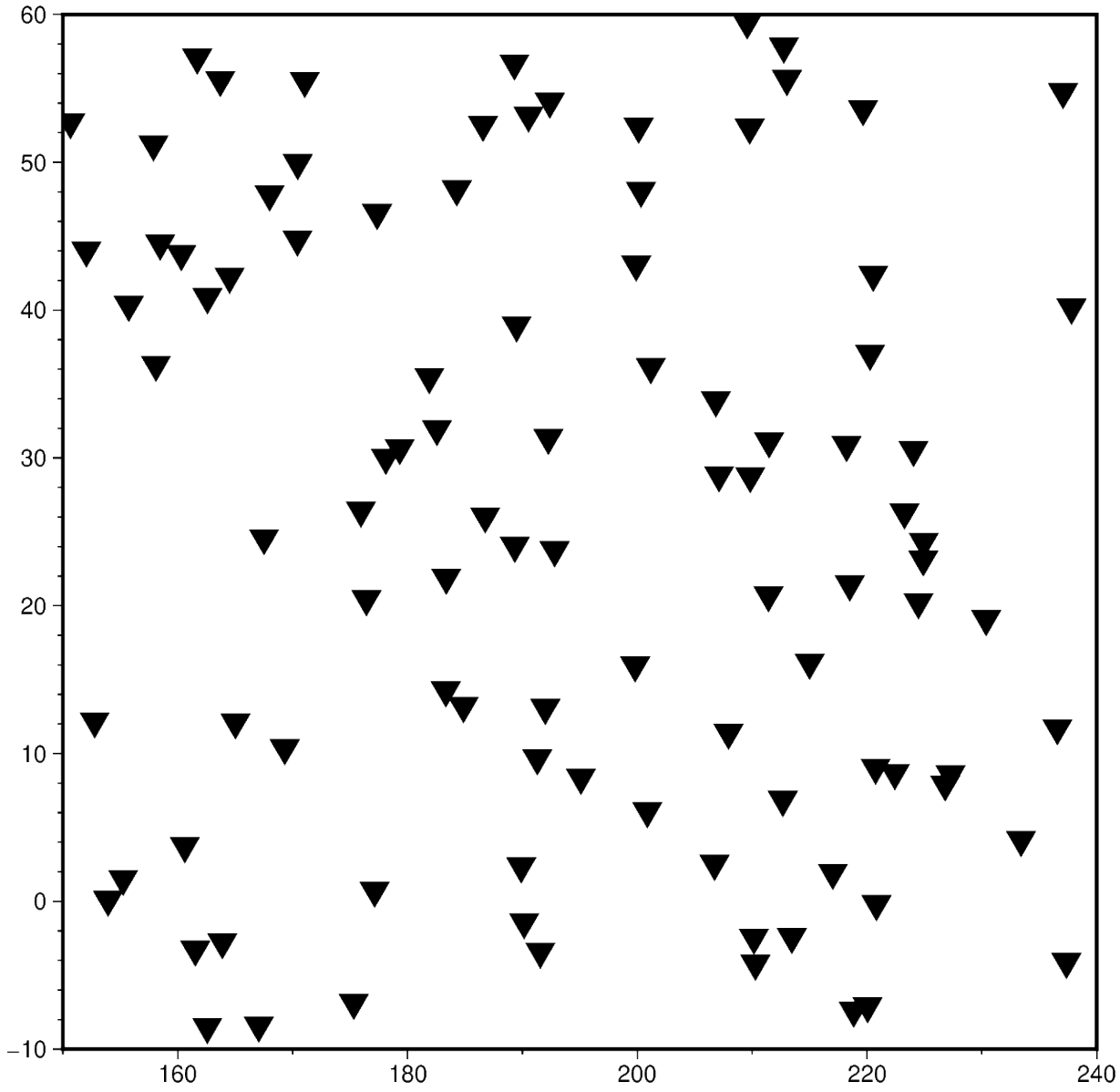
```
data = [[4.5, 1, 1.25, 4, 0.5], [4.5, 3, 1.25, 2.0, 0.2]]
fig.plot(data=data, style="R", fill="seagreen", pen="2p,black")
# Pie wedge
data = [[5.5, 1, 2.5, 45, 330], [5.5, 3, 1.5, 60, 300]]
fig.plot(data=data, style="w", fill="lightgray", pen="2p,black")
# Ring sector
data = [[6.5, 1, 2.5, 45, 330], [6.5, 3, 1.5, 60, 300]]
fig.plot(data=data, style="w+i1", fill="lightgray", pen="2p,black")

fig.show()
```



Total running time of the script: (0 minutes 0.427 seconds)

5.3.7 Points

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




```
import numpy as np
import pygmt

# Generate a random set of points to plot
rng = np.random.default_rng(seed=42)
region = [150, 240, -10, 60]
x = rng.uniform(low=region[0], high=region[1], size=100)
y = rng.uniform(low=region[2], high=region[3], size=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
```

(continues on next page)

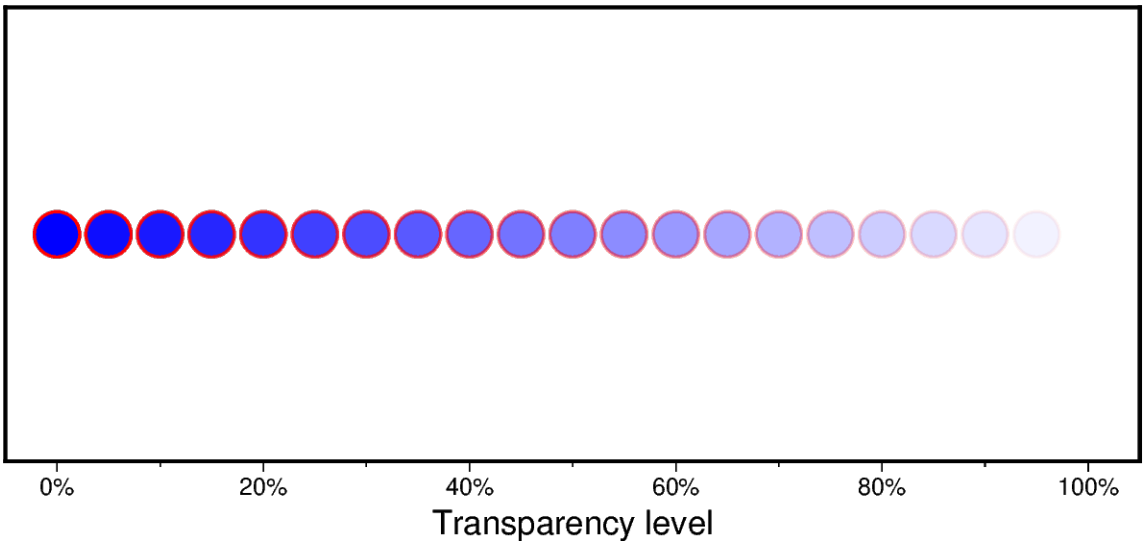
(continued from previous page)

```
fig.plot(x=x, y=y, style="i0.5c", fill="black")
fig.show()
```

Total running time of the script: (0 minutes 0.165 seconds)

5.3.8 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*.



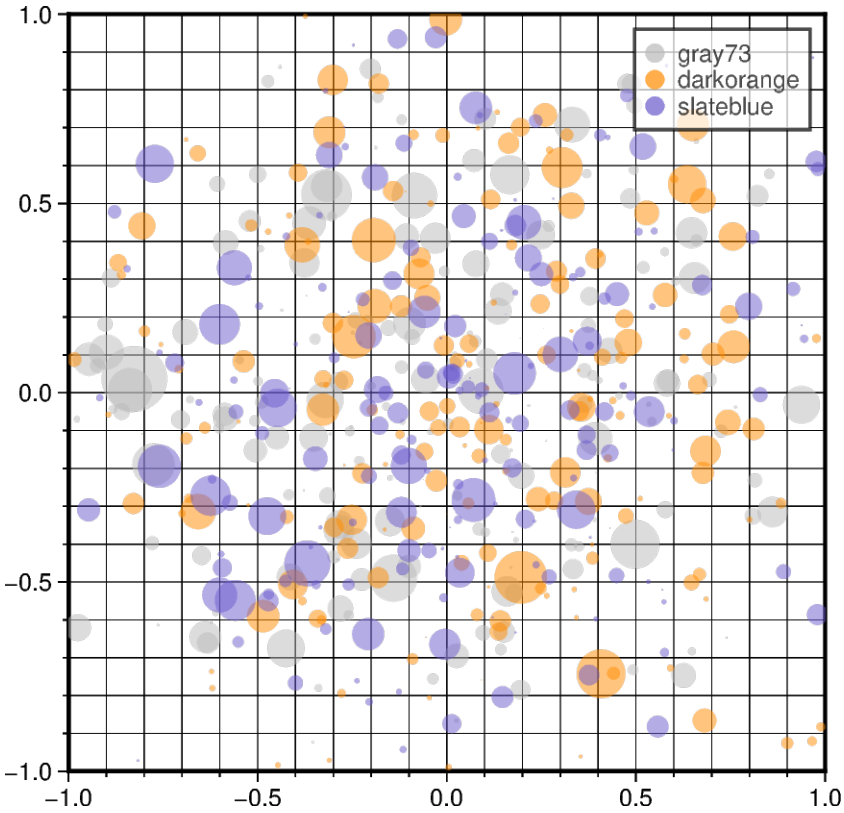

```
import numpy as np
import pygmt

# 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

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

Total running time of the script: (0 minutes 0.179 seconds)

5.3.9 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/stable/gallery/lines_bars_and_markers/scatter_with_legend.html




```
import numpy as np
import pygmt

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

fig = pygmt.Figure()
fig.basemap(
    region=[-1, 1, -1, 1],
    projection="X10c/10c",
    frame=["xa0.5fg", "ya0.5fg", "WSrt"],
)
for fill in ["gray73", "darkorange", "slateblue"]:
    # Generate standard normal distributions centered on 0
    # with standard deviations of 1
    x = rng.normal(loc=0, scale=0.5, size=n)  # random x data
    y = rng.normal(loc=0, scale=0.5, size=n)  # random y data
    size = rng.normal(loc=0, scale=0.5, size=n) * 0.5  # random size, in cm

    # plot data points as circles (style="c"), with different sizes
```

(continues on next page)

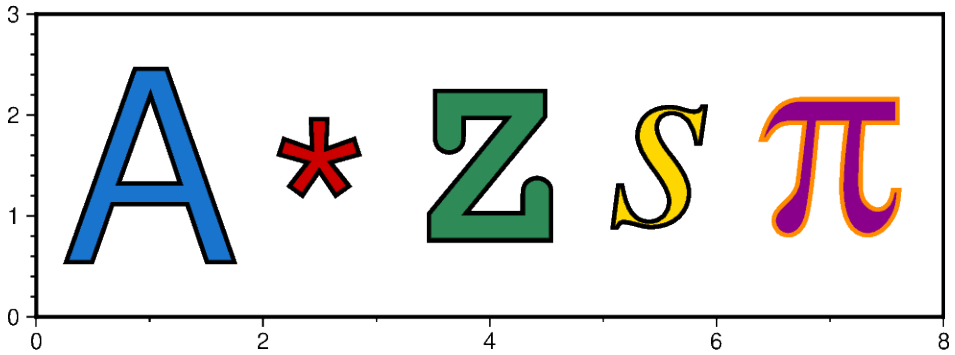
(continued from previous page)

```
    fig.plot(
        x=x,
        y=y,
        style="c",
        size=size,
        fill=fill,
        # Set the legend label,
        # and set the symbol size to be 0.25 cm (+S0.25c) in legend
        label=f"{fill}+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 0.241 seconds)

5.3.10 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,outlinecolor* to select a particular font [Default is FONT_ANNOT_PRIMARY] and outline color [Default is black] as well as **+j***justify* to change the justification [Default is CM]. For all supported fonts see *Supported Fonts*. The fill color of the text symbols can be set with the `fill` parameter, and the outline width can be customized with the `pen` parameter.




```
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", fill="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*", fill="red3", pen=pen)
# plot an uppercase "Z" of size 3.5c and use the "Courier-Bold" font,
# color fill is set to "seagreen"
```

(continues on next page)

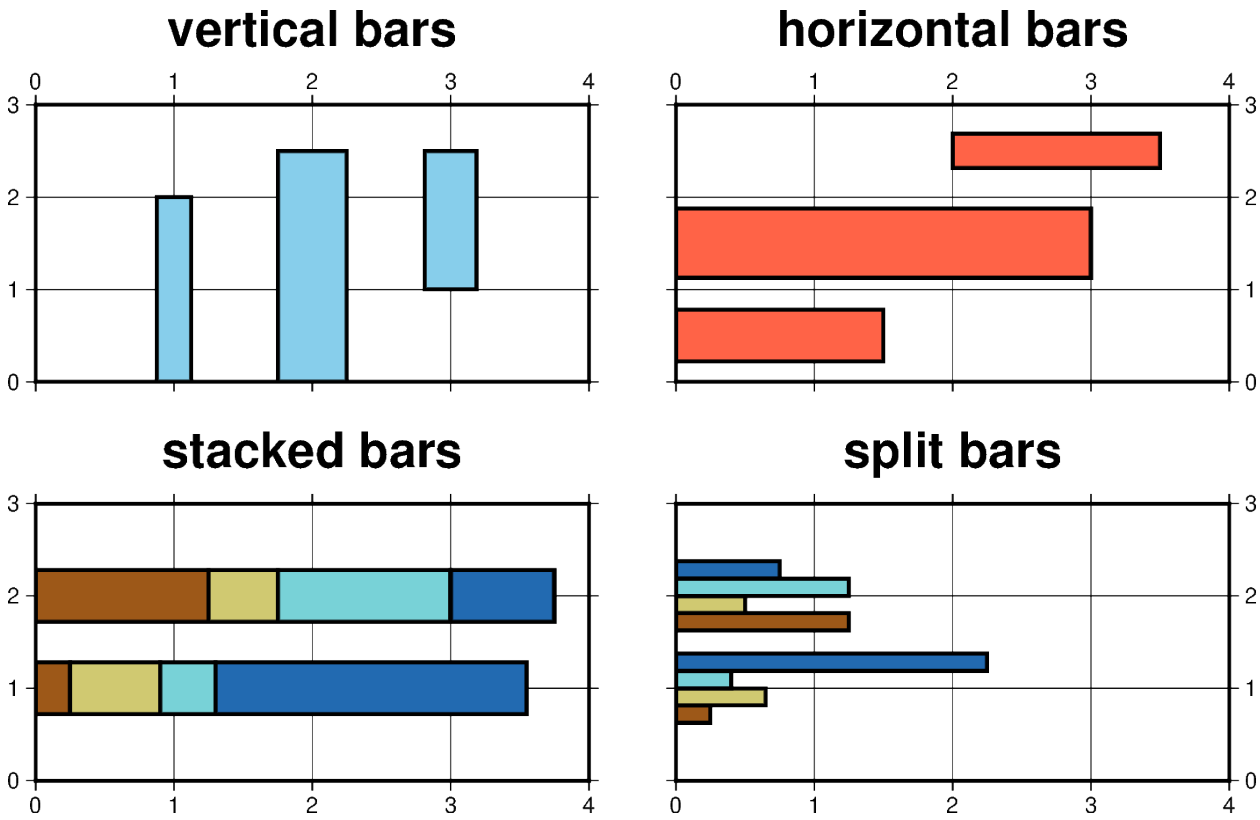
(continued from previous page)

```
fig.plot(x=4, y=1.5, style="l3.5c+tZ+fCourier-Bold", fill="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", fill="gold", pen=pen)
# plot the pi symbol of size 3.5c, the outline color of the symbol is set to
# "darkorange", the color fill is set to "magenta4"
fig.plot(x=7, y=1.5, style="l3.5c+tπ+fdarkorange", fill="magenta4", pen=pen)

fig.show()
```

Total running time of the script: (0 minutes 0.135 seconds)

5.3.11 Vertical or horizontal bars

The *pygmt.Figure.plot* method can plot vertical (**b**) or horizontal (**B**) bars by passing the corresponding shortcut to the `style` parameter. By default, *base* = 0 meaning that the bar is starting from 0. Append **+b**[*base*] to change this value. To plot multi-band bars, please append **+v**|**i***ny* (for verticals bars) or **+v**|**i***nx* (for horizontal ones), where *ny* or *nx* indicate the total number of bands in the bar (and hence the number of values required to follow the *x,y* coordinate pair in the input). Here, **+i** means we must accumulate the bar values from the increments *dy* or *dx*, while **+v** means we get the complete values relative to base. Normally, the bands are plotted as sections of a final single bar. Use **+s** to instead split the bar into *ny* or *nx* side-by-side, individual and thinner bars. Multi-band bars require `cmap=True` with one color per band.




```
import pygmt

fig = pygmt.Figure()
```

(continues on next page)

(continued from previous page)

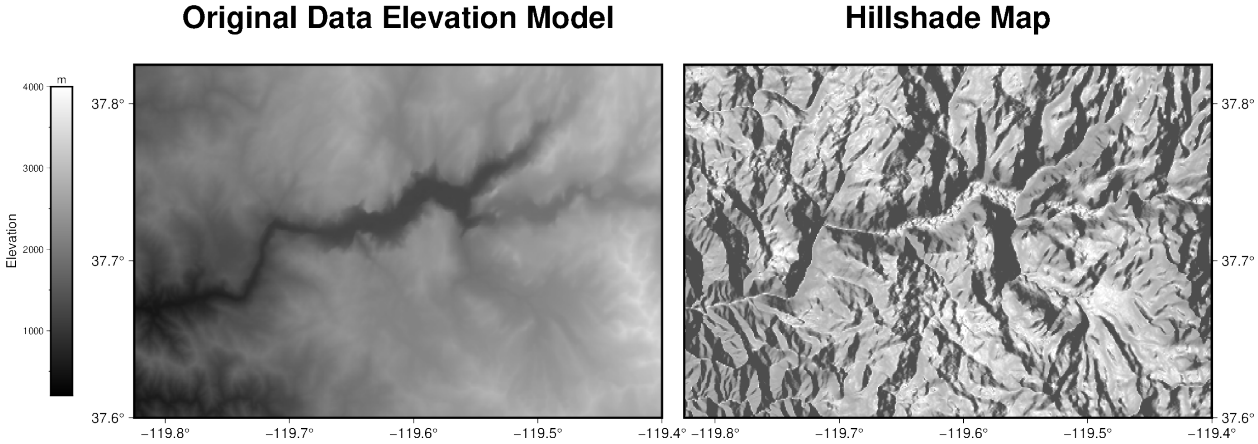
```
pygmt.makecpt(cmap="roma", series=[0, 4, 1])

with fig.subplot(
    nrows=2,
    ncols=2,
    subsize=("8c", "4c"),
    frame="ag",
    sharey=True,
    sharex=True,
    margins=["0.5c", "0.75c"],
):
    pen = "1.5p"
    with fig.set_panel(panel=0):
        fill = "skyblue"
        fig.basemap(region=[0, 4, 0, 3], frame="+tvertical bars")
        fig.plot(x=1, y=2, style="b0.5c", fill=fill, pen=pen)
        fig.plot(x=2, y=2.5, style="b1c", fill=fill, pen=pen)
        # +b1 means that the bar is starting from y=1 here
        fig.plot(x=3, y=2.5, style="b0.75c+b1", fill=fill, pen=pen)

    with fig.set_panel(panel=1):
        fill = "tomato"
        fig.basemap(region=[0, 4, 0, 3], frame="+thorizontal bars")
        fig.plot(x=1.5, y=0.5, style="B0.75c", fill=fill, pen=pen)
        fig.plot(x=3, y=1.5, style="B1c", fill=fill, pen=pen)
        # +b2 means that the bar is starting from x=2 here
        fig.plot(x=3.5, y=2.5, style="B0.5c+b2", fill=fill, pen=pen)

    # generate dictionary for plotting multi-band bars
    data = {
        "x1": [0.25, 1.25],
        "y": [1, 2],
        "x2": [0.65, 0.5],
        "x3": [0.4, 1.25],
        "x4": [2.25, 0.75],
    }

    with fig.set_panel(panel=2):
        fig.basemap(region=[0, 4, 0, 3], frame="+tstacked bars")
        fig.plot(data=data, style="B0.75c+i4", cmap=True, pen=pen)

    with fig.set_panel(panel=3):
        fig.basemap(region=[0, 4, 0, 3], frame="+tsplit bars")
        fig.plot(data=data, style="B1c+v4+s", cmap=True, pen=pen)

fig.show()
```

Total running time of the script: (0 minutes 0.220 seconds)

5.4 Images, contours, and fields

5.4.1 Calculating grid gradient and radiance

The *pygmt.grdgradient* function 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 an 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.




```
import pygmt

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

# Load sample grid (3 arc-seconds 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 -----------

pygmt.makecpt(cmap="gray", series=[200, 4000, 10])
fig.grdimage(
    grid=grid,
    projection="M12c",
    frame=["WSrt+tOriginal 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 -----------
```

(continues on next page)

(continued from previous page)

```
# 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+tHillshade Map", "xa0.1", "ya0.1"],
    cmap=True,
)

fig.show()
```

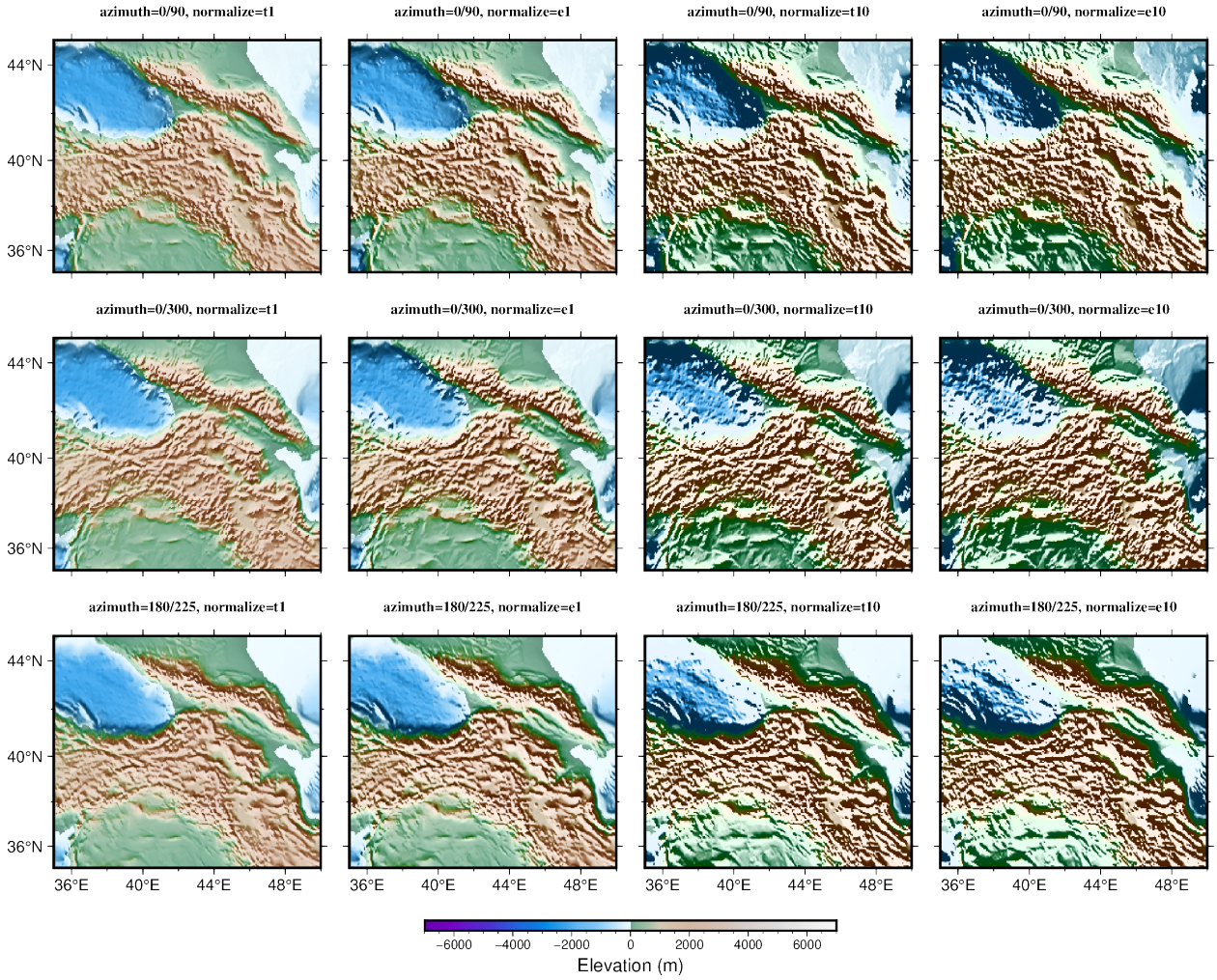
Total running time of the script: (0 minutes 0.393 seconds)

5.4.2 Calculating grid gradient with custom `azimuth` and `normalize` parameters

The *pygmt.grdgradient* function calculates the gradient of a grid file. As input, *pygmt.grdgradient* gets an `xarray.DataArray` object or a path string to a grid file. It then calculates the respective gradient and returns an `xarray.DataArray` object. The example below sets two main parameters:

- `azimuth` to set the illumination light source direction (0° is North, 90° is East, 180° is South, 270° is West).
- `normalize` to enhance the three-dimensional sense of the topography.

The `normalize` parameter calculates the azimuthal gradient of each point along a certain azimuth angle, then adjusts the brightness value of the color according to the positive/negative of the azimuthal gradient and the amplitude of each point.




```
grdblend [NOTICE]: Remote data courtesy of GMT data server oceania [http://oceania.
↪generic-mapping-tools.org]
grdblend [NOTICE]: SRTM15 Earth Relief v2.6 at 03x03 arc minutes reduced by Gaussian␣
↪Cartesian filtering (15.7 km fullwidth) [Tozer et al., 2019].
grdblend [NOTICE]:   -> Download 90x90 degree grid tile (earth_relief_03m_g): N00E000
```

```
import pygmt

# Load the 3 arc-minutes global relief grid in the target area around Caucasus
grid = pygmt.datasets.load_earth_relief(resolution="03m", region=[35, 50, 35, 45])

fig = pygmt.Figure()

# Define a colormap to be used for topography
pygmt.makecpt(cmap="terra", series=[-7000, 7000])
```

(continues on next page)

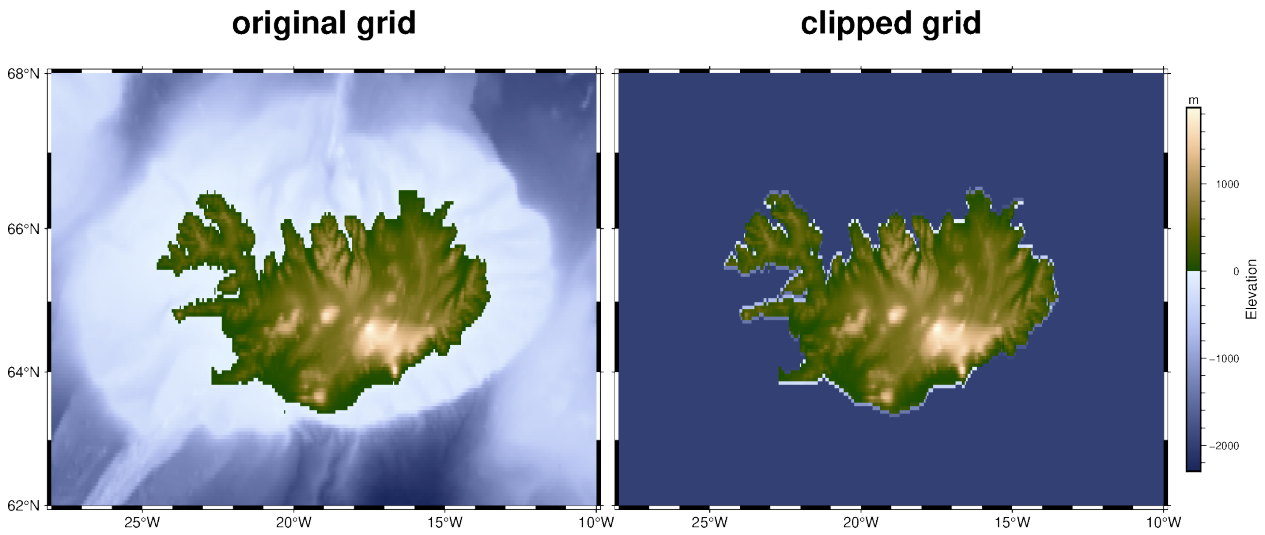
(continued from previous page)

```
# Define figure configuration
pygmt.config(FONT_TITLE="10p,5", MAP_TITLE_OFFSET="1p", MAP_FRAME_TYPE="plain")

# Setup subplot panels with three rows and four columns
with fig.subplot(
    nrows=3,
    ncols=4,
    figsize=("28c", "21c"),
    sharex="b",
    sharey="l",
):
    # E.g. "0/90" illuminates light source from the North (top) and East
    # (right), and so on
    for azi in ["0/90", "0/300", "180/225"]:
        # "e" and "t" are cumulative Laplace distribution and cumulative
        # Cauchy distribution, respectively
        # "amp" (e.g. 1 or 10) controls the brightness value of the color
        for nor in ["t1", "e1", "t10", "e10"]:
            # Making an intensity DataArray using azimuth and normalize
            # parameters
            shade = pygmt.grdgradient(grid=grid, azimuth=azi, normalize=nor)
            fig.grdimage(
                grid=grid,
                shading=shade,
                projection="M?",
                frame=["a4f2", f"+tazimuth={azi}, normalize={nor}"],
                cmap=True,
                panel=True,
            )

fig.colorbar(position="JBC+w10c/0.25c+h", frame="xa2000f500+lElevation (m)")

fig.show()
```

Total running time of the script: (0 minutes 4.112 seconds)

5.4.3 Clipping grid values

The *pygmt.grdclip* function 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.




```
grdblend [NOTICE]: Remote data courtesy of GMT data server oceania [http://oceania.
↪generic-mapping-tools.org]
grdblend [NOTICE]: SRTM15 Earth Relief v2.6 at 03x03 arc minutes reduced by Gaussian␣
↪Cartesian filtering (15.7 km fullwidth) [Tozer et al., 2019].
grdblend [NOTICE]:   -> Download 90x90 degree grid tile (earth_relief_03m_g): N00W090
```

```
import pygmt

fig = pygmt.Figure()

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

# Load sample grid (3 arc-minutes 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=["WSne+toriginal grid", "xa5f1", "ya2f1"],
)
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])
```

(continues on next page)

(continued from previous page)

```
# Plot clipped grid
fig.basemap(
    region=region,
    projection="M12c",
    frame=["wSne+tclipped grid", "xa5f1", "ya2f1"],
)
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 3.291 seconds)

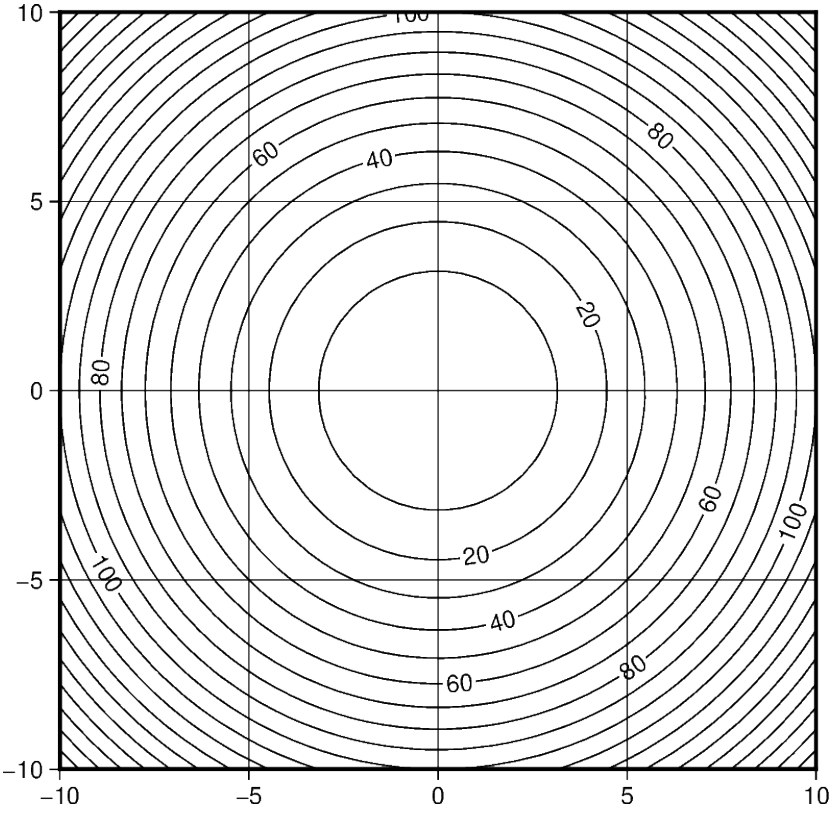
5.4.4 Contours

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` 1-D `numpy.ndarray` data columns.
2. `data` 2-D `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 1-D `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.

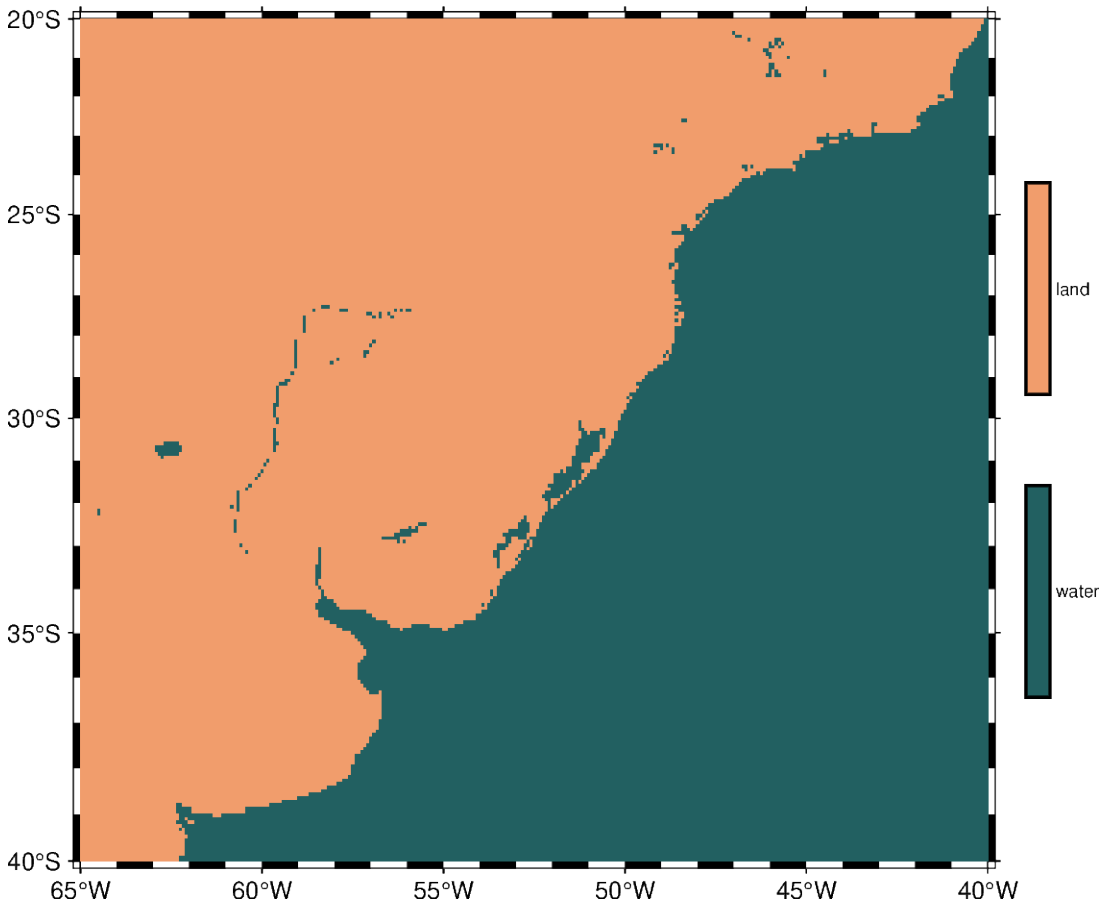


```
import numpy as np
import pygmt

# 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 1-D data columns
    x=x,
    y=y,
    z=z,
    # 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 0.220 seconds)

5.4.5 Create 'wet-dry' mask grid

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




```
grdimage [WARNING]: Your CPT is categorical. Enabling -nn+a to avoid interpolation␣
↪across categories.
```

```
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-minutes 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)
```

(continues on next page)

(continued from previous page)

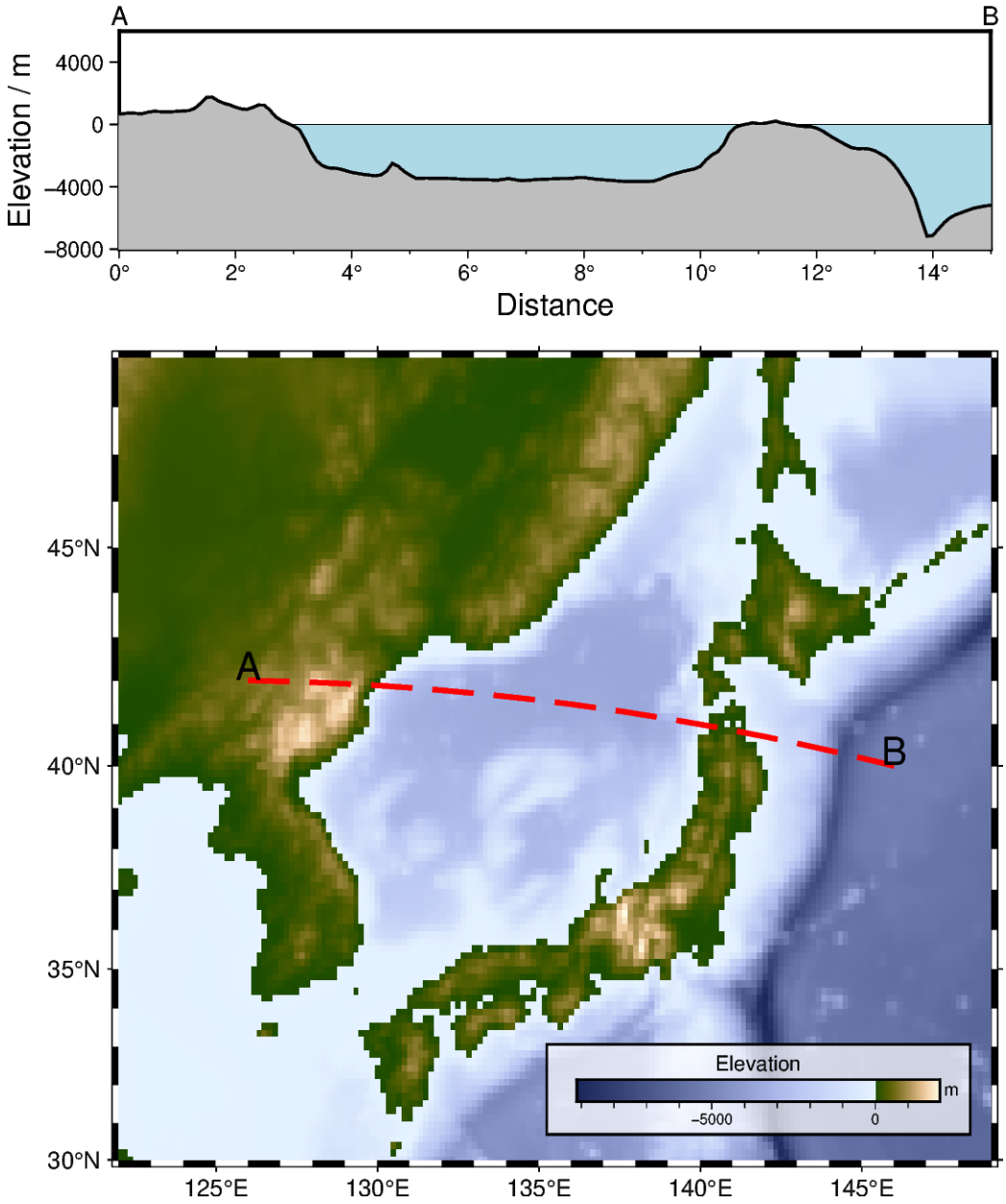
```
# Define a colormap to be used for two categories, define the range of the
# new discrete CPT using series=(lowest_value, highest_value, interval),
# use color_model="+cwater,land" to write the discrete color palette
# "batlow" in categorical format and add water/land as annotations for the
# colorbar.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

5.4.6 Cross-section along a transect

pygmt.project and *pygmt.grdtrack* can be used to focus on a quantity and its variation along a desired survey line. In this example, the elevation is extracted from a grid provided via *pygmt.datasets.load_earth_relief*. The figure consists of two parts, a map of the elevation in the study area showing the survey line and a Cartesian plot showing the elevation along the survey line.

This example is orientated on an example in the GMT/China documentation: https://docs.gmt-china.org/latest/examples/ex026/




```
import pygmt

# Define region of study area
# lon_min, lon_max, lat_min, lat_max in degrees East and North
region_map = [122, 149, 30, 49]

# Create a new pygmt.Figure instance
fig = pygmt.Figure()

# --------------------------------------------------------------------------
# Bottom: Map of elevation in study area

# Set up basic map using a Mercator projection with a width of 12 centimeters
```

(continues on next page)

(continued from previous page)

```
fig.basemap(region=region_map, projection="M12c", frame="af")

# Download grid for Earth relief with a resolution of 10 arc-minutes and gridline
# registration [Default]
grid_map = pygmt.datasets.load_earth_relief(resolution="10m", region=region_map)

# Plot the downloaded grid with color-coding based on the elevation
fig.grdimage(grid=grid_map, cmap="oleron")

# Add a colorbar for the elevation
fig.colorbar(
    # Place the colorbar inside the plot (lowercase "j") in the Bottom Right (BR)
    # corner with an offset ("+o") of 0.7 centimeters and 0.3 centimeters in x or y
    # directions, respectively; move the x label above the horizontal colorbar ("+ml")
    position="jBR+o0.7c/0.8c+h+w5c/0.3c+ml",
    # Add a box around the colobar with a fill ("+g") in "white" color and a
    # transparency ("@") of 30 % and with a 0.8-points thick, black, outline ("+p")
    box="+gwhite@30+p0.8p,black",
    # Add x and y labels ("+l")
    frame=["x+lElevation", "y+lm"],
)

# Choose a survey line
fig.plot(
    x=[126, 146],  # Longitude in degrees East
    y=[42, 40],  # Latitude in degrees North
    # Draw a 2-points thick, red, dashed line for the survey line
    pen="2p,red,dashed",
)

# Add labels "A" and "B" for the start and end points of the survey line
fig.text(
    x=[126, 146],
    y=[42, 40],
    text=["A", "B"],
    offset="0c/0.2c",  # Move text 0.2 centimeters up (y direction)
    font="15p",  # Use a font size of 15 points
)

# ---------------------------------------------------------------------------
# Top: Elevation along survey line

# Shift plot origin to the top by the height of the map ("+h") plus 1.5 centimeters
fig.shift_origin(yshift="h+1.5c")

fig.basemap(
    region=[0, 15, -8000, 6000],  # x_min, x_max, y_min, y_max
    # Cartesian projection with a width of 12 centimeters and a height of 3␣
↪centimeters
    projection="X12c/3c",
    # Add annotations ("a") and ticks ("f") as well as labels ("+l") at the west or
    # left and south or bottom sides ("WSrt")
```

(continues on next page)

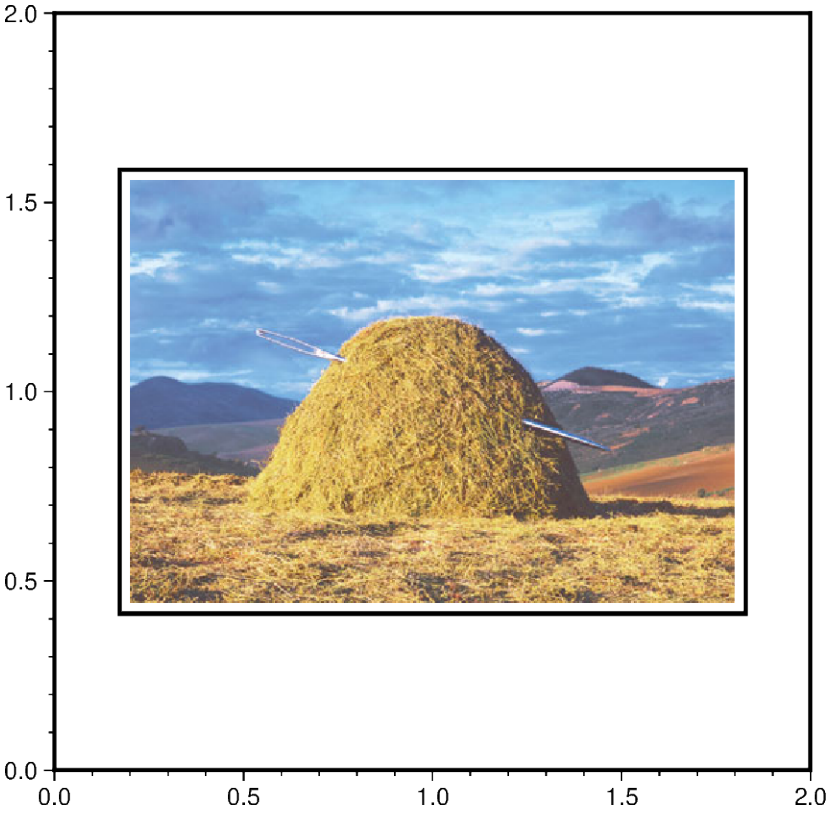
(continued from previous page)

```
    frame=["WSrt", "xa2f1+lDistance+u°", "ya4000+lElevation / m"],
)

# Add labels "A" and "B" for the start and end points of the survey line
fig.text(
    x=[0, 15],
    y=[7000, 7000],
    text=["A", "B"],
    no_clip=True,  # Do not clip text that fall outside the plot bounds
    font="10p",  # Use a font size of 10 points
)

# Generate points along a great circle corresponding to the survey line and store them
# in a pandas.DataFrame
track_df = pygmt.project(
    center=[126, 42],  # Start point of survey line (longitude, latitude)
    endpoint=[146, 40],  # End point of survey line (longitude, latitude)
    generate=0.1,  # Output data in steps of 0.1 degrees
)

# Extract the elevation at the generated points from the downloaded grid and add it as
# new column "elevation" to the pandas.DataFrame
track_df = pygmt.grdtrack(grid=grid_map, points=track_df, newcolname="elevation")

# Plot water masses
fig.plot(
    x=[0, 15],
    y=[0, 0],
    fill="lightblue",  # Fill the polygon in "lightblue"
    # Draw a 0.25-points thick, black, solid outline
    pen="0.25p,black,solid",
    close="+y-8000",  # Force closed polygon
)

# Plot elevation along the survey line
fig.plot(
    x=track_df.p,
    y=track_df.elevation,
    fill="gray",  # Fill the polygon in "gray"
    # Draw a 1-point thick, black, solid outline
    pen="1p,black,solid",
    close="+y-8000",  # Force closed polygon
)

fig.show()
```

Total running time of the script: (0 minutes 0.300 seconds)

5.4.7 Image on a figure

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.




```
from pathlib import Path

import pygmt

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

# Place and center ("+jCM") the image "needle.jpg" provided by GMT to the position
# ("+g") 1/1 on the current plot, scale it to a width of 8 centimeters ("+w") and draw
# a rectangular border around it
fig.image(
    imagefile="https://oceania.generic-mapping-tools.org/cache/needle.jpg",
    position="g1/1+w8c+jCM",
    box=True,
)

fig.show()

# Clean up the downloaded image in the current directory
Path("needle.jpg").unlink()
```

Total running time of the script: (0 minutes 1.164 seconds)

5.4.8 RGB image

The *pygmt.Figure.grdimage* method can be used to plot Red, Green, Blue (RGB) images, or any 3-band false color combination. Here, we'll use `rioxarray.open_rasterio` to read a GeoTIFF file into an `xarray.DataArray` format, and plot it on a map.

The example below shows a Worldview 2 satellite image over Lāhainā, Hawai'i during the August 2023 wildfires. Data is sourced from a Cloud-Optimized GeoTIFF (COG) file hosted on OpenAerialMap under a CC BY-NC 4.0 license.

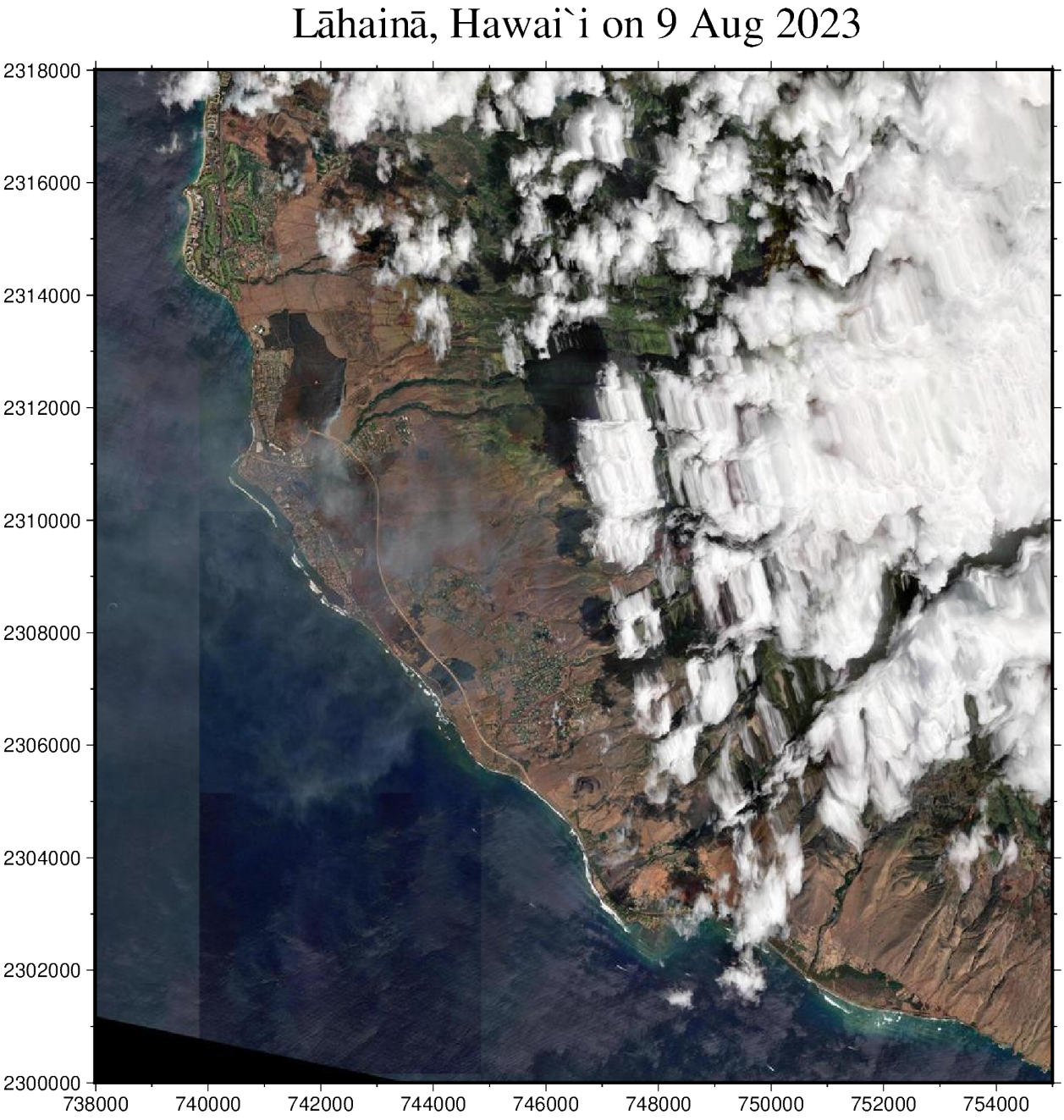
```
import pygmt
import rioxarray
```

Read 3-band data from GeoTIFF into an xarray.DataArray object:

```
with rioxarray.open_rasterio(
    filename="https://oin-hotosm-temp.s3.us-east-1.amazonaws.com/
↪64d6a49a19cb3a000147a65b/0/64d6a49a19cb3a000147a65c.tif",
    overview_level=5,
) as img:
    # Subset to area of Lāhainā in EPSG:32604 coordinates
    image = img.rio.clip_box(minx=738000, maxx=755000, miny=2300000, maxy=2318000)
    image = image.load()  # Force loading the DataArray into memory
image
```

Plot the RGB imagery:

```
fig = pygmt.Figure()
with pygmt.config(FONT_TITLE="Times-Roman"):  # Set title font to Times-Roman
    fig.grdimage(
        grid=image,
        # Use a map scale where 1 cm on the map equals 1 km on the ground
        projection="x1:100000",
        frame=[r"WSne+tL@!a^hain@!a^, Hawai`i on 9 Aug 2023", "af"],
    )
fig.show()
```

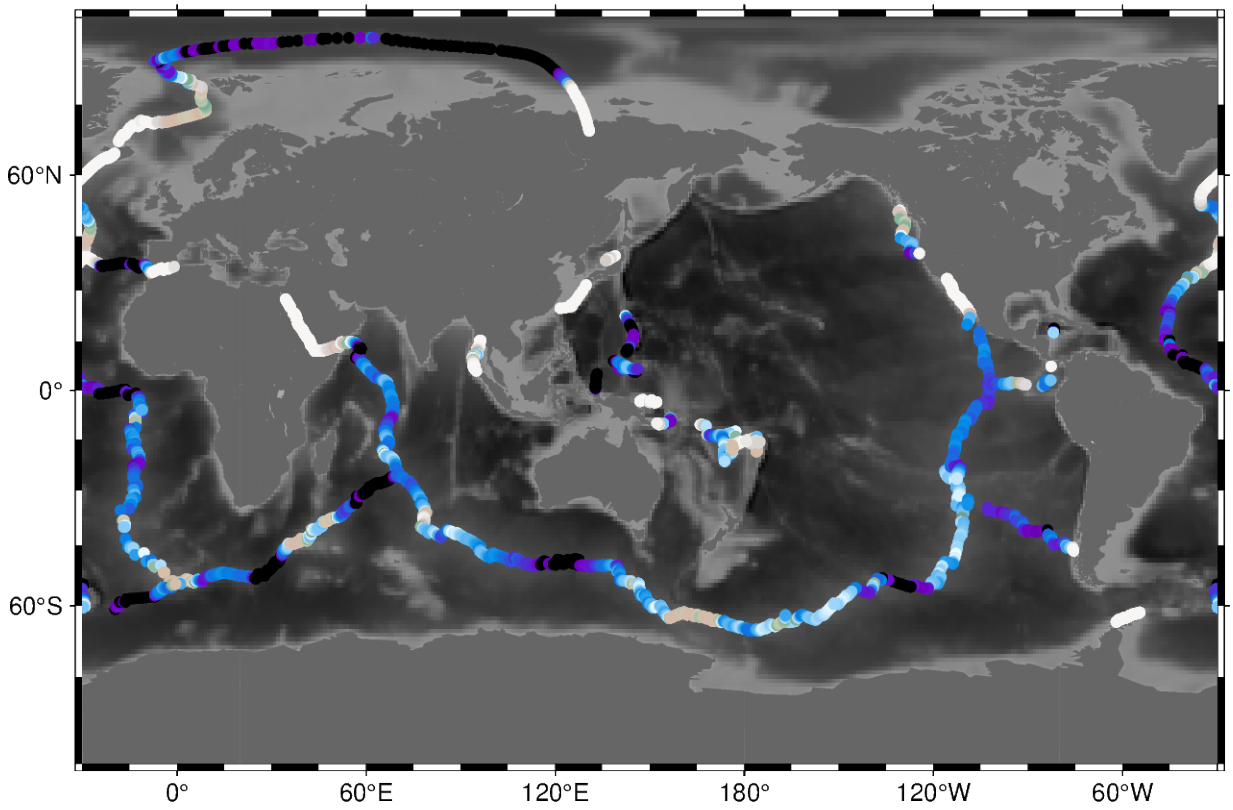


Total running time of the script: (0 minutes 1.063 seconds)

5.4.9 Sampling along tracks

The *pygmt.grdtrack* function samples a raster grid's value along specified points. We will need to input a 2-D 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.



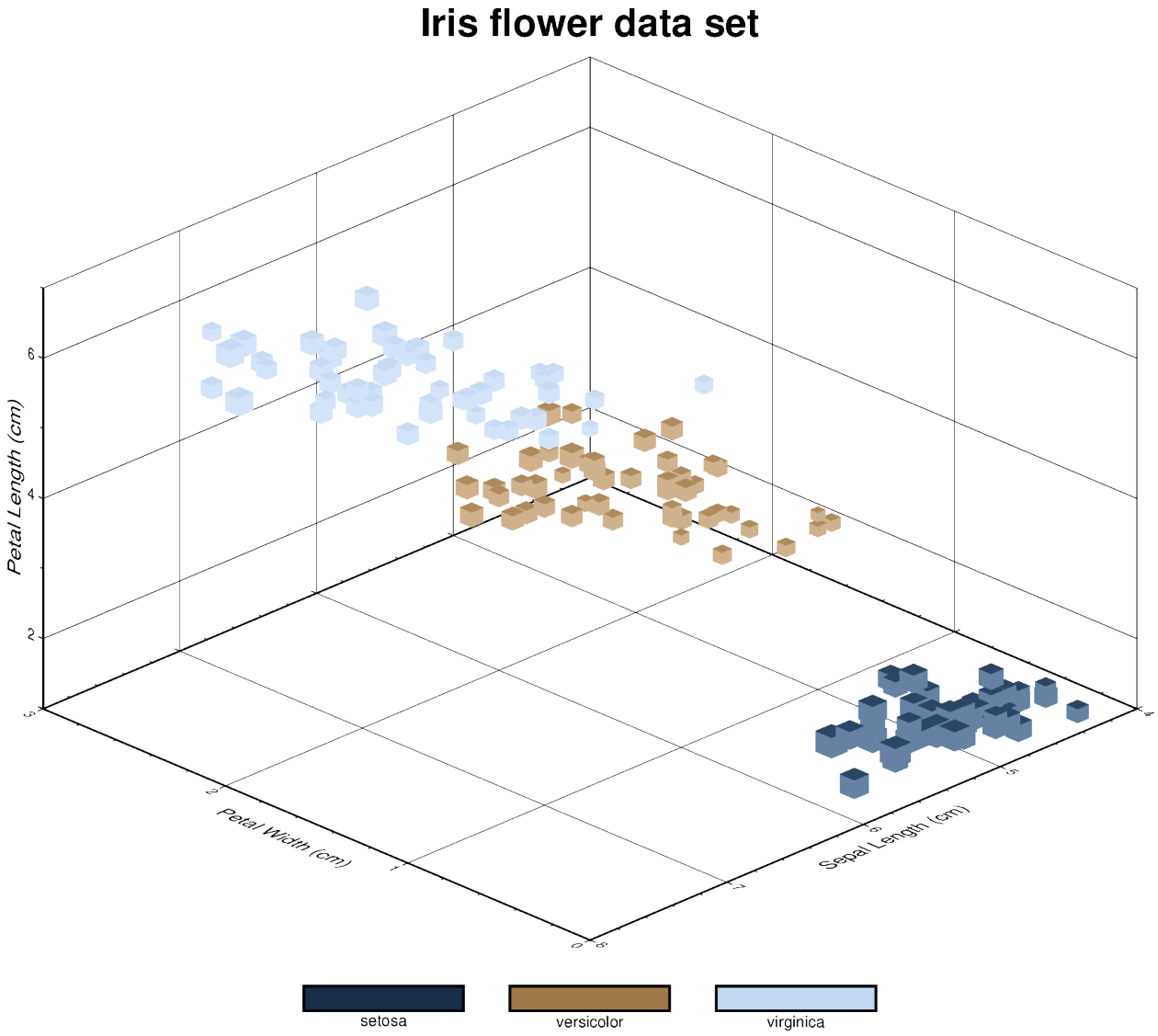

```
import pygmt

# Load sample grid and point datasets
grid = pygmt.datasets.load_earth_relief()
points = pygmt.datasets.load_sample_data(name="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",
    fill=(track.bathymetry - track.bathymetry.mean()) / track.bathymetry.std(),
)
fig.show()
```

Total running time of the script: (0 minutes 0.373 seconds)

5.5 3D Plots

5.5.1 3-D scatter plots

The *pygmt.Figure.plot3d* method can be used to plot symbols in 3-D. In the example below, we show how the Iris flower dataset can be visualized using a perspective 3-D 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.




```
import pandas as pd
import pygmt

# Load sample iris data
df = pd.read_csv("https://github.com/mwaskom/seaborn-data/raw/master/iris.csv")

# Convert 'species' column to categorical dtype
# By default, pandas sorts the individual categories in an alphabetical order.
```

(continues on next page)

(continued from previous page)

```
# For a non-alphabetical order, you have to manually adjust the list of
# categories. For handling and manipulating categorical data in pandas,
# have a look at:
# https://pandas.pydata.org/docs/user_guide/categorical.html
df.species = df.species.astype(dtype="category")

# Make a list of the individual categories of the 'species' column
# ['setosa', 'versicolor', 'virginica']
# They are (corresponding to the categorical number code) by default in
# alphabetical order and later used for the colorbar labels
labels = list(df.species.cat.categories)

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

# Define a colormap 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",
    # Use the minimum and maximum of the categorical number code
    # to set the lowest_value and the highest_value of the CPT
    series=(df.species.cat.codes.min(), df.species.cat.codes.max(), 1),
    # Convert ['setosa', 'versicolor', 'virginica'] to
    # 'setosa,versicolor,virginica'
    color_model="+c" + ",".join(labels),
)

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 the sepal width, scaled by 0.1
    size=0.1 * df.sepal_width,
    # Use 3-D cubes ("u") as symbols with size in centimeters ("c")
    style="uc",
    # Points colored by categorical number code (refers to the species)
    fill=df.species.cat.codes.astype(int),
```

(continues on next page)

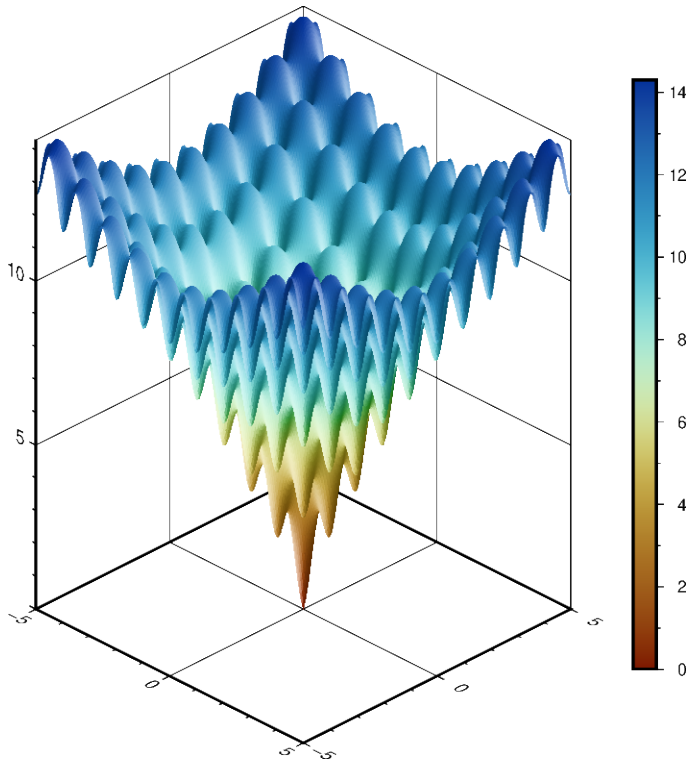
(continued from previous page)

```
    # Use colormap created by makecpt
    cmap=True,
    # Set map dimensions (xmin, xmax, ymin, ymax, zmin, zmax)
    region=region,
    # Set frame parameters
    frame=[
        "WsNeZ3+tIris flower data set",  # z axis label positioned on 3rd corner, add␣
↪title
        "xafg+lPetal Width (cm)",
        "yafg+lSepal Length (cm)",
        "zafg+lPetal Length (cm)",
    ],
    # Set perspective to azimuth NorthWest (315°), at elevation 25°
    perspective=[315, 25],
    # Vertical exaggeration factor
    zscale=1.5,
)

# Shift the plot origin in x direction temporarily and add the colorbar
with fig.shift_origin(xshift=3.1):
    fig.colorbar()

fig.show()
```

Total running time of the script: (0 minutes 0.358 seconds)

5.5.2 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"`.




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


# Define an interesting function of two variables, see:
# https://en.wikipedia.org/wiki/Ackley_function
def ackley(x, y):
    """
    Ackley function.
    """
    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 centimeters
```

(continues on next page)

(continued from previous page)

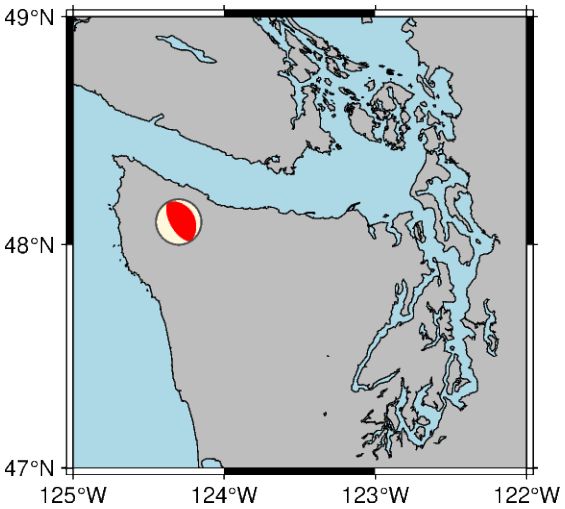
```
fig.grdview(
    data,
    # Set annotations and gridlines in steps of five, and
    # tick marks in steps of one
    frame=["a5f1g5", "za5f1g5"],
    projection=f"x{SCALE}c",
    zscale=f"{SCALE}c",
    surftype="s",
    cmap="roma",
    perspective=[135, 30],  # Azimuth southeast (135°), at elevation 30°
    shading="+a45",
)

# Add colorbar for gridded data
fig.colorbar(
    frame="a2f1",  # Set annotations in steps of two, tick marks in steps of one
    position="JMR",  # Place colorbar in the Middle Right (MR) corner
)

fig.show()
```

Total running time of the script: (0 minutes 2.269 seconds)

5.6 Seismology and geodesy

5.6.1 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 dictionary using the `spec` parameter (or they can be specified as a 1-D or 2-D array, or within a file). The size of the beachballs can be set using the `scale` parameter. The compressive and extensive quadrants can be filled either with a color or a pattern via the `compressionfill` and `extensionfill` parameters, respectively. Use the `pen` parameter to adjust the outline of the beachballs.




```
import pygmt
```

(continues on next page)

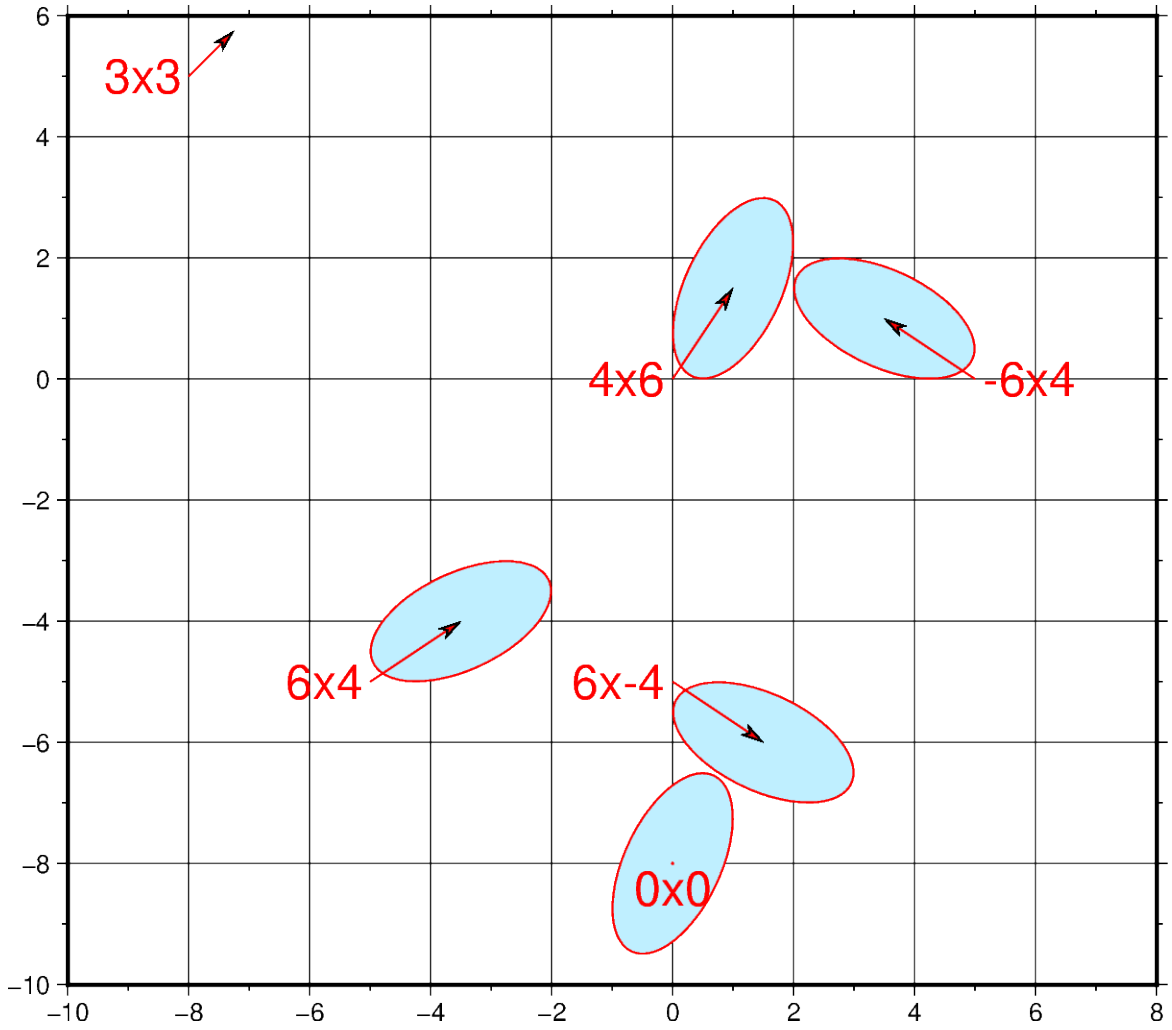
(continued from previous page)

```
fig = pygmt.Figure()

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

# Store focal mechanism parameters in a dictionary based on the Aki & Richards
# convention
focal_mechanism = {"strike": 330, "dip": 30, "rake": 90, "magnitude": 3}

# Pass the focal mechanism data through the spec parameter. In addition provide
# scale, event location, and event depth
fig.meca(
    spec=focal_mechanism,
    scale="1c",  # in centimeters
    longitude=-124.3,
    latitude=48.1,
    depth=12.0,
    # Fill compressive quadrants with color "red"
    # [Default is "black"]
    compressionfill="red",
    # Fill extensive quadrants with color "cornsilk"
    # [Default is "white"]
    extensionfill="cornsilk",
    # Draw a 0.5 points thick dark gray ("gray30") solid outline via
    # the pen parameter [Default is "0.25p,black,solid"]
    pen="0.5p,gray30,solid",
)

fig.show()
```

Total running time of the script: (0 minutes 0.153 seconds)

5.6.2 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 lightblue 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.




```
import pandas as pd
import pygmt

fig = pygmt.Figure()
df = pd.DataFrame(
    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],
    projection="x0.8c",
```

(continues on next page)

(continued from previous page)

```
    frame=["WSne", "2g2f"],
    spec="e0.2/0.39+f18",
    uncertaintyfill="lightblue1",
    pen="0.6p,red",
    line=True,
    vector="0.3c+p1p+e+gred",
)

fig.show()
```

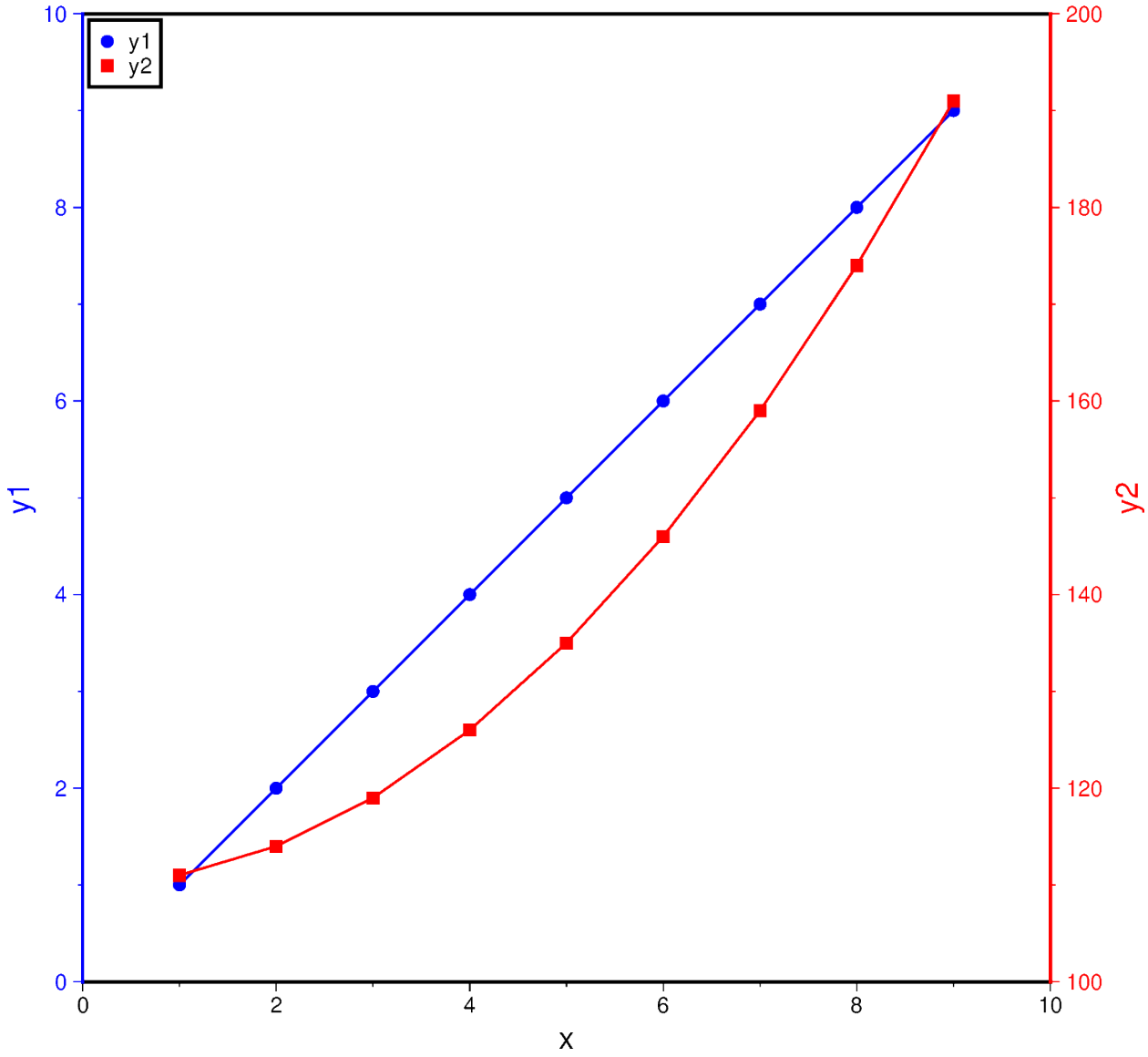
Total running time of the script: (0 minutes 0.208 seconds)

5.7 Base maps

5.7.1 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**). Lowercase 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.




```
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

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"])
```

(continues on next page)

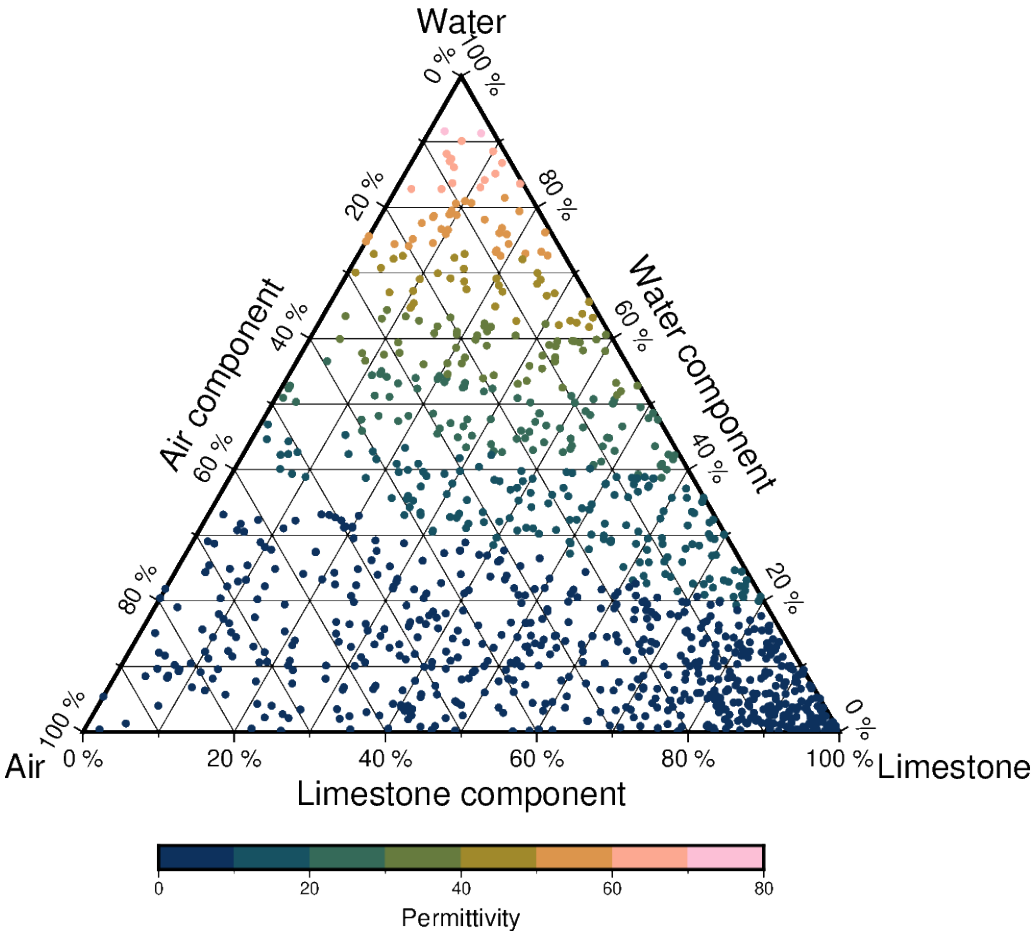
(continued from previous page)

```
# 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", fill="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", fill="red", label="y2")

# Create a legend in the Top Left (TL) corner of the plot with an
# offset of 0.1 centimeters
fig.legend(position="jTL+o0.1c", box=True)

fig.show()
```

Total running time of the script: (0 minutes 0.201 seconds)

5.7.2 Ternary diagram

The *pygmt.Figure.ternary* method can draw ternary diagrams. The example shows how to plot circles with a diameter of 0.1 centimeters (`style="c0.1c"`) on a 10-centimeters-wide (`width="10c"`) ternary diagram at the positions listed in the first three columns of the sample dataset `rock_compositions`, with default annotations and gridline spacings, using the specified labeling defined via `alabel`, `blabel`, and `clabel`. Points are colored based on the values given in the fourth columns of the sample dataset via `cmap=True`.




```
import pygmt

fig = pygmt.Figure()

# Load sample data
data = pygmt.datasets.load_sample_data(name="rock_compositions")

# Define a colormap to be used for the values given in the fourth column
# of the input dataset
pygmt.makecpt(cmap="batlow", series=[0, 80, 10])

fig.ternary(
    data,
    region=[0, 100, 0, 100, 0, 100],
    width="10c",
    style="c0.1c",
    alabel="Limestone",
    blabel="Water",
    clabel="Air",
    cmap=True,
    frame=[
        "aafg+lLimestone component+u %",
        "bafg+lWater component+u %",
```

(continues on next page)

(continued from previous page)

```
        "cafg+lAir component+u %",
    ],
)

# Add a colorbar indicating the values given in the fourth column of
# the input dataset
fig.colorbar(position="JBC+o0c/1.5c", frame="x+lPermittivity")
fig.show()
```

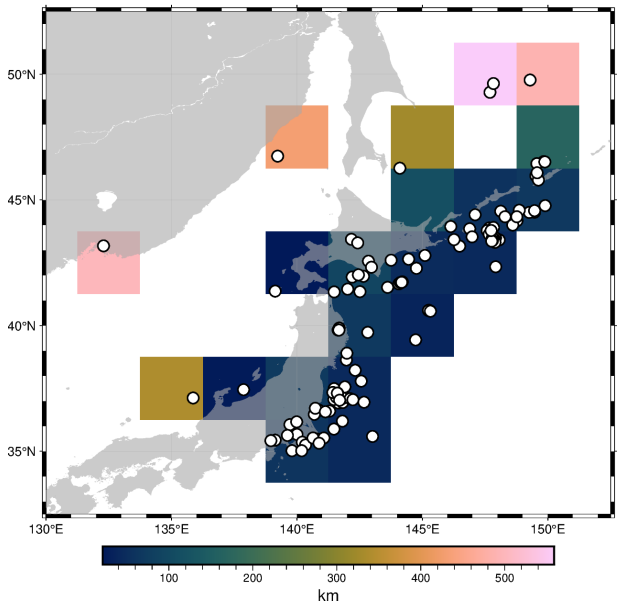
Total running time of the script: (0 minutes 0.199 seconds)

5.8 Histograms

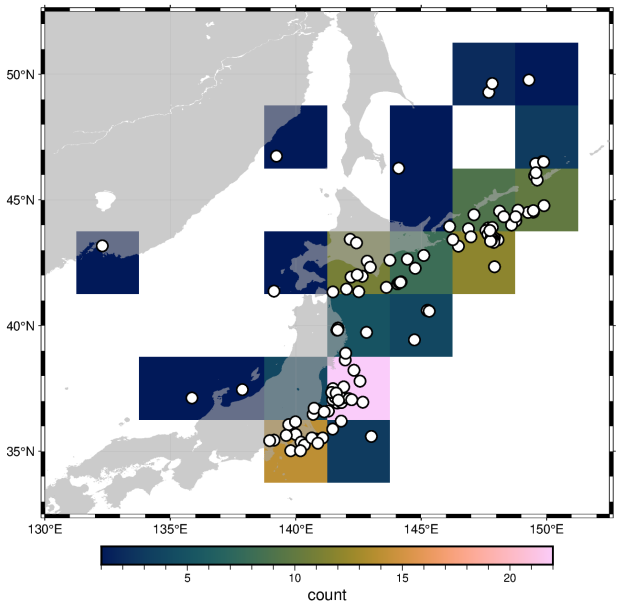
5.8.1 Blockmean

The *pygmt.blockmean* function calculates different quantities inside blocks/bins whose dimensions are defined via the `spacing` parameter. The following examples show how to calculate the averages of the given values inside each bin and how to report the number of points inside each bin.

Mean earthquake depth inside each block



Number of points inside each block




```
import pygmt

# Load sample data
data = pygmt.datasets.load_sample_data(name="japan_quakes")
# Select only needed columns
data = data[["longitude", "latitude", "depth_km"]]

# Set the region for the plot
region = [130, 152.5, 32.5, 52.5]
# Define spacing in x and y direction (150x150 arc-minute blocks)
spacing = "150m"
```

(continues on next page)

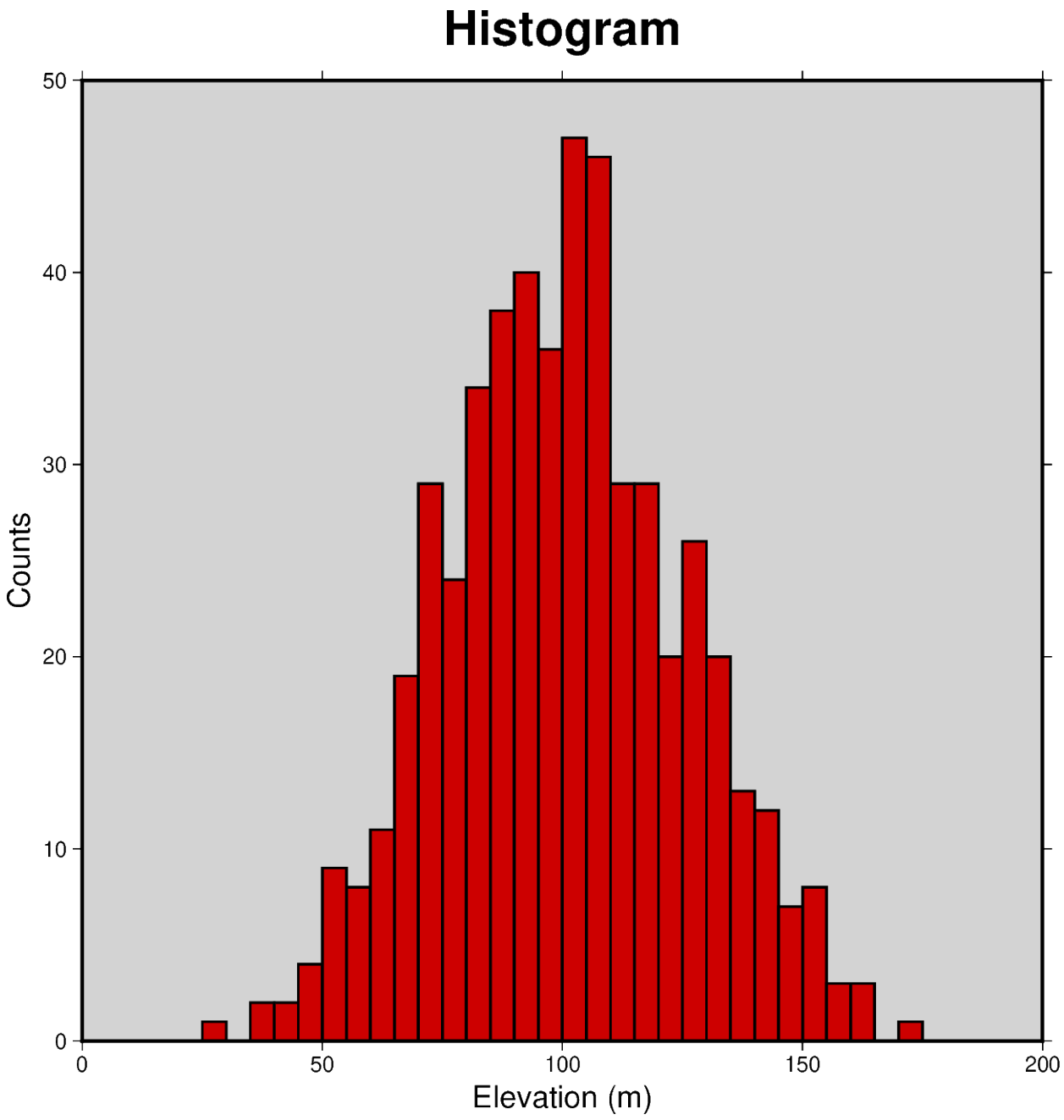
(continued from previous page)

```
fig = pygmt.Figure()

# Calculate mean depth in kilometers from all events within
# 150x150 arc-minute bins using blockmean
df = pygmt.blockmean(data=data, region=region, spacing=spacing)
# Convert to grid
grd = pygmt.xyz2grd(data=df, region=region, spacing=spacing)

fig.grdimage(
    grid=grd,
    region=region,
    frame=["af", "+tMean earthquake depth inside each block"],
    cmap="batlow",
)
# Plot slightly transparent landmasses on top
fig.coast(land="darkgray", transparency=40)
# Plot original data points
fig.plot(x=data.longitude, y=data.latitude, style="c0.3c", fill="white", pen="1p,black
↪")
fig.colorbar(frame="x+lkm")

fig.shift_origin(xshift="w+5c")

# Calculate number of total locations within 150x150 arc-minute bins
# with blockmean's summary parameter
df = pygmt.blockmean(data=data, region=region, spacing=spacing, summary="n")
grd = pygmt.xyz2grd(data=df, region=region, spacing=spacing)

fig.grdimage(
    grid=grd,
    region=region,
    frame=["af", "+tNumber of points inside each block"],
    cmap="batlow",
)
fig.coast(land="darkgray", transparency=40)
fig.plot(x=data.longitude, y=data.latitude, style="c0.3c", fill="white", pen="1p,black
↪")
fig.colorbar(frame="x+lcount")

fig.show()
```

Total running time of the script: (0 minutes 0.549 seconds)

5.8.2 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 the histogram (frequency count or percentage) can be selected via the `histtype` parameter.




```
import numpy as np
import pygmt

# Generate random elevation data from a normal distribution
rng = np.random.default_rng(seed=100)
mean = 100  # mean of distribution
stddev = 25  # standard deviation of distribution
data = rng.normal(loc=mean, scale=stddev, size=521)


fig = pygmt.Figure()
```

(continues on next page)

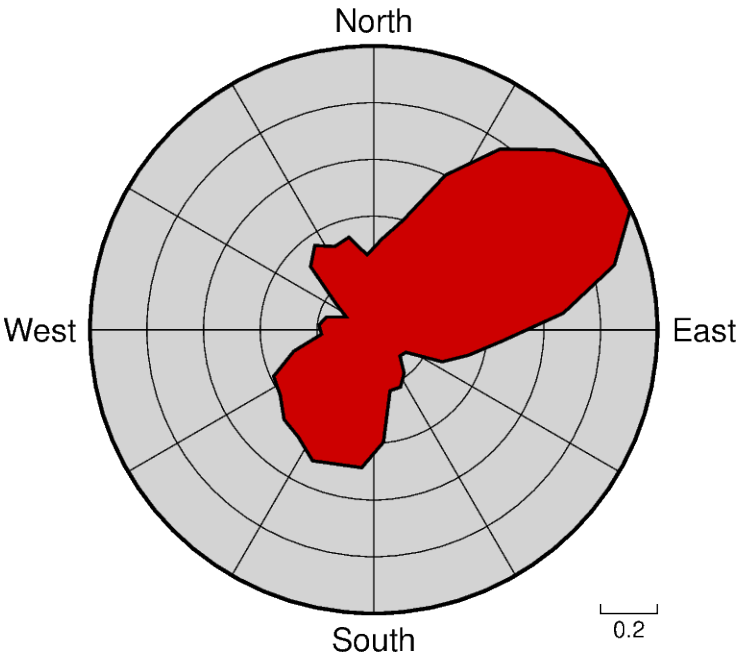
(continued from previous page)

```
fig.histogram(
    data=data,
    # Define the frame, add a title, and set the background color to
    # "lightgray". Add labels to the x-axis and y-axis
    frame=["WSne+tHistogram+glightgray", "x+lElevation (m)", "y+lCounts"],
    # Generate evenly spaced bins by increments of 5
    series=5,
    # Use "red3" as color fill for the bars
    fill="red3",
    # Use the pen parameter to draw the outlines with a width of 1 point
    pen="1p",
    # Choose histogram type 0, i.e., counts [Default]
    histtype=0,
)

fig.show()
```

Total running time of the script: (0 minutes 0.220 seconds)

5.8.3 Rose diagram

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




```
import pygmt

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

fig = pygmt.Figure()
```

(continues on next page)

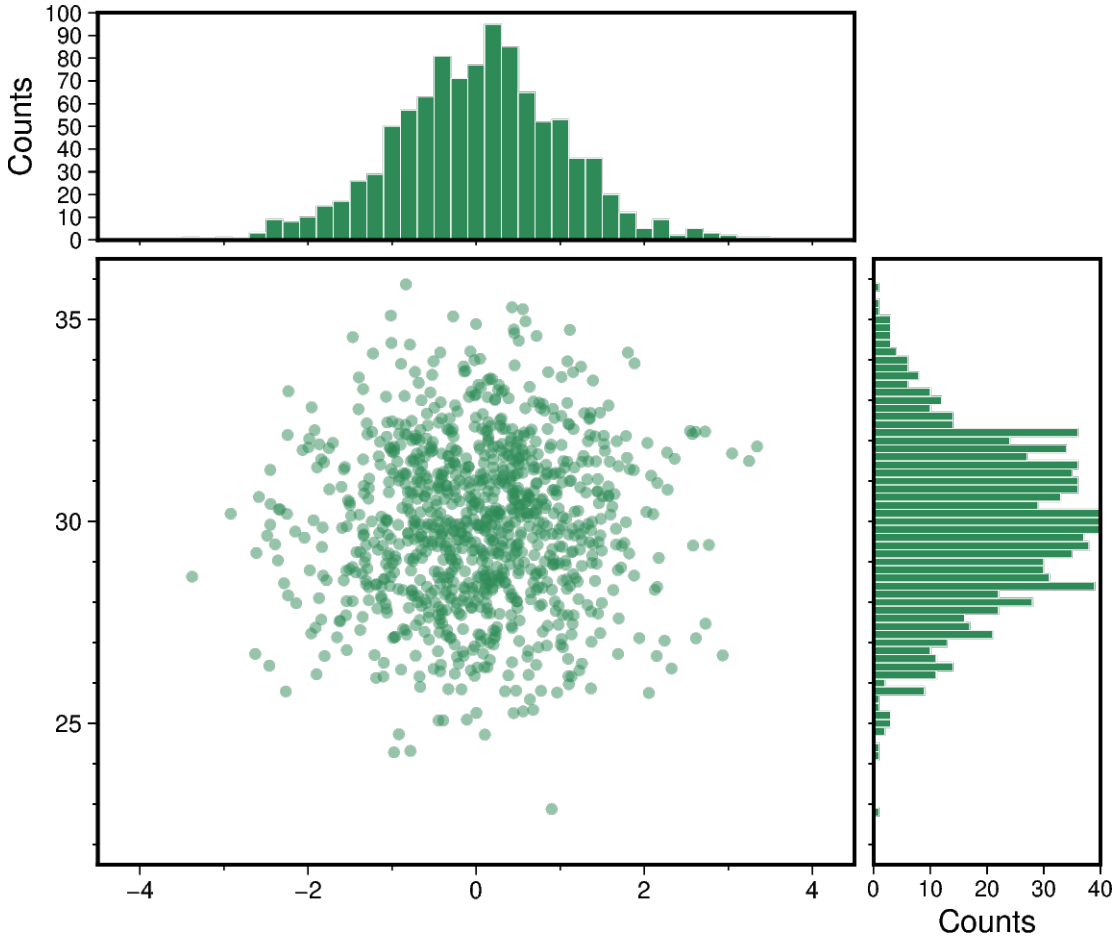
(continued from previous page)

```
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
    norm=True,
    # use red3 as color fill for the sectors
    fill="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 0.128 seconds)

5.8.4 Scatter plot with histograms

To create a scatter plot with histograms at the sides of the plot one can use *pygmt.Figure.plot* in combination with *pygmt.Figure.histogram*. The positions of the histograms are plotted by offsetting them from the main scatter plot using *pygmt.Figure.shift_origin*.




```
import numpy as np
import pygmt

# Generate random x, y coordinates from a standard normal distribution.
# x values are centered on 0 with a standard deviation of 1, and y values are centered
# on 30 with a standard deviation of 2.
rng = np.random.default_rng()
x = rng.normal(loc=0, scale=1, size=1000)
y = rng.normal(loc=30, scale=2, size=1000)

# Get axis limits from the data limits. Extend the limits by 0.5 to add some margin.
xmin = np.floor(x.min()) - 0.5
xmax = np.ceil(x.max()) + 0.5
ymin = np.floor(y.min()) - 0.5
ymax = np.ceil(y.max()) + 0.5

# Set fill color for symbols and bars.
fill = "seagreen"

# Set the dimensions of the scatter plot.
width, height = 10, 8

fig = pygmt.Figure()
```

(continues on next page)

(continued from previous page)

```
fig.basemap(
    region=[xmin, xmax, ymin, ymax],
    projection=f"X{width}/{height}",
    frame=["WSrt", "af"],
)

# Plot data points as circles with a diameter of 0.15 centimeters and set transparency
# level for all circles to deal with overplotting.
fig.plot(x=x, y=y, style="c0.15c", fill=fill, transparency=50)

# Shift the plot origin in y direction temporarily and add top margin histogram.
with fig.shift_origin(yshift=height + 0.25):
    fig.histogram(
        projection=f"X{width}/3",
        frame=["Wsrt", "xf", "yaf+lCounts"],
        # Give the same value for ymin and ymax to have them calculated automatically.
        region=[xmin, xmax, 0, 0],
        data=x,
        fill=fill,
        pen="0.1p,white",
        histtype=0,
        series=0.2,
    )

# Shift the plot origin in x direction temporarily and add right margin histogram.
with fig.shift_origin(xshift=width + 0.25):
    # Plot the horizontal histogram.
    fig.histogram(
        horizontal=True,
        projection=f"X3/{height}",
        # Note that the x- and y-axes are flipped, with the y-axis plotted␣
↪horizontally.
        frame=["wSrt", "xf", "yaf+lCounts"],
        region=[ymin, ymax, 0, 0],
        data=y,
        fill=fill,
        pen="0.1p,white",
        histtype=0,
        series=0.2,
    )
fig.show()
```

Total running time of the script: (0 minutes 0.270 seconds)

5.9 Plot embellishments

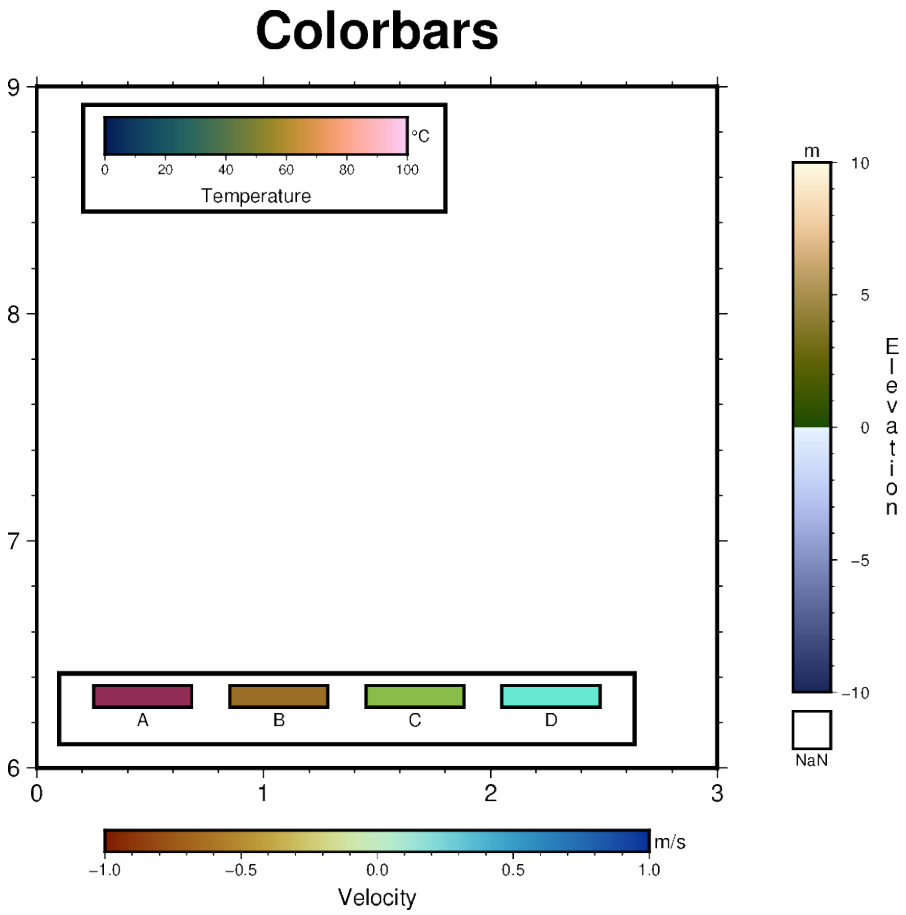
5.9.1 Colorbar

The *pygmt.Figure.colorbar* method creates a color scalebar. The colormap is set via the `cmap` parameter. A full list of available color palette tables can be found at https://docs.generic-mapping-tools.org/6.5/reference/cpts.html. Use the `frame` parameter to add labels to the **x** and **y** axes of the colorbar by appending **+l** followed by the desired text. To add and adjust the annotations (**a**) and ticks (**f**) append the letter followed by the desired interval. The placement of the

colorbar is set via the `position` parameter. There are the following options:

- **j/J**: placed inside/outside the plot bounding box 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 170° East, latitude 45° South.
- **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 Bottom Left (**BL**). Append `+h` to `position` to get a horizontal colorbar instead of a vertical one (`+v`).




```
import pygmt

fig = pygmt.Figure()
fig.basemap(region=[0, 3, 6, 9], projection="x3c", frame=["af", "WSne+tColorbars"])

# ============
# Create a colorbar designed for seismic tomography - roma
# Colorbar is placed at Bottom Center (BC) by default if no position is given
# Add quantity and unit as labels ("+l") to the x and y axes
# Add annotations ("+a") in steps of 0.5 and ticks ("+f") in steps of 0.1
fig.colorbar(cmap="roma", frame=["xa0.5f0.1+lVelocity", "y+lm/s"])

# ============
```

(continues on next page)

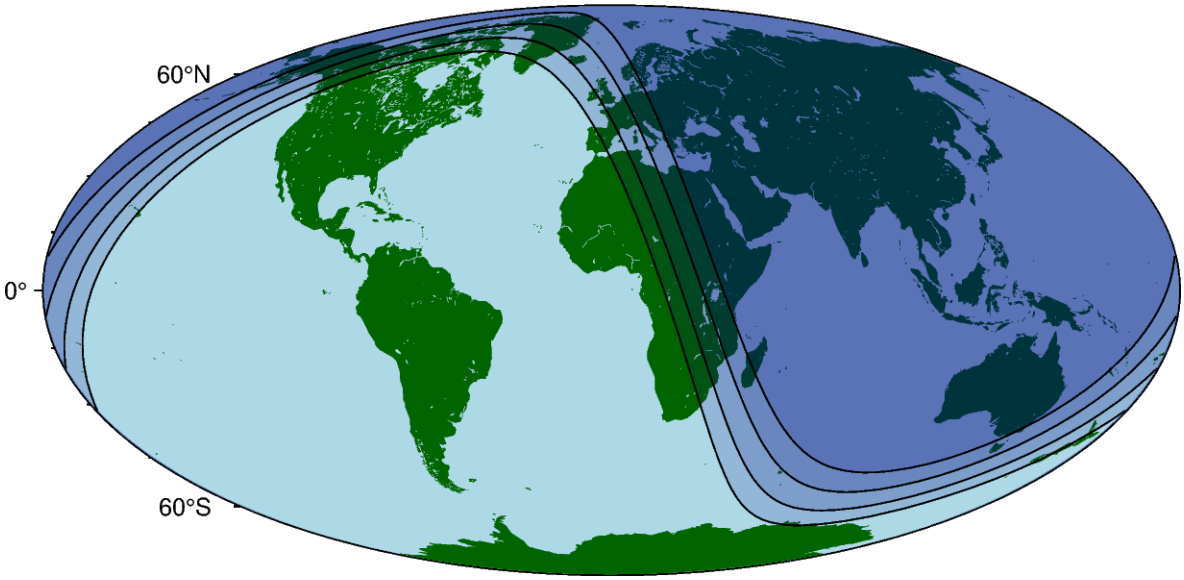
(continued from previous page)

```
# 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",
    box=True,
    frame=["x+lTemperature", "y+l°C"],
    scale=100,
)

# ============
# Create a colorbar suitable for surface topography - oleron
fig.colorbar(
    cmap="oleron",
    # Colorbar placed outside the plot bounding box (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,
)

# ============
# Create a colorbar suitable for categorical data - hawaii
# Set up the colormap
pygmt.makecpt(
    cmap="hawaii",
    series=[0, 3, 1],
    # Comma-separated string for the annotations of the colorbar
    color_model="+cA,B,C,D",
)
# Plot the colorbar
fig.colorbar(
    cmap=True,  # Use colormap set up above
    # Colorbar placed inside the plot bounding box (j) in the Bottom Left (BL) corner,
    # with an offset (+o) by 0.5 cm horizontally and 0.8 cm vertically from the anchor
    # point, and plotted horizontally (+h)
    position="jBL+o0.5c/0.8c+h",
    box=True,
    # Divide colorbar into equal-sized rectangles
    equalsize=0.5,
)

fig.show()
```

Total running time of the script: (0 minutes 0.177 seconds)

5.9.2 Day-night terminator 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.

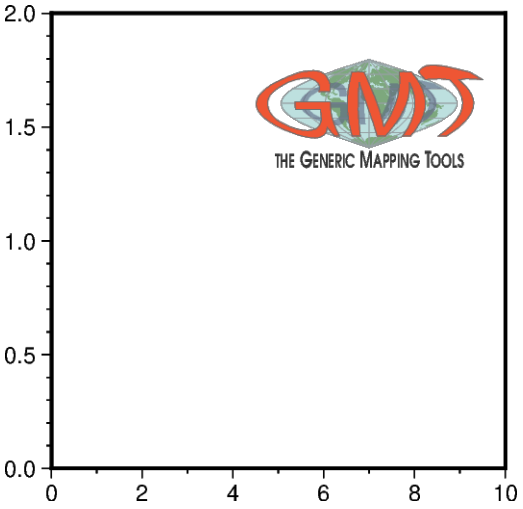



```
import datetime

import pygmt

fig = pygmt.Figure()
# Create a global map using the Mollweide projection, centered at 0°E, with a width of
# 15 centimeters.
fig.basemap(region="d", projection="W15c", frame=True)
fig.coast(land="darkgreen", water="lightblue")

# Set a time for the day-night terminator and twilights to 17:00 UTC on January 1,␣
↪2000
reference_time = datetime.datetime(
    year=2000, month=1, day=1, hour=17, minute=0, second=0
)

# Plot the day-night terminator and twilights
for terminator in ["day_night", "civil", "nautical", "astronomical"]:
    fig.solar(
        terminator=terminator,
        terminator_datetime=reference_time,
        # Set the fill for the night area to navy blue with 85 % transparency
        fill="navyblue@85",
        pen="0.5p",  # Set the outline to be 0.5 points thick
    )

fig.show()
```

Total running time of the script: (0 minutes 0.374 seconds)

5.9.3 GMT logo

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

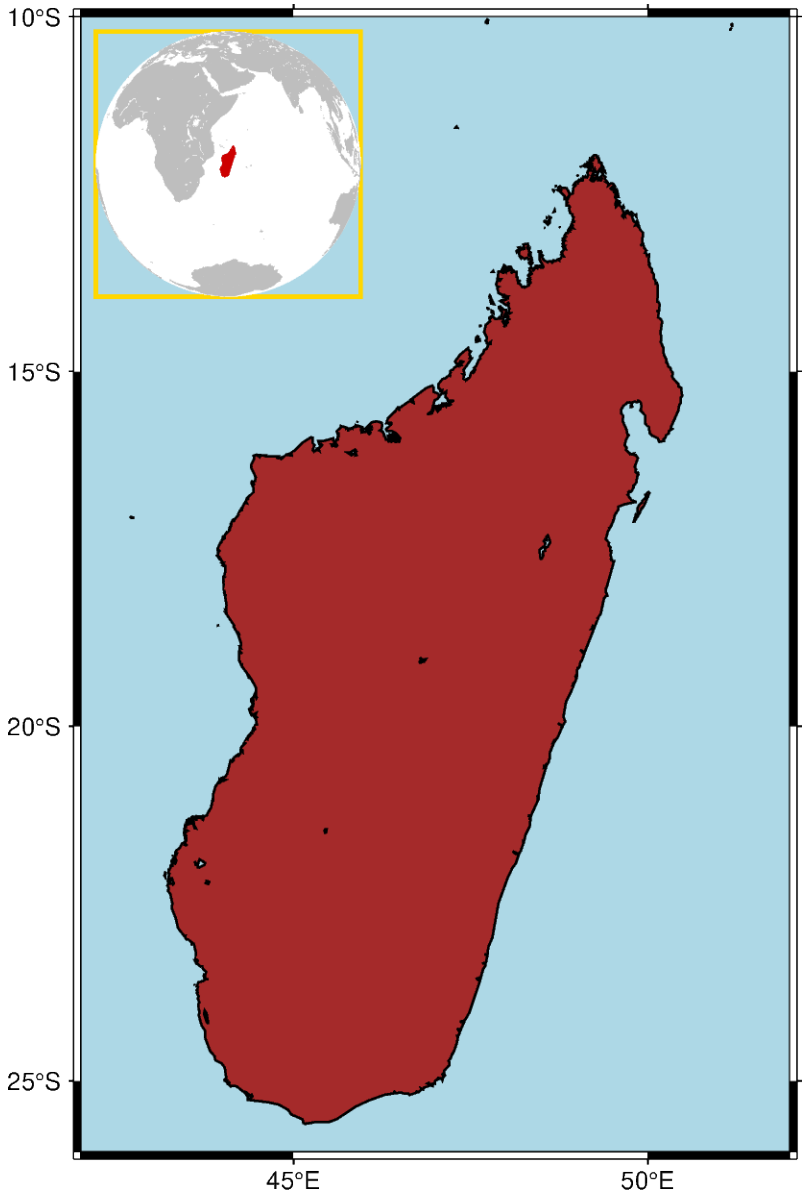



```
import pygmt

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

# Add the GMT logo in the Top Right (TR) corner of the current plot, scaled up to be 3
# centimeters 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 0.207 seconds)

5.9.4 Inset

The *pygmt.Figure.inset* method adds an inset figure inside a larger figure. The method 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.




```
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, placing it in the Top Left (TL) corner with a width of 3.5 cm and
# x- and y-offsets of 0.2 cm. The margin is set to 0, and the border is "gold" with a
# pen size of 1.5 points.
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".
```

(continues on next page)

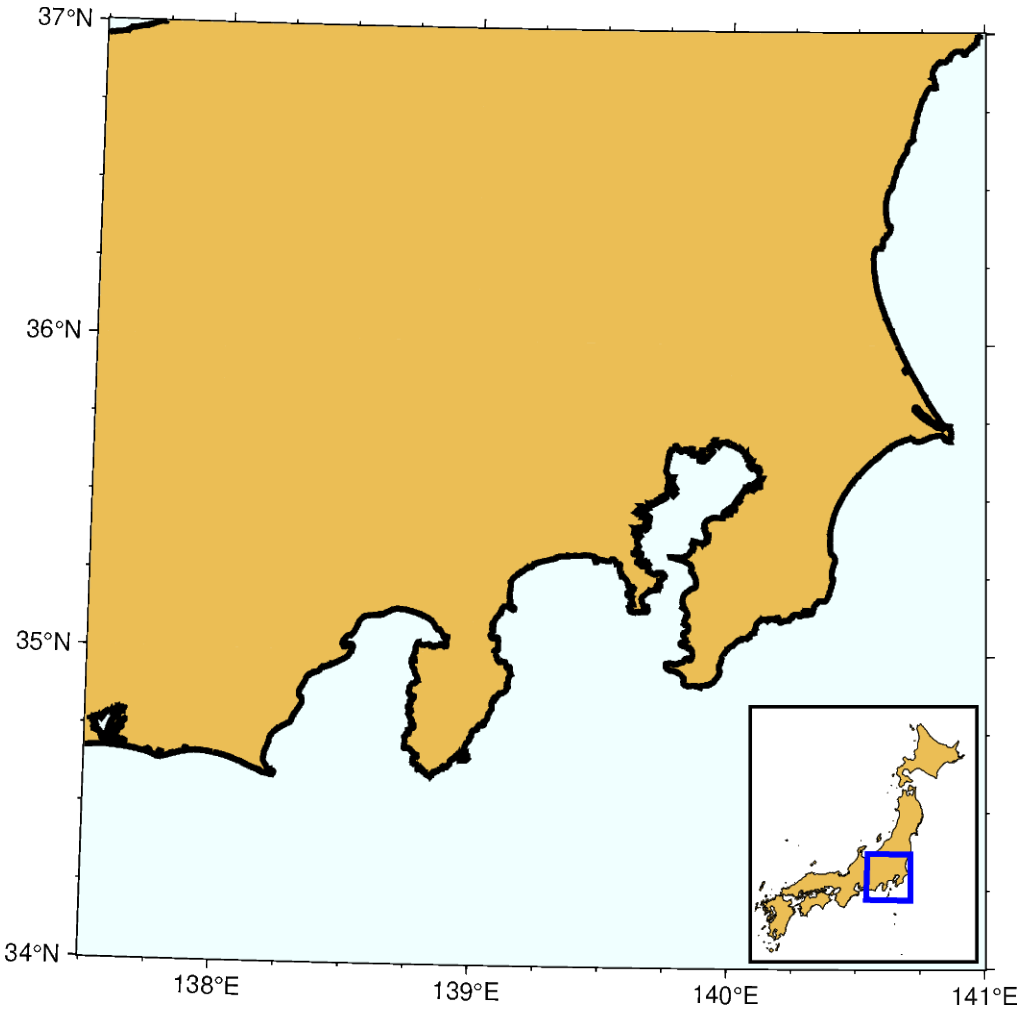
(continued from previous page)

```
    fig.coast(
        region="g",
        projection="G47/-20/?",
        land="gray",
        water="white",
        dcw="MG+gred3",
    )
fig.show()
```

Total running time of the script: (0 minutes 0.470 seconds)

5.9.5 Inset map showing a rectangular region

The *pygmt.Figure.inset* method adds an inset figure inside a larger figure. The method 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.




```
import pygmt

# Set the region of the main figure
```

(continues on next page)

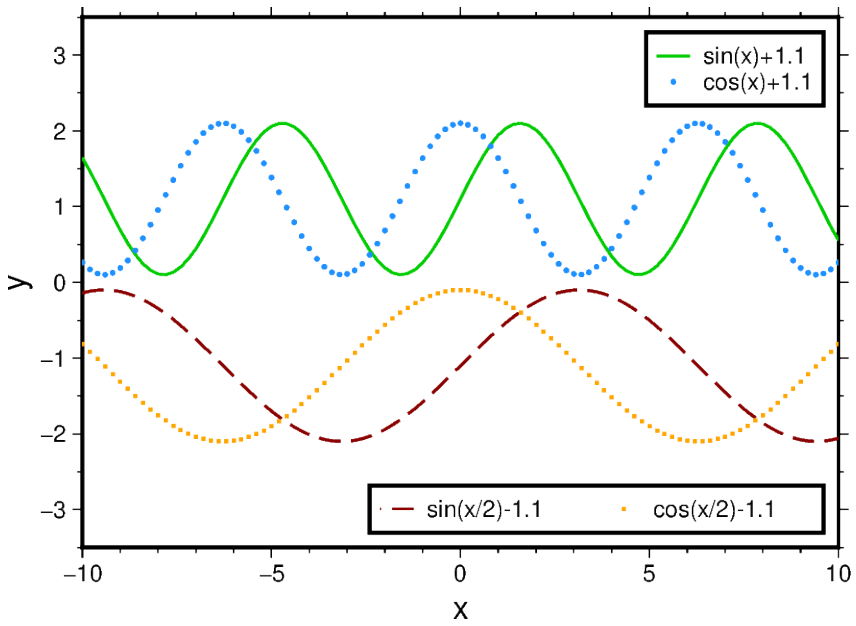
(continued from previous page)

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

# Create an inset map, placing it in the Bottom Right (BR) corner with x- and
# y-offsets of 0.1 cm, respectively.
# The inset map contains the Japan main land. "U54S/3c" means UTM projection
# with a map width of 3 cm. The inset width and height are automatically
# calculated from the specified ``region`` and ``projection`` parameters.
# Draws a rectangular box around the inset with a fill color of "white" and
# a pen of "1p".
with fig.inset(
    position="jBR+o0.1c",
    box="+gwhite+p1p",
    region=[129, 146, 30, 46],
    projection="U54S/3c",
):
    # Highlight the Japan area in "lightbrown"
    # and draw its outline with a pen of "0.2p".
    fig.coast(
        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 upper 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 0.336 seconds)

5.9.6 Legend

The *pygmt.Figure.legend* method can automatically create a legend for symbols plotted using *pygmt.Figure.plot*. A legend entry is only added when the `label` parameter is used to state the desired text. Optionally, to adjust the legend, users can append different modifiers. A list of all available modifiers can be found at https://docs.generic-mapping-tools.org/6.5/gmt.html#l-full. To create a multiple-column legend **+N** is used with the desired number of columns. For more complicated legends, users may want to write an ASCII file with instructions for the layout of the legend items and pass it to the `spec` parameter of *pygmt.Figure.legend*. For details on how to set up such a file, please see the GMT documentation at https://docs.generic-mapping-tools.org/6.5/legend.html#legend-codes.




```
import numpy as np
import pygmt

# Set up some test data
x = np.arange(-10, 10.2, 0.2)
y1 = np.sin(x) + 1.1
y2 = np.cos(x) + 1.1
y3 = np.sin(x / 2) - 1.1
y4 = np.cos(x / 2) - 1.1

# Create new Figure() object
fig = pygmt.Figure()

fig.basemap(
    projection="X10c/7c",
    region=[-10, 10, -3.5, 3.5],
    frame=["WSne", "xaf+lx", "ya1f0.5+ly"],
)

# ------------------------------------------------------------------------
# Top: Vertical legend (one column, default)

# Use the label parameter to state the text label for the legend entry
fig.plot(x=x, y=y1, pen="1p,green3", label="sin(x)+1.1")

fig.plot(x=x, y=y2, style="c0.07c", fill="dodgerblue", label="cos(x)+1.1")

# Add a legend to the plot; place it within the plot bounding box with both
# reference ("J") and anchor ("+j") points being the Top Right (TR) corner and an
# offset of 0.2 centimeters in x and y directions; surround the legend with a box
fig.legend(position="JTR+jTR+o0.2c", box=True)

# ------------------------------------------------------------------------
```

(continues on next page)

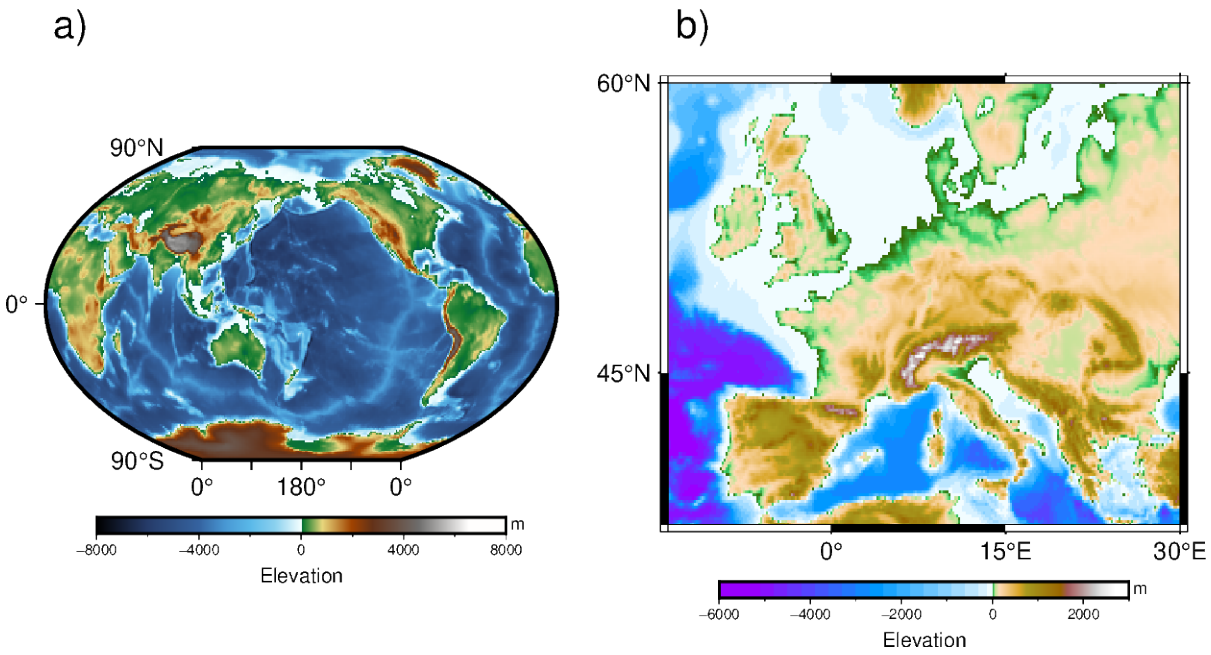
(continued from previous page)

```
# Bottom: Horizontal legend (here two columns)

# +N sets the number of columns corresponding to the given number, here 2
fig.plot(x=x, y=y3, pen="1p,darkred,-", label="sin(x/2)-1.1+N2")

fig.plot(x=x, y=y4, style="s0.07c", fill="orange", label="cos(x/2)-1.1")

# For a multi-column legend, users have to provide the width via "+w", here it is
# set to 6 centimeters; reference and anchor points are the Bottom Right (BR) corner
fig.legend(position="JBR+jBR+o0.2c+w6c", box=True)

fig.show()
```

Total running time of the script: (0 minutes 0.146 seconds)

5.9.7 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/6.5/reference/features.html#gmt-modern-mode-hierarchical-levels for details.




```
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 are
```

(continues on next page)

(continued from previous page)

```
# 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
    # function is a panel-level CPT
    with fig.set_panel(panel=0):
        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="a")
        fig.colorbar(frame=["a4000f2000", "x+lElevation", "y+lm"])
    # Activate the second panel so that the colormap created by the makecpt
    # function 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="a
↪")
        fig.colorbar(frame=["a2000f1000", "x+lElevation", "y+lm"])

fig.show()
```

Total running time of the script: (0 minutes 0.306 seconds)

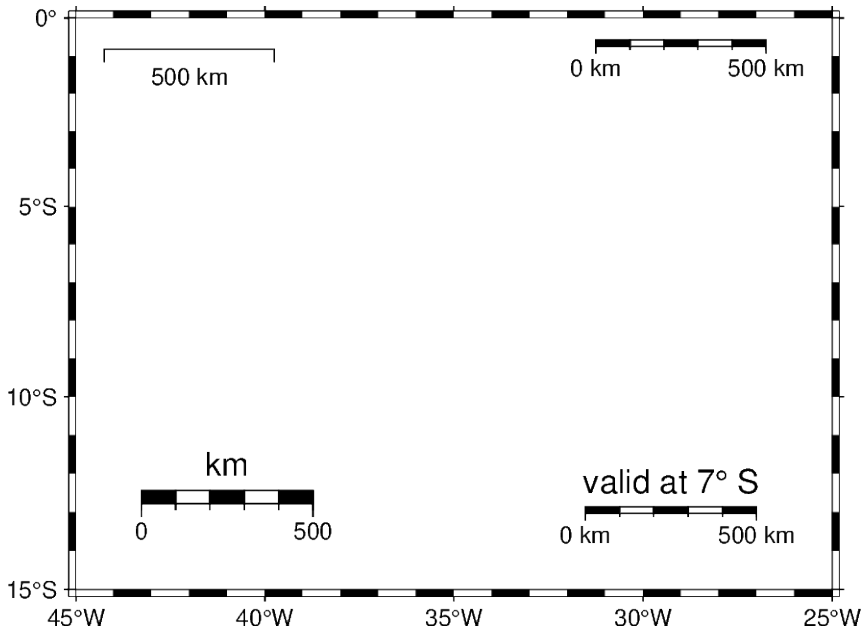
5.9.8 Scale bar

The `map_scale` parameter of the *pygmt.Figure.basemap* and *pygmt.Figure.coast* methods is used to add a scale bar to a map. This example shows how such a scale bar can be customized:

- position: **g**|**j**|**J**|**n**|**x**. Set the position of the reference point. Choose from
  - **g**: Give map coordinates as *longitude*/*latitude*.
  - **j**|**J**: Specify a two-character (order independent) code. Choose from vertical **T**(op), **M**(iddle), or **B**(ottom) and horizontal **L**(eft), **C**(entre), or **R**(ight). Lower / uppercase **j** / **J** mean inside / outside of the map bounding box.
  - **n**: Give normalized bounding box coordinates as *nx*/*ny*.
  - **x**: Give plot coordinates as *x*/*y*.
- length: **+w**. Give a distance value, and, optionally a distance unit. Choose from **e** (meters), **f** (feet), **k** (kilometers) [Default], **M** (statute miles), **n** (nautical miles), or **u** (US survey feet).
- origin: **+c**[*slon*/]*slat*. Control where on the map the scale bar applies. If **+c** is not given the reference point is used. If only **+c** is appended the middle of the map is used. Note that *slon* is only optional for projections with constant scale along parallels, e.g., Mercator projection.
- justify: **+j**. Set the anchor point. Specify a two-character (order independent) code. Choose from vertical **T**(op), **M**(iddle), or **B**(ottom) and horizontal **L**(eft), **C**(entre), or **R**(ight).
- offset: **+o***offset* or **+o***xoffset*/*yoffset*. Give either a common shift or individual shifts in x (longitude) and y (latitude) directions.
- height: Use MAP_SCALE_HEIGHT via *pygmt.config*.

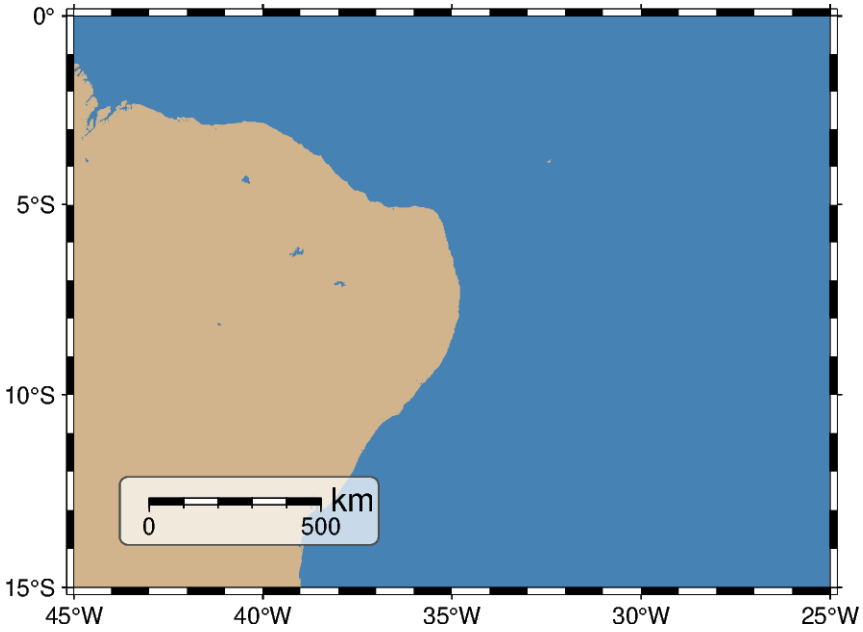
- fancy style: **+f**. Get a scale bar that looks like train tracks.
- unit: **+u**. Add the distance unit given via **+w** to the single distance values.
- label: **+l**. Add the distance unit given via **+w** as label. Append text to get a customized label instead.
- alignment: **+a**. Set the label alignment. Choose from **t**(op) [Default], **b**(ottom), **l**(eft), or **r**(ight).

```
import pygmt

# Create a new Figure instance
fig = pygmt.Figure()

# Mercator projection with 10 centimeters width
fig.basemap(region=[-45, -25, -15, 0], projection="M0/0/10c", frame=["WSne", "af"])

# ------------------------------------------------------------------------------
# Top Left: Add a plain scale bar
# It is placed based on geographic coordinates (g) 42° West and 1° South,
# applies at the reference point (+c is not given), and represents a
# length (+w) of 500 kilometers
fig.basemap(map_scale="g-42/-1+w500k")

# ------------------------------------------------------------------------------
# Top Right: Add a fancy scale bar
# It is placed based on normalized bounding box coordinates (n)
# Use a fancy style (+f) to get a scale bar that looks like train tracks
# Add the distance unit (+u) to the single distance values
fig.basemap(map_scale="n0.8/0.95+w500k+f+u")

# ------------------------------------------------------------------------------
# Bottom Left: Add a thick scale bar
# Adjust the GMT default parameter MAP_SCALE_HEIGHT locally (the change applies
# only to the code within the "with" statement)
# It applies (+c) at the middle of the map (no location is appended to +c)
# Without appending text, +l adds the distance unit as label
with pygmt.config(MAP_SCALE_HEIGHT="10p"):
    fig.basemap(map_scale="n0.2/0.15+c+w500k+f+l")

# ------------------------------------------------------------------------------
# Bottom Right: Add a scale bar valid for a specific location
# It is placed at BottomRight (j) using MiddleRight as anchor point (+j) with
# an offset (+o) of 1 centimeter in both x and y directions
# It applies (+c) at -7° South, add a customized label by appending text to +l
fig.basemap(map_scale="jBR+jMR+o1c/1c+c-7+w500k+f+u+lvalid at 7° S")

fig.show()
```



The `box` parameter allows surrounding the scale bar. This can be useful when adding a scale bar to a colorful map. To fill the box, append **+g** with the desired color (or pattern). The outline of the box can be adjusted by appending **+p** with the desired thickness, color, and style. To force rounded edges append **+r** with the desired radius.

```
# Create a new Figure instance
fig = pygmt.Figure()

fig.coast(
    region=[-45, -25, -15, 0],
    projection="M10c",
    land="tan",
    water="steelblue",
    frame=["WSne", "af"],
    # Set the label alignment (+a) to right (r)
    map_scale="jBL+o1c/1c+c-7+w500k+f+lkm+ar",
    # Fill the box in white with a transparency of 30 percent, add a solid
    # outline in darkgray (gray30) with a thickness of 0.5 points, and use
    # rounded edges with a radius of 3 points
    box="+gwhite@30+p0.5p,gray30,solid+r3p",
)

fig.show()
```

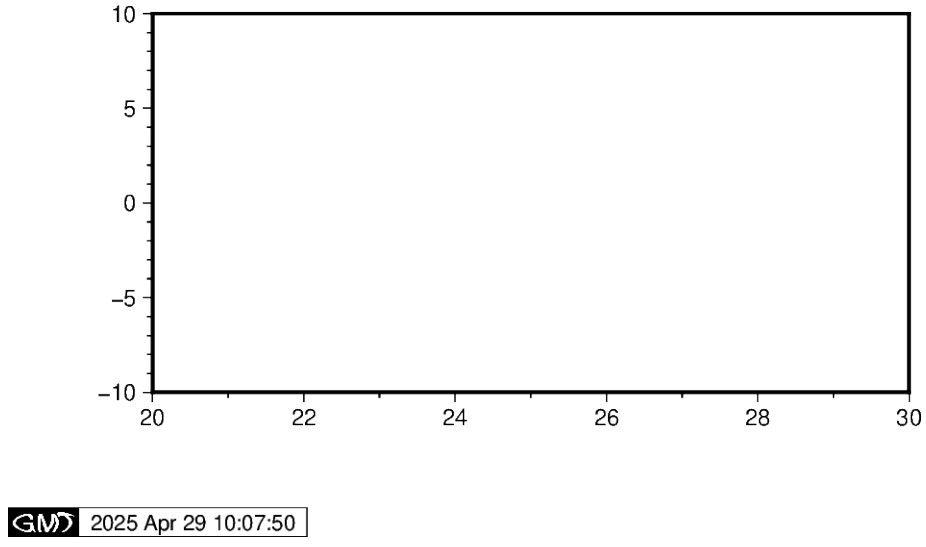


Total running time of the script: (0 minutes 0.336 seconds)

5.9.9 Timestamp

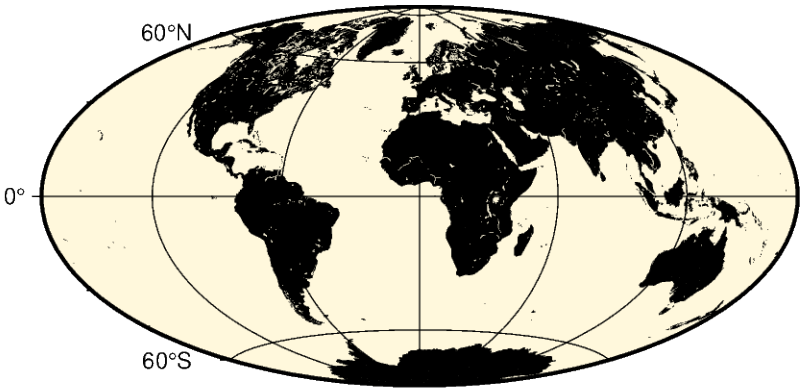
The *pygmt.Figure.timestamp* method can draw the GMT timestamp logo on the plot. The timestamp will always be shown relative to the Bottom Left (BL) corner of the plot. By default, the `offset` and `justify` parameters are set to `("-54p", "-54p")` (x, y directions) and `"BL"` (Bottom Left), respectively.

```
import os

import pygmt

fig = pygmt.Figure()
fig.basemap(region=[20, 30, -10, 10], projection="X10c/5c", frame=True)
fig.timestamp()
fig.show()
```



Additionally, a custom label can be added via the `label` parameter. The font can be defined via the `font` parameter and the timestamp string format via `timefmt`.

```
os.environ["TZ"] = "Pacific/Honolulu"  # optionally set the time zone

fig = pygmt.Figure()
fig.coast(region="d", projection="H10c", land="black", water="cornsilk", frame="afg")
fig.timestamp(
    label="Powered by PyGMT",
    justify="TL",
    font="Times-Bold",
    timefmt="%Y-%m-%dT%H:%M:%S%z",
)
fig.show()
```



Total running time of the script: (0 minutes 0.336 seconds)

CHAPTER

SIX

PROJECTIONS

PyGMT supports many map projections; see *GMT Map Projections* for an overview. Use the `projection` parameter to specify which one you want to use in all plotting methods. 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:

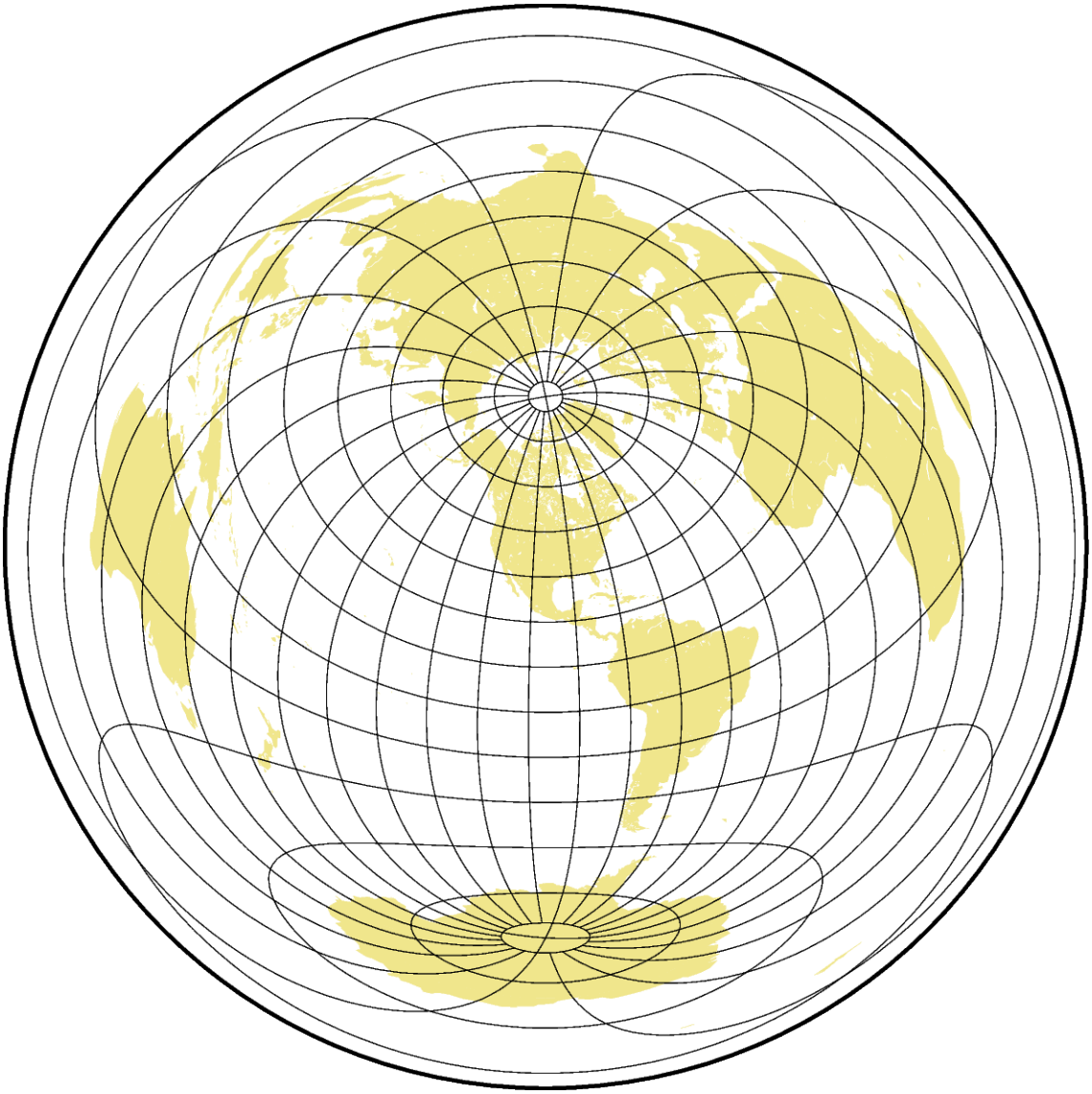
6.1 Azimuthal Projections

6.1.1 Azimuthal equidistant projection

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*.



```
coast [WARNING]: Fill/clip continent option (-G) may not work for this projection.
coast [WARNING]: If the antipode (0/4.67036e-310) is in the ocean then chances are␣
↪good it will work.
coast [WARNING]: Otherwise, avoid projection center coordinates that are exact␣
↪multiples of 80 degrees.
```

```
import pygmt

fig = pygmt.Figure()
fig.coast(
    region="g", projection="E-100/40/15c", frame="afg", land="khaki", water="white"
)
```

(continues on next page)

(continued from previous page)

```
fig.show()
```

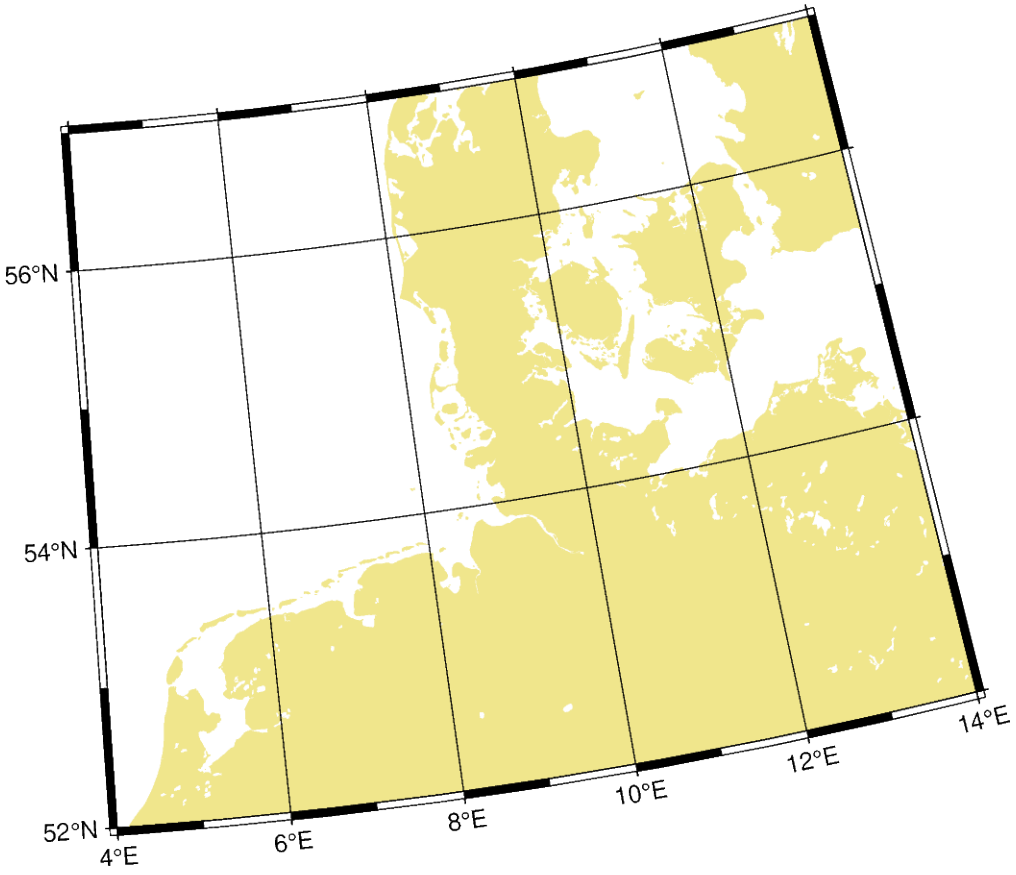
Total running time of the script: (0 minutes 0.360 seconds)

6.1.2 General stereographic projection

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.




```
import pygmt

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

(continues on next page)

(continued from previous page)

```
    water="white",
)
fig.show()
```

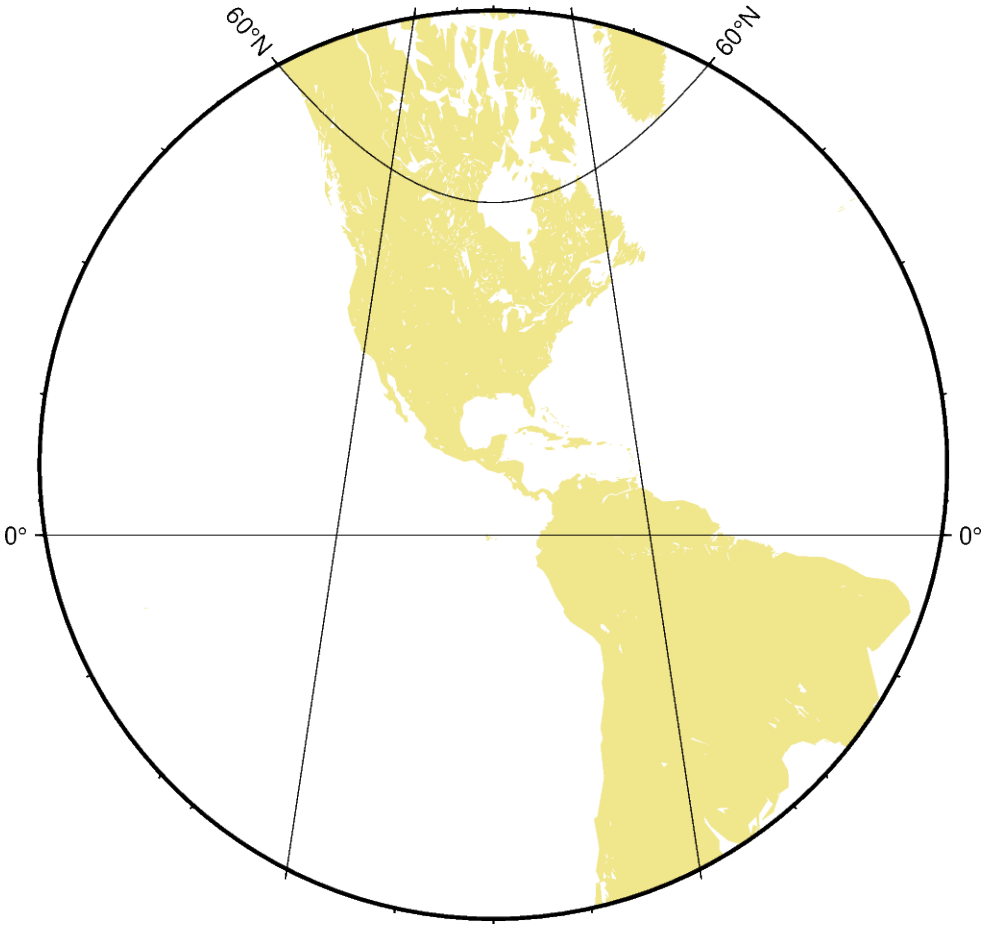
Total running time of the script: (0 minutes 0.192 seconds)

6.1.3 Gnomonic projection

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.




```
import pygmt

fig = pygmt.Figure()
fig.coast(
    region="g",
```

(continues on next page)

(continued from previous page)

```
    projection="F-90/15/12c",
    frame="afg",
    land="khaki",
    water="white",
)
fig.show()
```

Total running time of the script: (0 minutes 0.236 seconds)

6.1.4 Lambert azimuthal equal-area projection

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.



```
import pygmt

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

(continues on next page)

(continued from previous page)

```
    region="g",
    projection="A30/-20/60/12c",
    frame="afg",
    land="khaki",
    water="white",
)
fig.show()
```

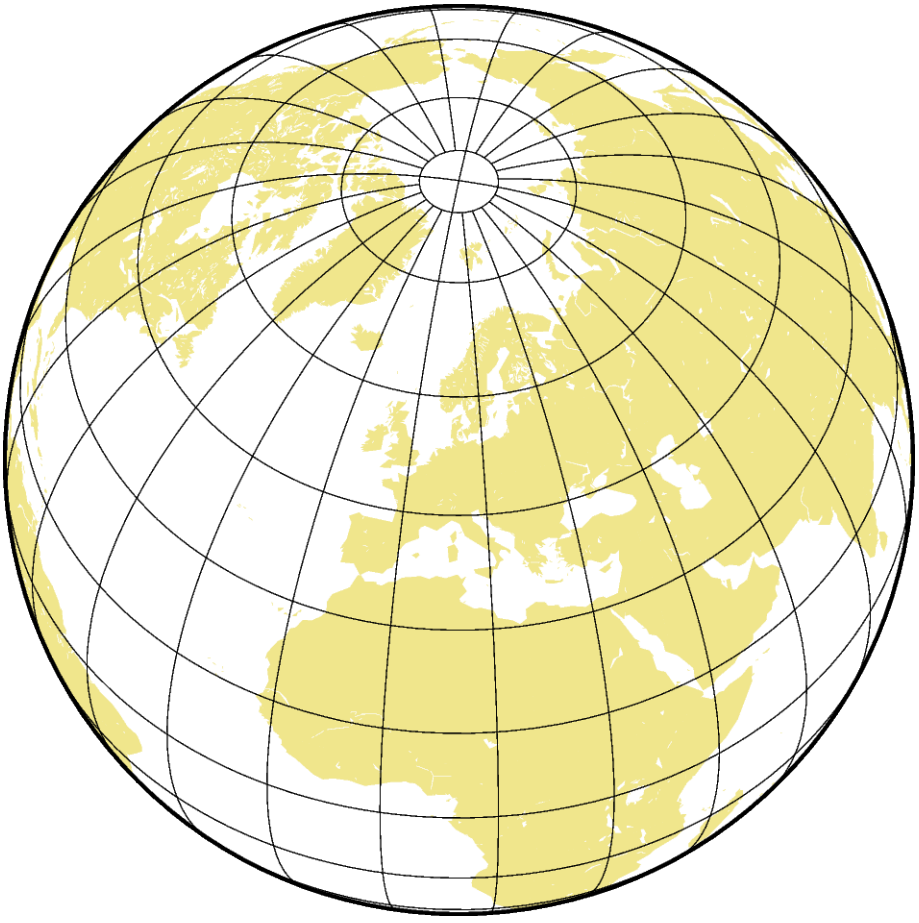
Total running time of the script: (0 minutes 0.218 seconds)

6.1.5 Orthographic projection

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.



```
import pygmt

fig = pygmt.Figure()
```

(continues on next page)

(continued from previous page)

```
fig.coast(
    region="g",
    projection="G10/52/12c",
    frame="afg",
    land="khaki",
    water="white",
)
fig.show()
```

Total running time of the script: (0 minutes 0.246 seconds)

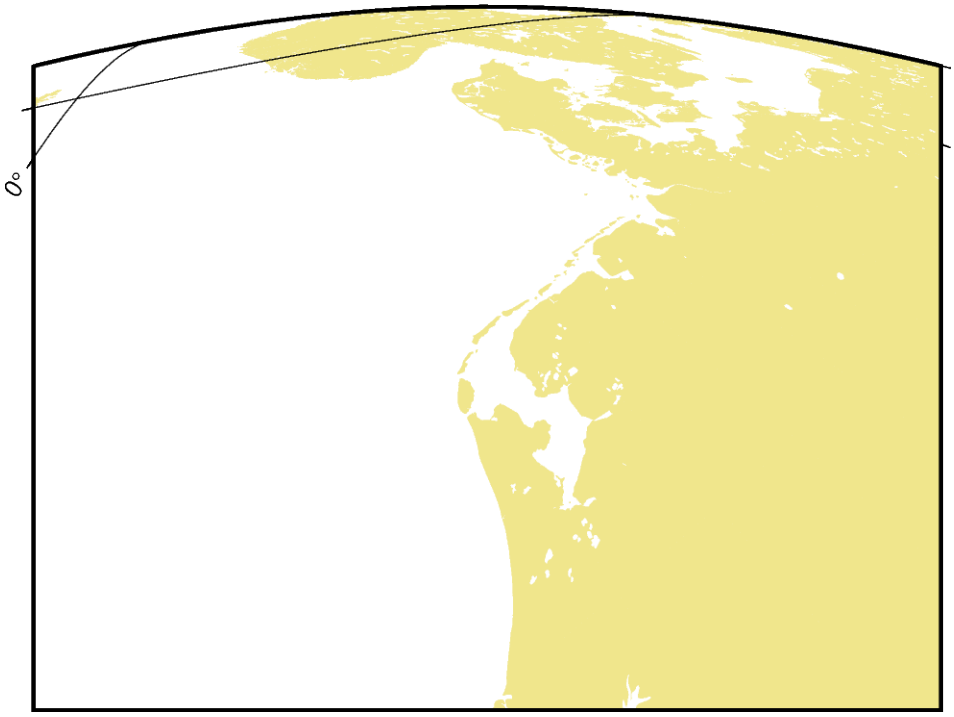
6.1.6 Perspective projection

The 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/scale*[**+a***azimuth*][**+t***tilt*][**+v***vwidth/vheight*][**+w***twist*][**+z***altitude*] or **G***lon0/lat0/width*[**+a***azimuth*][**+t***tilt*][**+v***vwidth/vheight*][**+v**

The projection type is set with **g** or **G**. *lon0/lat0* specifies the projection center and *scale* or *width* determine the size of the figure. With **+a***azimuth* the direction (in degrees) in which you are looking is specified, measured clockwise from north. **+t***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. The viewport angle in degrees is described via **+v***vwidth/vheight* and **+w***twist* is the clockwise rotation of the image (in degrees). **+z***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).

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.




```
import pygmt

fig = pygmt.Figure()
```

(continues on next page)

(continued from previous page)

```
fig.coast(
    region="g",
    projection="G4/52/12c+a30+t45+v60/60+w0+z250",
    frame="afg",
    land="khaki",
    water="white",
)
fig.show()
```

Total running time of the script: (0 minutes 1.508 seconds)

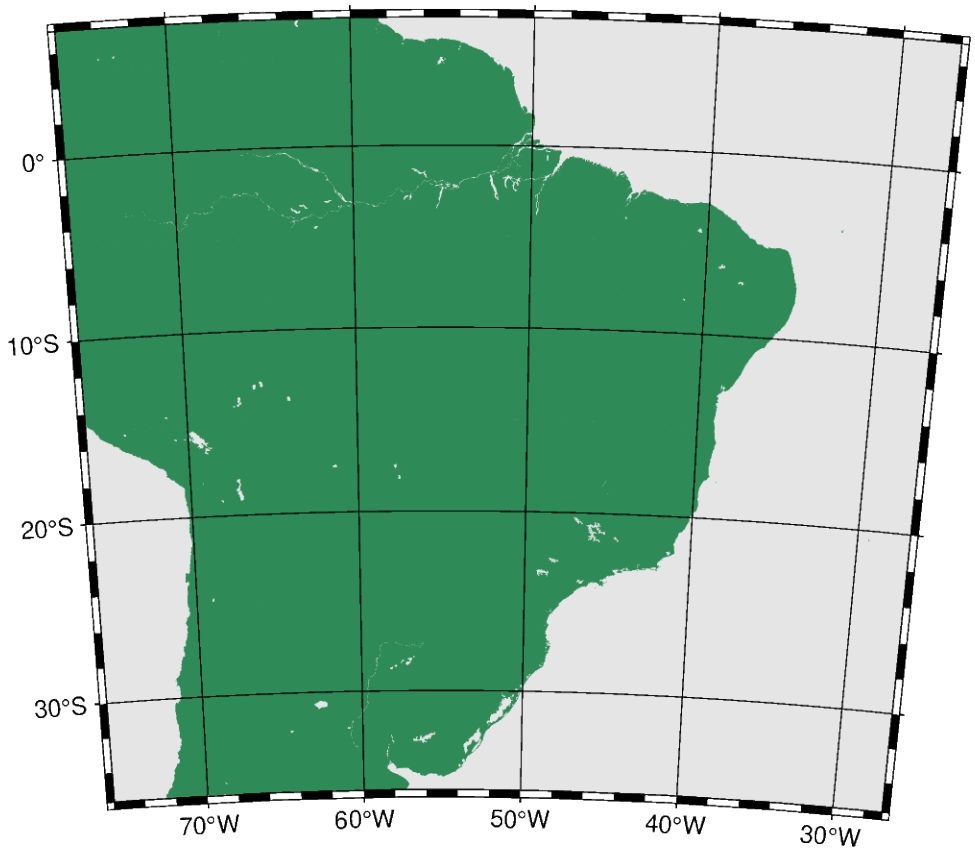
6.2 Conic Projections

6.2.1 Albers conic equal-area projection

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*.



```
import pygmt

fig = pygmt.Figure()
# Use the ISO country code for Brazil and add a padding of 2 degrees (+R2)
fig.coast(
    region="BR+R2",
    projection="B-55/-15/-25/0/12c",
    frame="afg",
    land="seagreen",
    water="gray90",
)
fig.show()
```

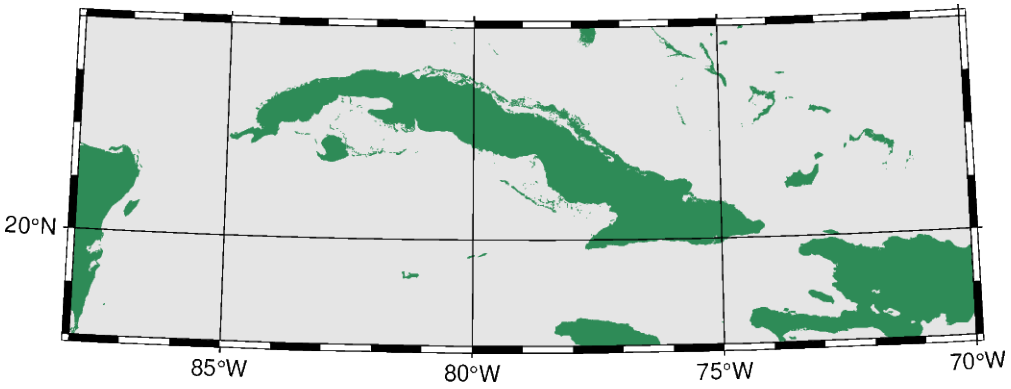
Total running time of the script: (0 minutes 0.286 seconds)

6.2.2 Equidistant conic projection

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*.




```
import pygmt

fig = pygmt.Figure()
fig.coast(
    region=[-88, -70, 18, 24],
    projection="D-79/21/19/23/12c",
    frame="afg",
    land="seagreen",
    water="gray90",
)
fig.show()
```

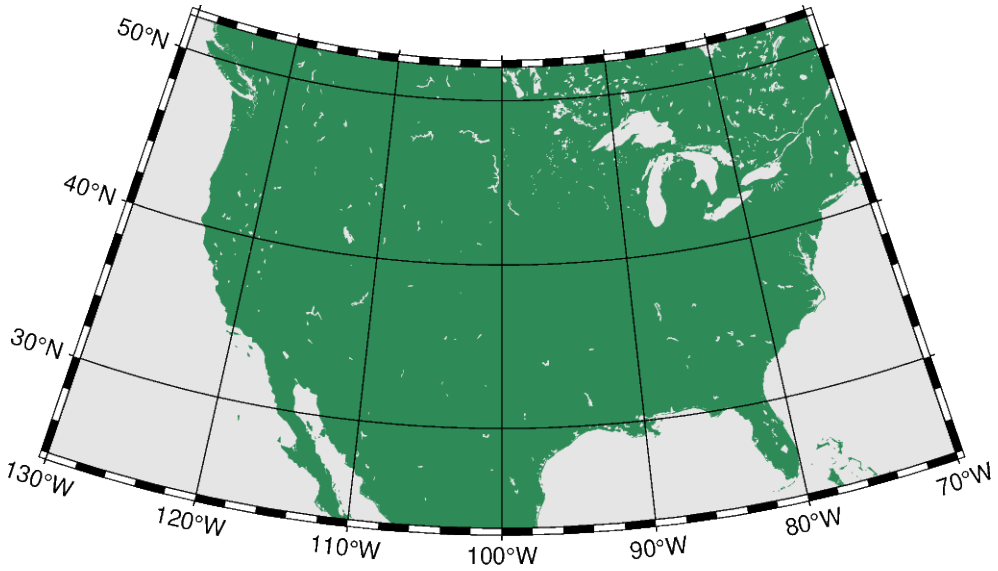
Total running time of the script: (0 minutes 0.154 seconds)

6.2.3 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*.




```
import pygmt

fig = pygmt.Figure()
fig.coast(
    region=[-130, -70, 24, 52],
    projection="L-100/35/33/45/12c",
    frame="afg",
    land="seagreen",
    water="gray90",
)

fig.show()
```

Total running time of the script: (0 minutes 0.186 seconds)

6.2.4 Polyconic projection

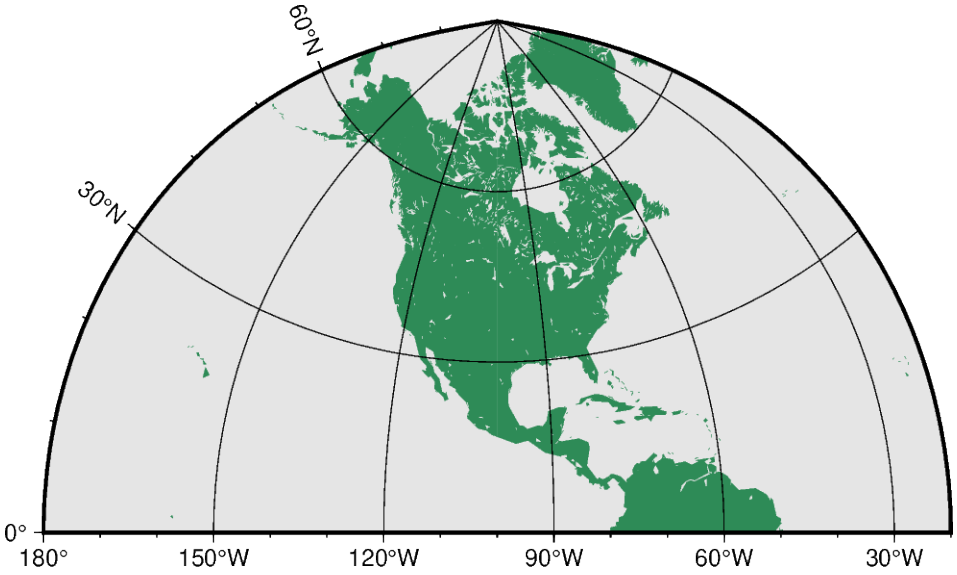
The polyconic projection, in Europe usually referred to as the American polyconic projection, was introduced shortly before 1820 by the Swiss-American cartographer Ferdinand Rodulph Hassler (1770-1843). As head of the Survey of the Coast, he was looking for a projection that would give the least distortion 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/[*lon0*/[*lat0*/]]*scale* or **Poly**/[*lon0*/[*lat0*/]]*width*

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




```
import pygmt

fig = pygmt.Figure()
fig.coast(
    region=[-180, -20, 0, 90],
    projection="Poly/12c",
    frame="afg",
    land="seagreen",
    water="gray90",
)
fig.show()
```

Total running time of the script: (0 minutes 0.165 seconds)

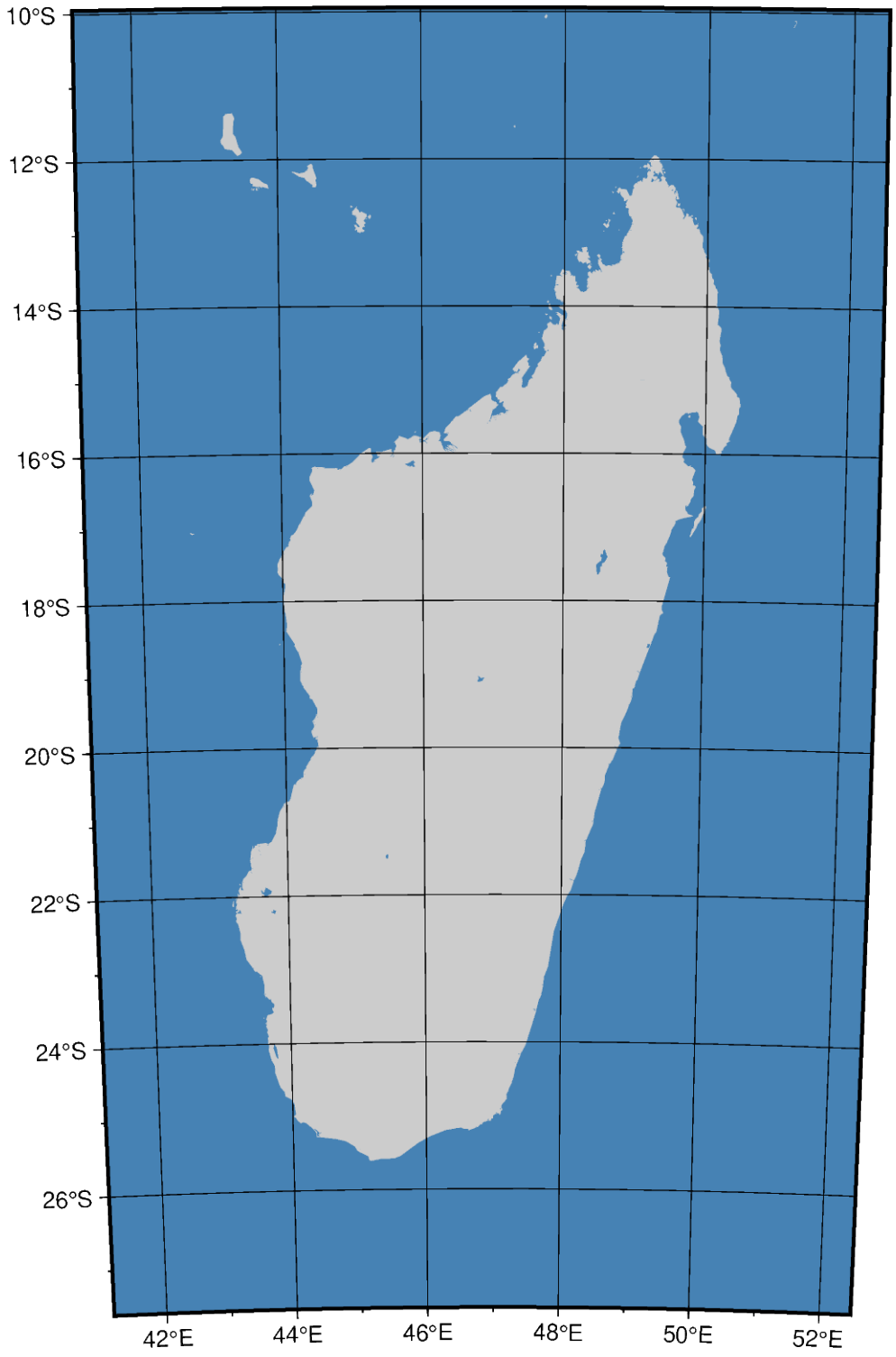
6.3 Cylindric Projections

6.3.1 Cassini cylindrical projection

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*.




```
import pygmt

fig = pygmt.Figure()
# Use the ISO code for Madagascar (MG) and pad it by 2 degrees (+R2)
fig.coast(
    region="MG+R2",
```

(continues on next page)

(continued from previous page)

```
    projection="C47/-19/12c",
    frame="afg",
    land="gray80",
    water="steelblue",
)
fig.show()
```

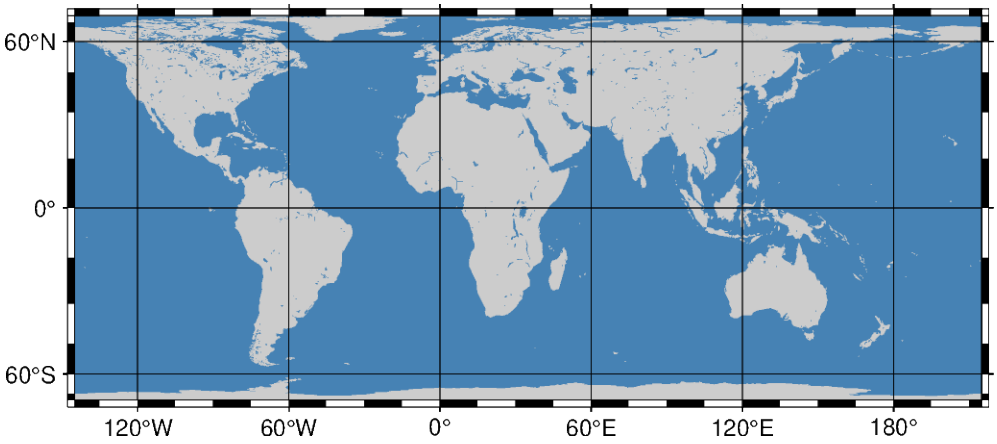
Total running time of the script: (0 minutes 0.293 seconds)

6.3.2 Cylindrical equal-area projection

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*.




```
import pygmt

fig = pygmt.Figure()
# Use region "d" to specify global region (-180/180/-90/90)
fig.coast(
    region="d",
    projection="Y35/30/12c",
    frame="afg",
    land="gray80",
    water="steelblue",
)
fig.show()
```

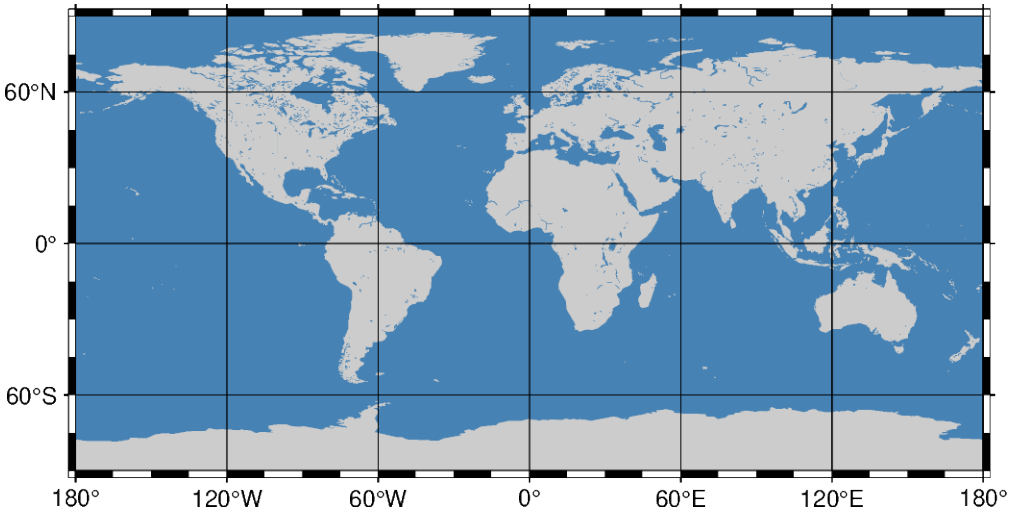
Total running time of the script: (0 minutes 0.194 seconds)

6.3.3 Cylindrical equidistant projection

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[*lon0*/[*lat0*/]]*scale* or **Q**[*lon0*/[*lat0*/]]*width*

The projection is set with **q** or **Q**, and the figure size is set with *scale* or *width*. Optionally, the central meridian can be set with *lon0* [Default is the middle of the map]. Optionally, the standard parallel can be set with *lat0* [Default is the equator]. When supplied, the central meridian must be supplied as well.




```
import pygmt

fig = pygmt.Figure()
# Use region "d" to specify global region (-180/180/-90/90)
fig.coast(
    region="d",
    projection="Q12c",
    frame="afg",
    land="gray80",
    water="steelblue",
)
fig.show()
```

Total running time of the script: (0 minutes 0.198 seconds)

6.3.4 Cylindrical stereographic projection

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.

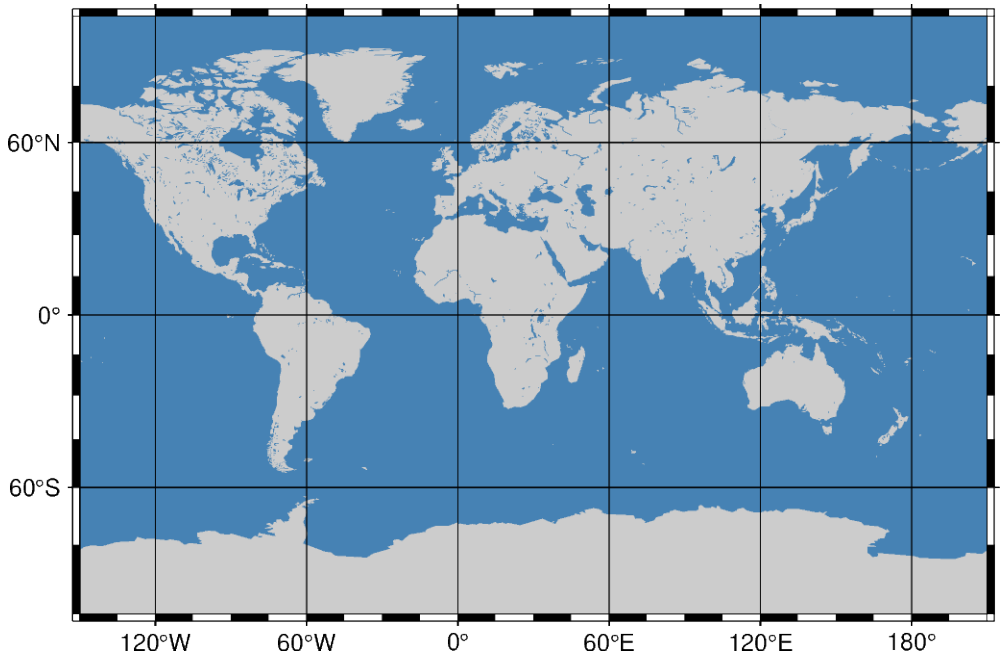
cyl_stere/[*lon0*/[*lat0*/]]*scale* or **Cyl_stere/**[*lon0*/[*lat0*/]]*width*

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




```
import pygmt

fig = pygmt.Figure()
fig.coast(
    region="g",
    projection="Cyl_stere/30/-20/12c",
    frame="afg",
    land="gray80",
    water="steelblue",
)
fig.show()
```

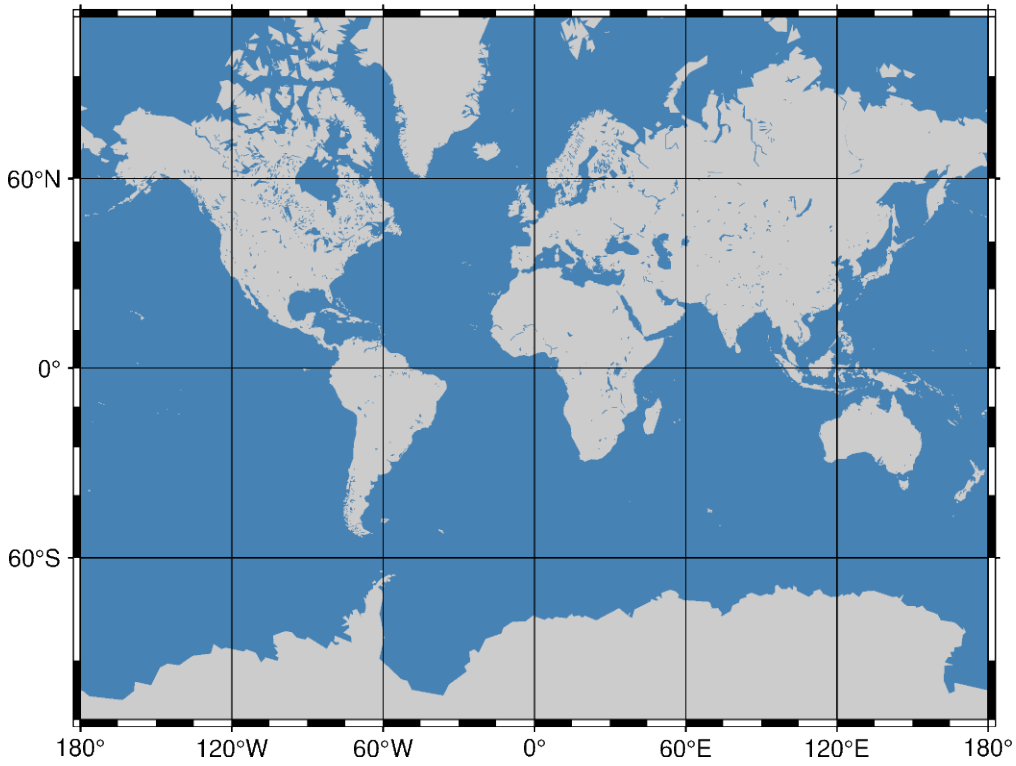
Total running time of the script: (0 minutes 0.209 seconds)

6.3.5 Mercator projection

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*.




```
import pygmt

fig = pygmt.Figure()
fig.coast(
    region=[0, 360, -80, 80],
    projection="M0/0/12c",
    frame="afg",
    land="gray80",
    water="steelblue",
)
fig.show()
```

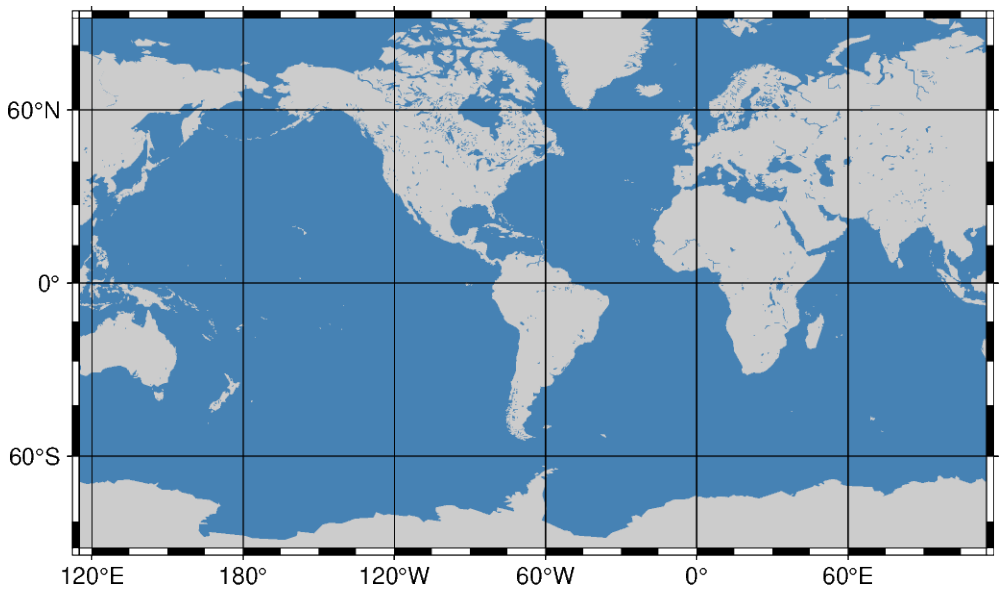
Total running time of the script: (0 minutes 0.216 seconds)

6.3.6 Miller cylindrical projection

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*.




```
import pygmt

fig = pygmt.Figure()
fig.coast(
    region=[-180, 180, -80, 80],
    projection="J-65/12c",
    frame="afg",
    land="gray80",
    water="steelblue",
)
fig.show()
```

Total running time of the script: (0 minutes 0.204 seconds)

6.3.7 Oblique Mercator projection

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.

The projection is set with **o** or **O**. There are three different specification ways (**a**|**A**, **b**|**B**, **c**|**C**) available. For all three definitions, the uppercase letter mean the projection pole is set in the southern hemisphere [Default is northern hemisphere]. Align the y-axis with the optional modifier **+v**. The figure size is set with *scale* or *width*.

1. Using the origin and azimuth

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

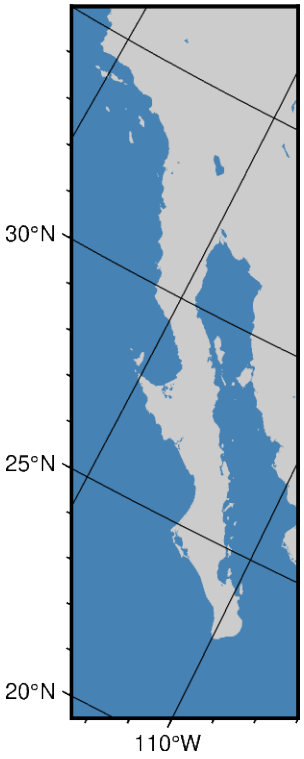
The central meridian is set by *lon0/lat0*. The oblique equator is set by *azimuth*.

```
import pygmt

fig = pygmt.Figure()
fig.coast(
    projection="Oa-120/25/-30/3c+v",
    # Set bottom left and top right coordinates of the figure with "+r"
```

(continues on next page)

(continued from previous page)

```
    region="-122/35/-107/22+r",
    frame="afg",
    land="gray80",
    water="steelblue",
)
fig.show()
```

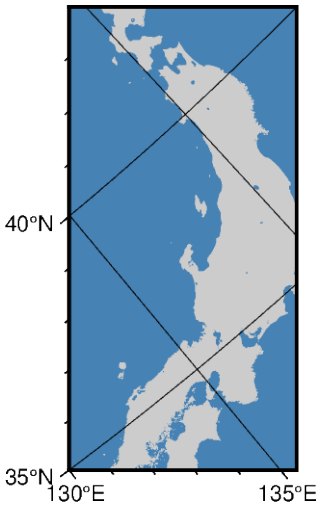


2. Using two points

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

The central meridian is set by *lon0/lat0*. The oblique equator is set by *lon1/lat1*.

```
fig = pygmt.Figure()
fig.coast(
    projection="Ob130/35/25/35/3c",
    region="130/35/145/40+r",
    frame="afg",
    land="gray80",
    water="steelblue",
)
fig.show()
```

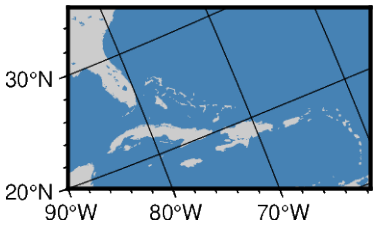


3. Using the origin and projection pole

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

The central meridian is set by *lon0/lat0*. The projection pole is set by *lonp/latp*.

```
fig = pygmt.Figure()
fig.coast(
    projection="Oc280/25.5/22/69/4c",
    region="270/20/305/25+r",
    frame="afg",
    land="gray80",
    water="steelblue",
)
fig.show()
```



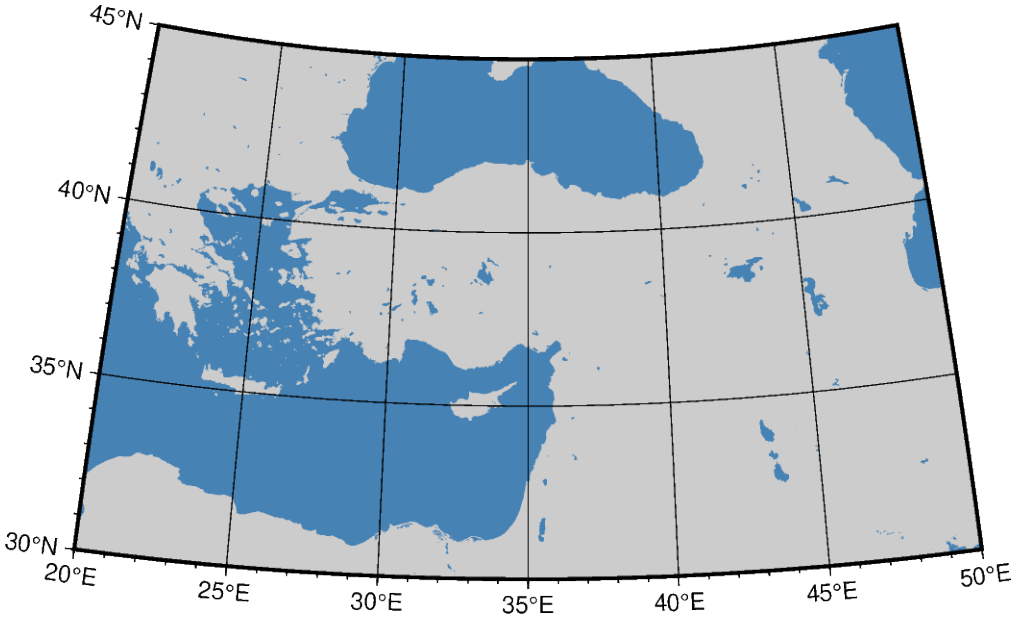
Total running time of the script: (0 minutes 0.410 seconds)

6.3.8 Transverse Mercator projection

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*.




```
import pygmt

fig = pygmt.Figure()
fig.coast(
    region=[20, 50, 30, 45],
    projection="T35/12c",
    frame="afg",
    land="gray80",
    water="steelblue",
)
fig.show()
```

Total running time of the script: (0 minutes 0.176 seconds)

6.3.9 Universal Transverse Mercator projection

A particular subset of the *transverse Merctor* 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° (in longitude) 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. See Figure *Universal Transverse Mercator* for all zone designations.

In order to minimize the distortion in any given zone, a scale factor of 0.9996 has been factored into the formulae (although a standard, you can change this with PROJ_SCALE_FACTOR). This makes the UTM projection a *secant* projection and not a *tangent* projection like the *transverse Merctor*. 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. For larger regions we use the conformal latitude in the general spherical formulae instead.

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*.

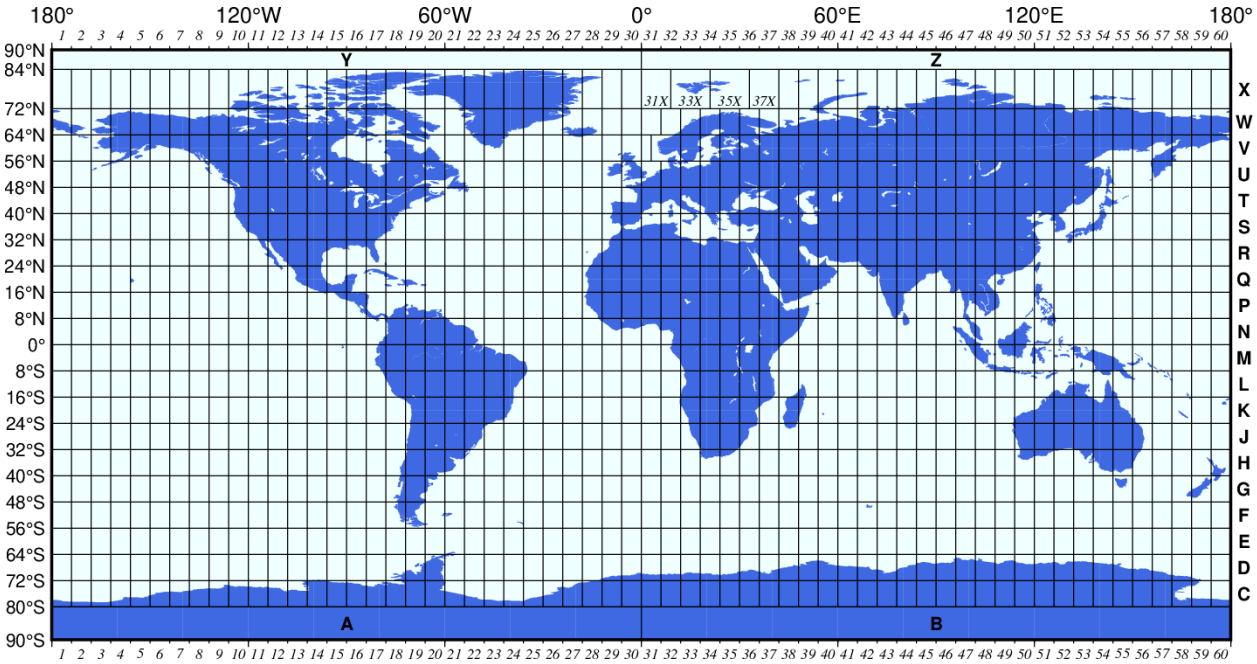
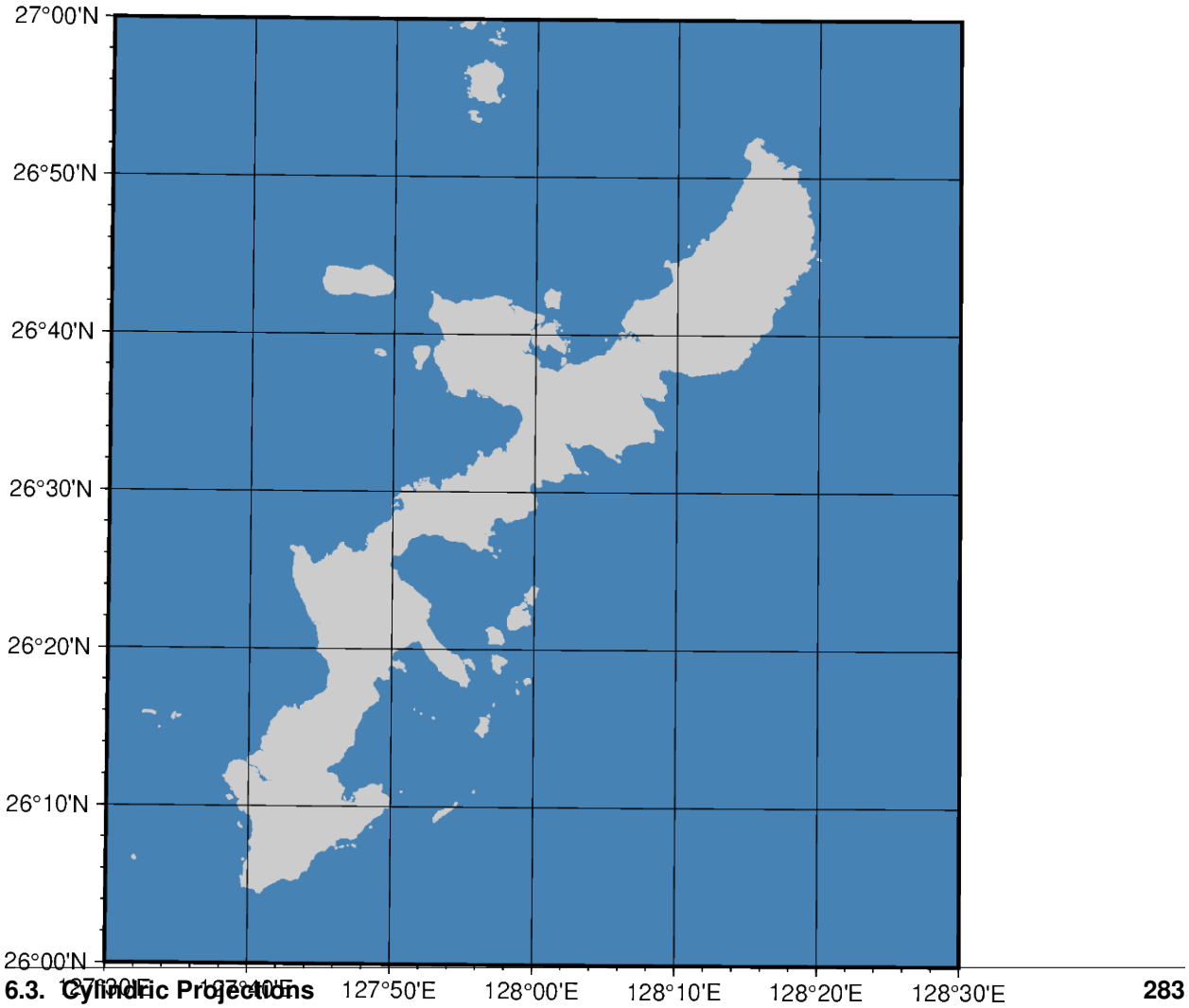


Fig. 1: Universal Transverse Mercator zone layout.



```
import pygmt

fig = pygmt.Figure()
# UTM Zone is set to 52R
fig.coast(
    region=[127.5, 128.5, 26, 27],
    projection="U52R/12c",
    frame="afg",
    land="gray80",
    water="steelblue",
)
fig.show()
```

Total running time of the script: (0 minutes 0.200 seconds)

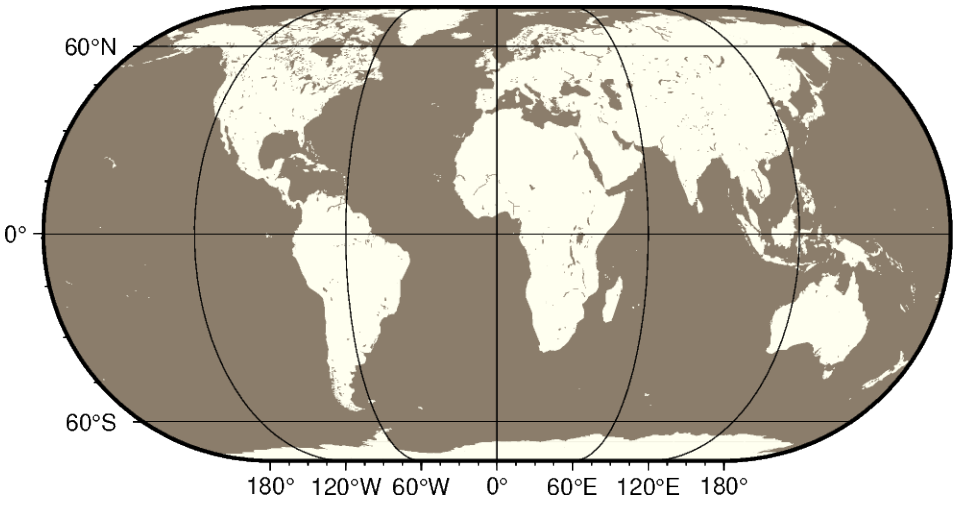
6.4 Miscellaneous Projections

6.4.1 Eckert IV equal-area projection

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*.




```
import pygmt

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

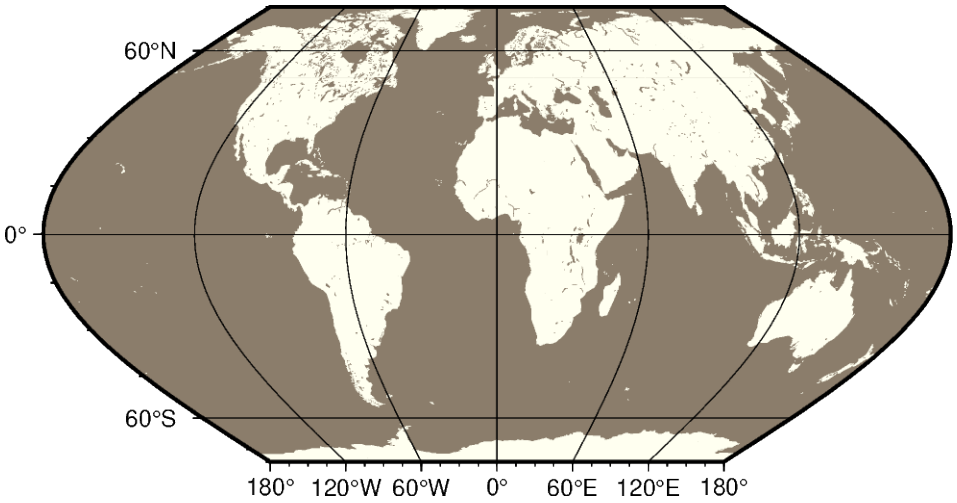
Total running time of the script: (0 minutes 0.214 seconds)

6.4.2 Eckert VI equal-area projection

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*.




```
import pygmt

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

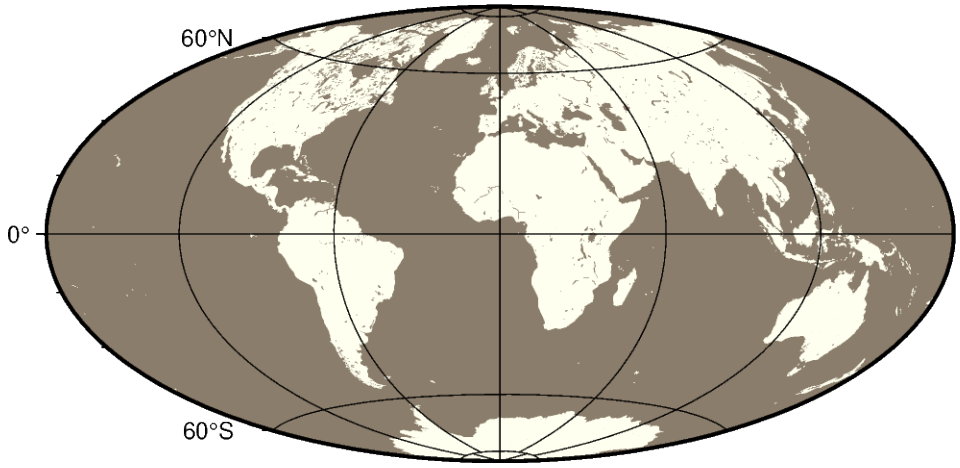
Total running time of the script: (0 minutes 0.210 seconds)

6.4.3 Hammer projection

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*.




```
import pygmt

fig = pygmt.Figure()
# Use region "d" to specify global region (-180/180/-90/90)
fig.coast(region="d", projection="H12c", frame="afg", land="ivory", water="bisque4")
fig.show()
```

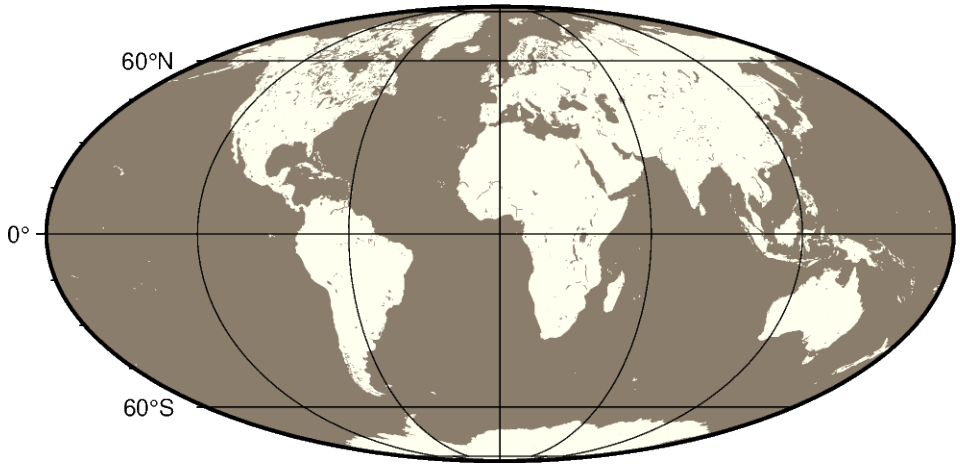
Total running time of the script: (0 minutes 0.204 seconds)

6.4.4 Mollweide projection

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*.




```
import pygmt

fig = pygmt.Figure()
```

(continues on next page)

(continued from previous page)

```
# Use region "d" to specify global region (-180/180/-90/90)
fig.coast(region="d", projection="W12c", frame="afg", land="ivory", water="bisque4")
fig.show()
```

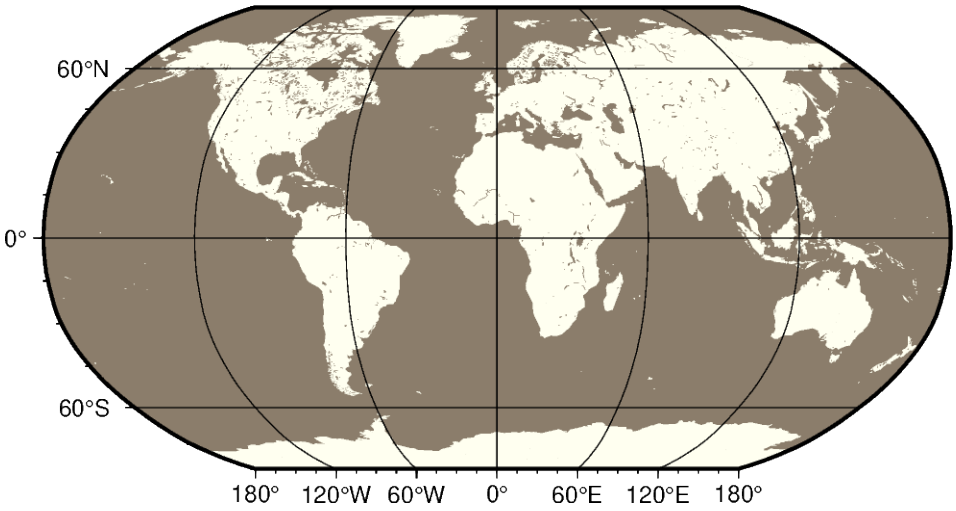
Total running time of the script: (0 minutes 0.214 seconds)

6.4.5 Robinson projection

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[1]. The scale is true along latitudes 38° north and south. 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*.




```
import pygmt

fig = pygmt.Figure()
# Use region "d" to specify global region (-180/180/-90/90)
fig.coast(region="d", projection="N12c", frame="afg", land="ivory", water="bisque4")
fig.show()
```

Total running time of the script: (0 minutes 0.203 seconds)

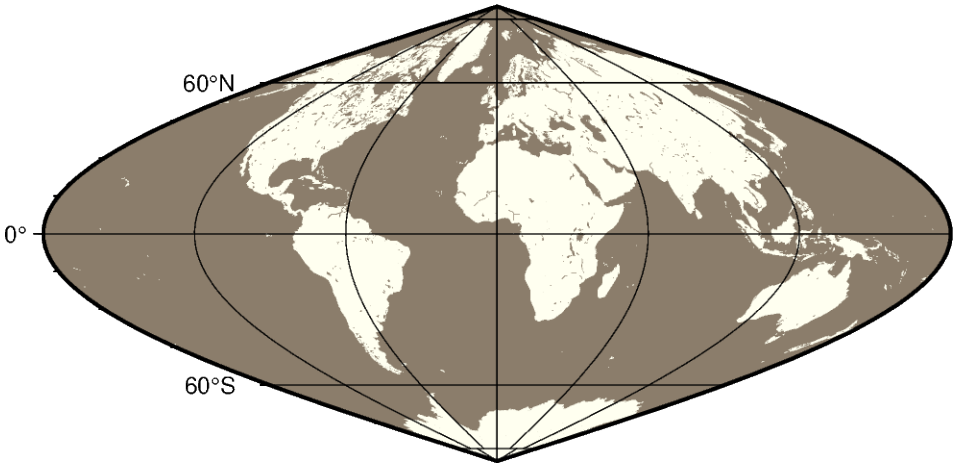
6.4.6 Sinusoidal projection

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).

[1] Robinson provided a table of y-coordinates for latitudes every 5°. To project values for intermediate latitudes one must interpolate the table. Different interpolants may result in slightly different maps. GMT uses the interpolant selected by the parameter GMT_INTERPOLANT in the gmt.conf file.

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*.




```
import pygmt

fig = pygmt.Figure()
# Use region "d" to specify global region (-180/180/-90/90)
fig.coast(region="d", projection="I12c", frame="afg", land="ivory", water="bisque4")
fig.show()
```

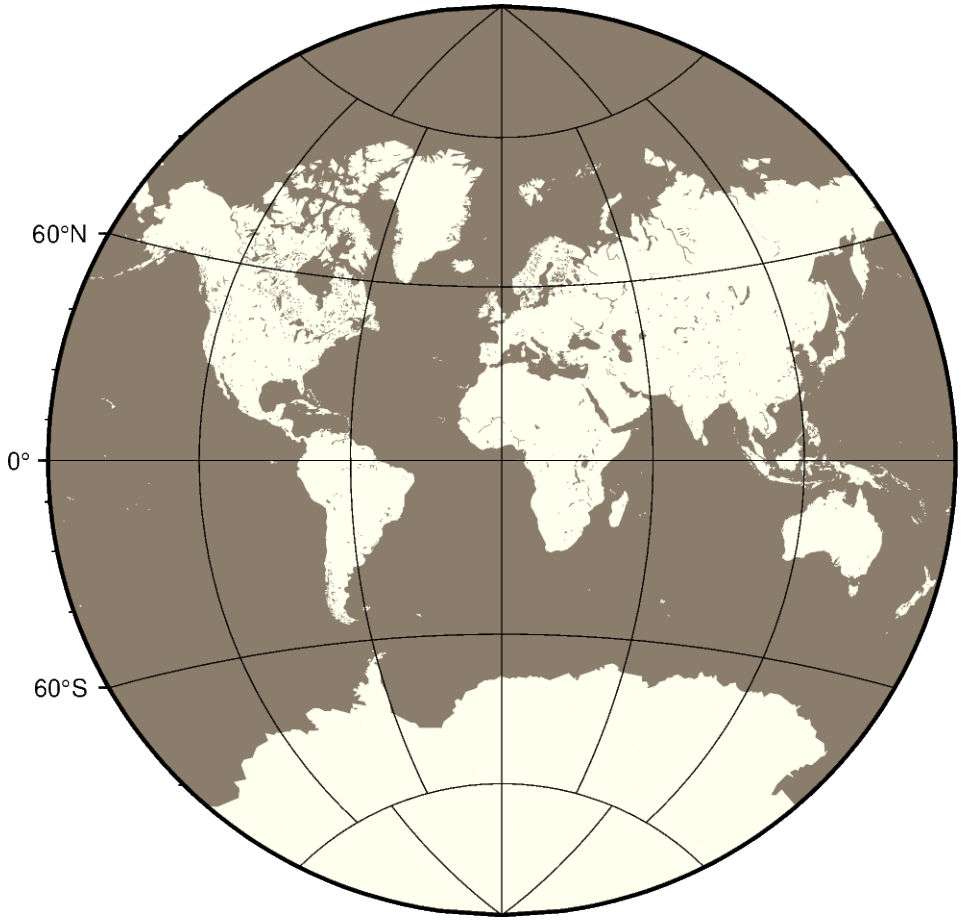
Total running time of the script: (0 minutes 0.199 seconds)

6.4.7 Van der Grinten projection

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*.




```
import pygmt

fig = pygmt.Figure()
# Use region "d" to specify global region (-180/180/-90/90)
fig.coast(region="d", projection="V12c", frame="afg", land="ivory", water="bisque4")
fig.show()
```

Total running time of the script: (0 minutes 0.238 seconds)

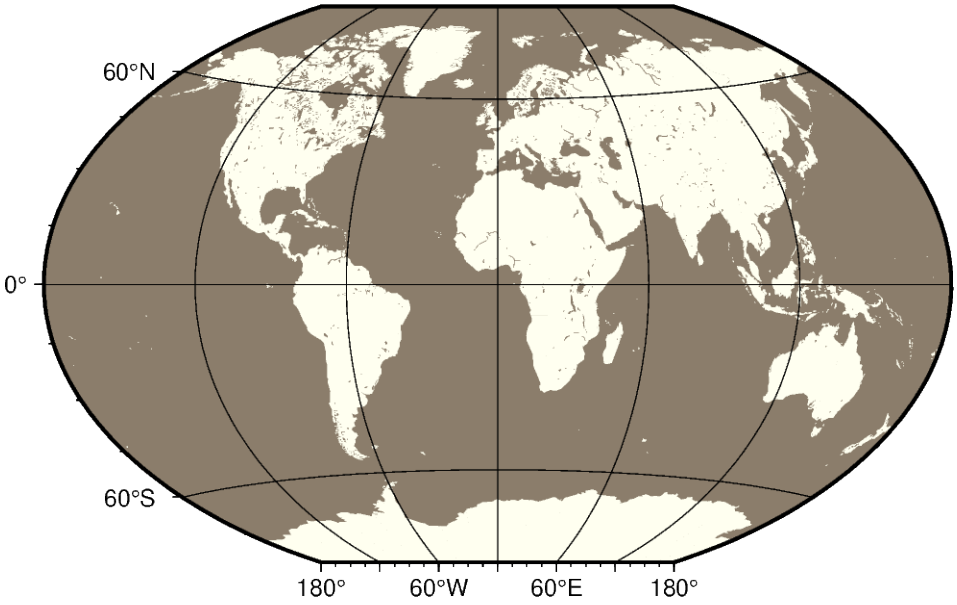
6.4.8 Winkel Tripel projection

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*.




```
import pygmt

fig = pygmt.Figure()
# Use region "d" to specify global region (-180/180/-90/90)
fig.coast(region="d", projection="R12c", frame="afg", land="ivory", water="bisque4")
fig.show()
```

Total running time of the script: (0 minutes 0.238 seconds)

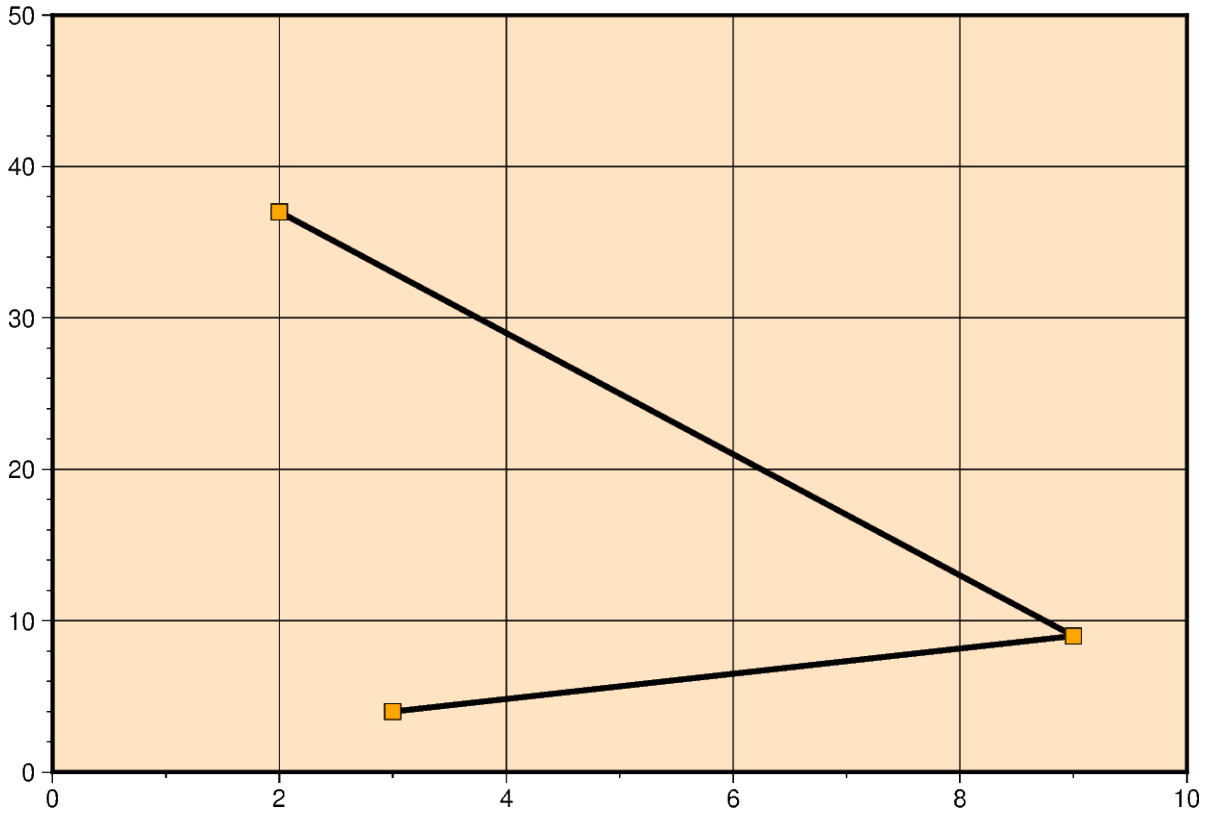
6.5 Non-geographic Projections

6.5.1 Cartesian linear

X*width*[/*height*] or **x***x-scale*[/*y-scale*]

Give the *width* of the figure and the optional *height*. The lowercase version **x** is similar to **X** but expects an *x-scale* and an optional *y-scale*.

The Cartesian linear projection is primarily designed for regular floating point data. To plot geographical data in a linear projection, see the upstream GMT documentation Geographic coordinates. To make the linear plot using calendar date/time as input coordinates, see the tutorial *Plotting datetime charts*. GMT documentation Calendar time coordinates.



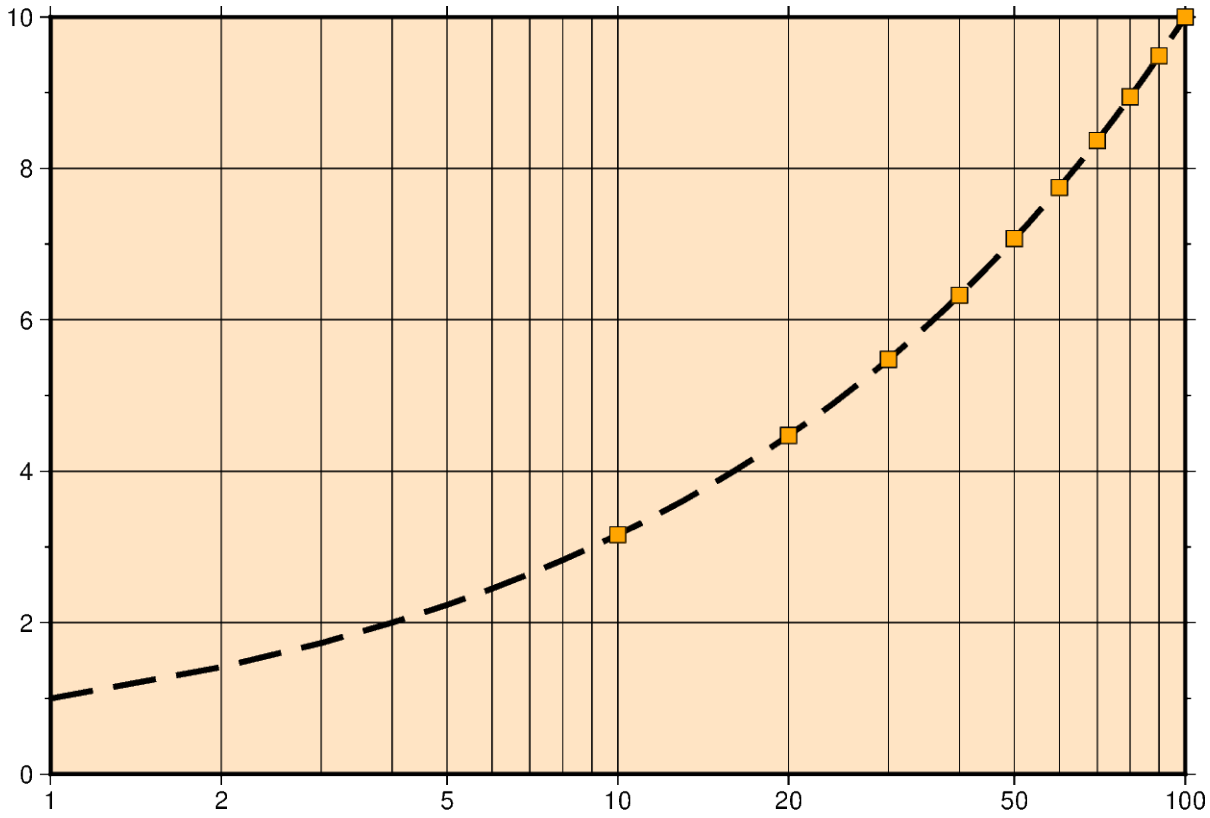

```
import pygmt

fig = pygmt.Figure()
# The region parameter is specified as x_min, x_max, y_min, y_max
fig.basemap(region=[0, 10, 0, 50], projection="X15c/10c", frame=["afg", "+gbisque"])
fig.plot(x=[3, 9, 2], y=[4, 9, 37], pen="2p,black")
# Plot data points on top of the line
# Use squares with a size of 0.3 centimeters, an "orange" fill and a "black" outline
fig.plot(x=[3, 9, 2], y=[4, 9, 37], style="s0.3c", fill="orange", pen="black")
fig.show()
```

Total running time of the script: (0 minutes 0.147 seconds)

6.5.2 Cartesian logarithmic

X*width*[**l**][/*height*[**l**]] or **x***x-scale*[**l**][/*y-scale*[**l**]]

Give the *width* of the figure and the optional *height*. The lowercase version **x** is similar to **X** but expects an *x-scale* and an optional *y-scale*. Each axis with a logarithmic transformation requires **l** after its size argument.




```
import numpy as np
import pygmt

# 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.basemap(
    region=[1, 100, 0, 10],
    # Set a logarithmic transformation on the x-axis
    projection="X15cl/10c",
    # Set the figures frame and color as well as
    # annotations, ticks, and gridlines
    frame=["WSne+gbisque", "xa2g3", "ya2f1g2"],
)

# Set the line thickness to "2p", the color to "black", and the style to "dashed"
fig.plot(x=xline, y=yline, pen="2p,black,dashed")

# Plot the square root values on top of the line
# Use squares with a size of 0.3 centimeters, an "orange" fill and a "black" outline
```

(continues on next page)

(continued from previous page)

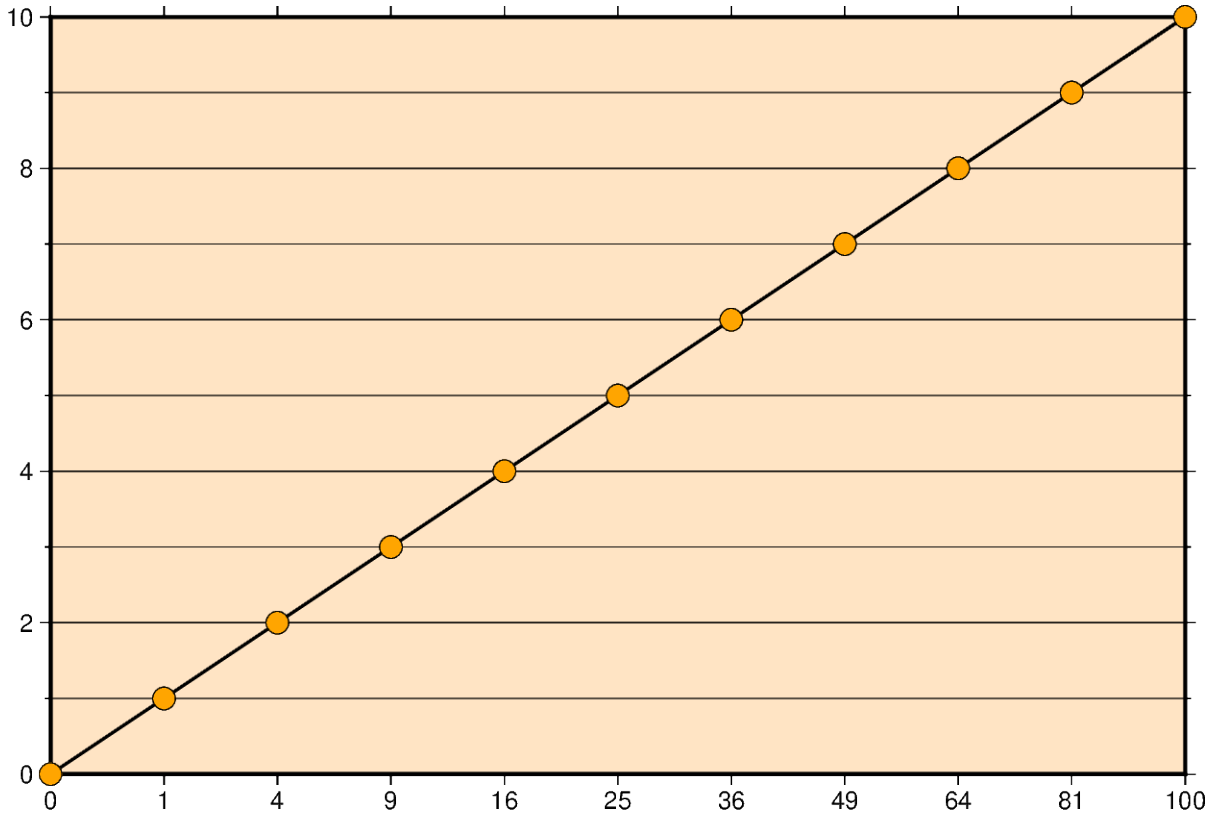
```
# Symbols are not clipped if they go off the figure
fig.plot(x=xpoints, y=ypoints, style="s0.3c", fill="orange", pen="black", no_
↪clip=True)
fig.show()
```

Total running time of the script: (0 minutes 0.157 seconds)

6.5.3 Cartesian power

X*width*[**p***pvalue*][/*height*[**p***pvalue*]] or **x***x-scale*[**p***pvalue*][/*y-scale*[**p***pvalue*]]

Give the *width* of the figure and the optional argument *height*. The lowercase version **x** is similar to **X** but expects an *x-scale* and an optional *y-scale*. Each axis with a power transformation requires **p** and the exponent for that axis after its size argument.




```
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.basemap(
    region=[0, 100, 0, 10],
    # Set the power transformation of the x-axis, with a power of 0.5
```

(continues on next page)

(continued from previous page)

```
    projection="X15cp0.5/10c",
    # Set the figures frame as well as annotations and ticks
    # The "p" forces to show only square numbers as annotations of the x-axis
    frame=["WSne+gbisque", "xfga1p", "ya2f1g"],
)

# Set the line thickness to "thick" (equals "1p", i.e. 1 point)
# Use as color "black" (default) and as style "solid" (default)
fig.plot(x=xvalues, y=yvalues, pen="thick,black,solid")

# Plot the data points on top of the line
# Use circles with 0.3 centimeters diameter, with an "orange" fill and a "black"␣
↪outline
# Symbols are not clipped if they go off the figure
fig.plot(x=xvalues, y=yvalues, style="c0.3c", fill="orange", pen="black", no_
↪clip=True)
fig.show()
```

Total running time of the script: (0 minutes 0.150 seconds)

6.5.4 Polar

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

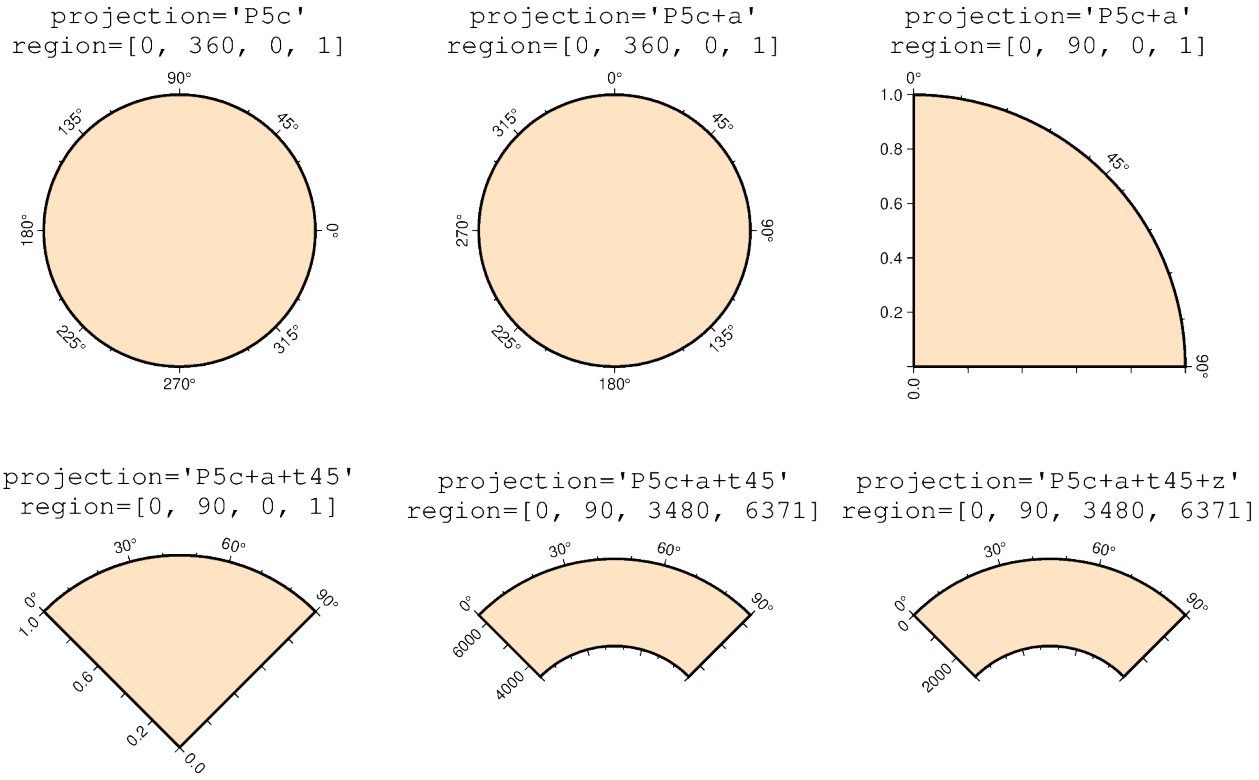
The full syntax for polar projections is:

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 lowercase 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 are 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.




```
import pygmt

fig = pygmt.Figure()

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

# ============
# Top left
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 title; @^ allows for a line break within the title
    frame=["xa45f", "+gbisque+tprojection='P5c' @^ region=[0, 360, 0, 1]"],
)

fig.shift_origin(xshift="w+3c")

# ============
# Top middle
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 title; @^ allows for a line break within the title
```

(continues on next page)

(continued from previous page)

```
    frame=["xa45f", "+gbisque+tprojection='P5c+a' @^ region=[0, 360, 0, 1]"],
)

fig.shift_origin(xshift="w+3c")

# ============
# Top right
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 title; @^ allows for a line break within the title
    frame=["xa45f", "ya0.2", "WNe+gbisque+tprojection='P5c+a' @^ region=[0, 90, 0, 1]
↪"],
)

fig.shift_origin(xshift="-2w-6c", yshift="-h-2c")

# ============
# Bottom left
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 title; @^ allows for a line break within the title
    frame=[
        "xa30f",
        "ya0.2",
        "WNe+gbisque+tprojection='P5c+a+t45' @^ region=[0, 90, 0, 1]",
    ],
)

fig.shift_origin(xshift="w+3c", yshift="1.3c")

# ============
# Bottom middle
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 title; @^ allows for a line break within the title
    frame=[
        "xa30f",
        "ya",
        "WNse+gbisque+tprojection='P5c+a+t45' @^ region=[0, 90, 3480, 6371]",
```

(continues on next page)

(continued from previous page)

```
    ],
)

fig.shift_origin(xshift="w+3c")

# ============
# Bottom right
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 title; @^ allows for a line break within the title
    frame=[
        "xa30f",
        "ya",
        "WNse+gbisque+tprojection='P5c+a+t45+\\z' @^ region=[0, 90, 3480, 6371]",
    ],
)

fig.show()
```

Total running time of the script: (0 minutes 0.283 seconds)

CHAPTER

SEVEN

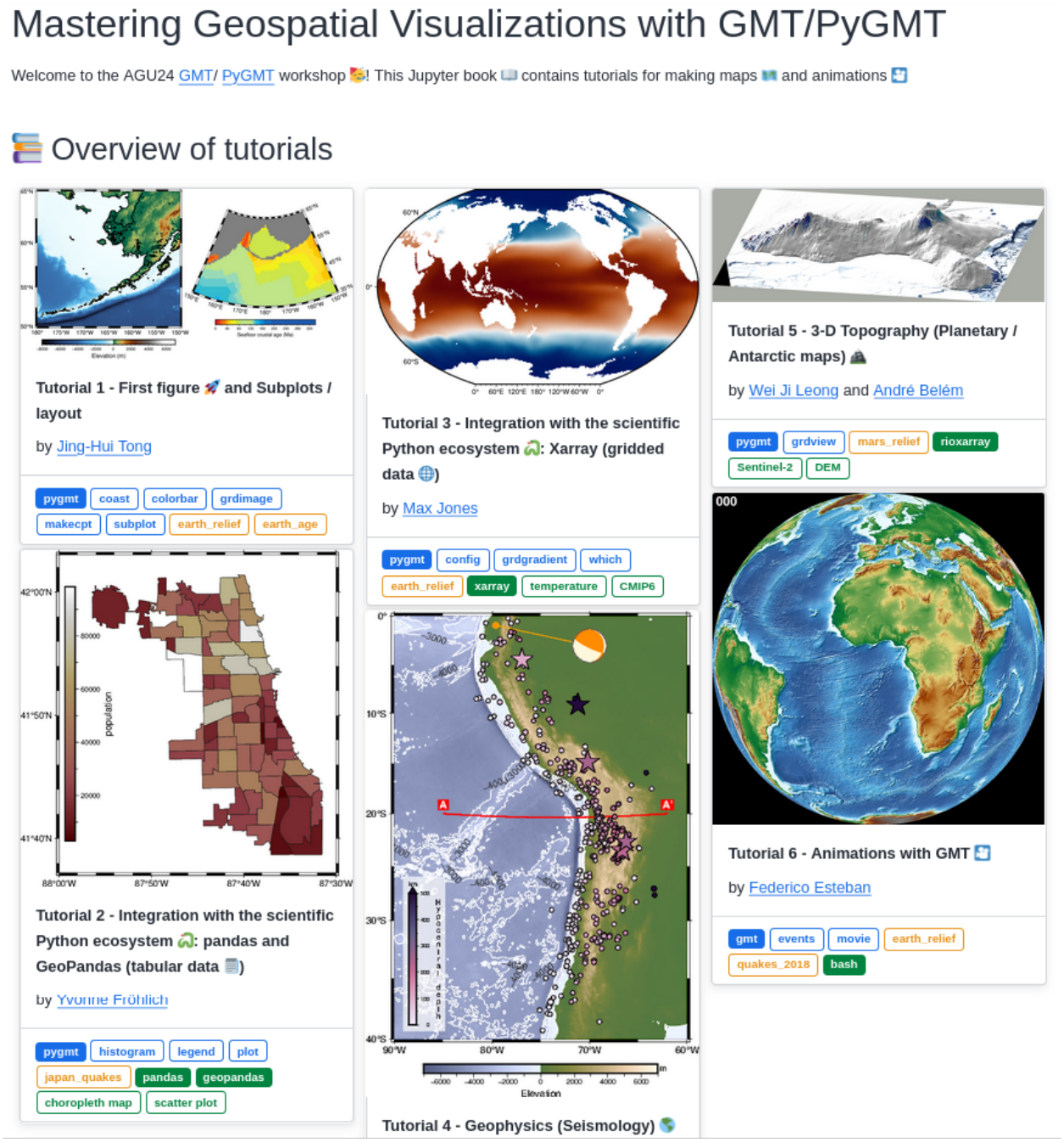
EXTERNAL RESOURCES

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.

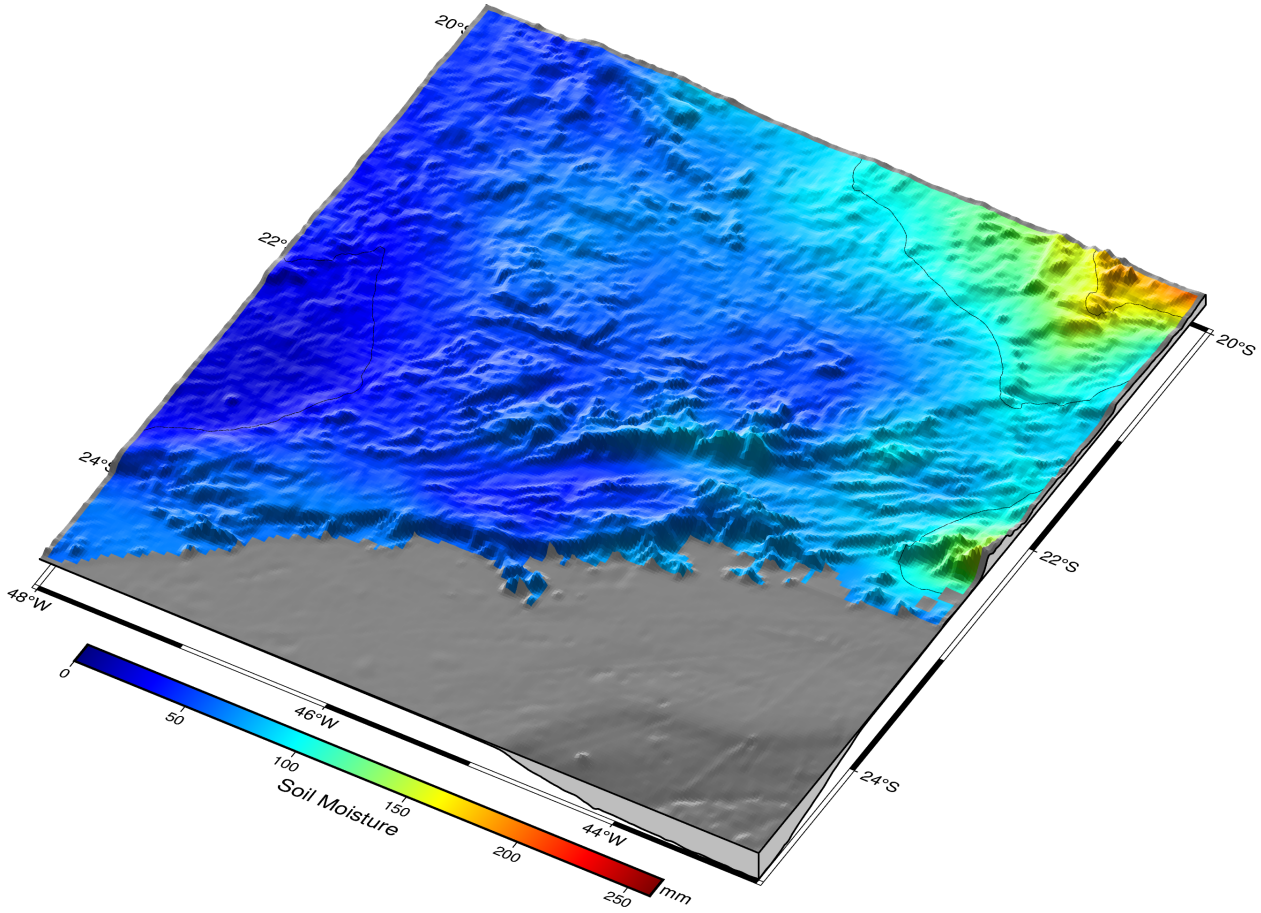
7.1 Tutorials

2024 AGU PREWS9: Mastering Geospatial Visualizations with GMT/PyGMT



Wei Ji Leong, Yvonne Fröhlich, Jing-Hui Tong, Federico Esteban, Max Jones, Andre Belem

https://www.generic-mapping-tools.org/agu24workshop/ 2024 PyGMT Webinar using Google Colab (in Por-



tuguese)

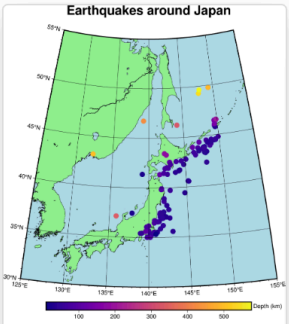
Andre Belem

https://github.com/andrebelem/Oficina_PyGMT 2022 EGU SC5.2: Crafting beautiful maps with PyGMT

🌍 Crafting beautiful maps with PyGMT

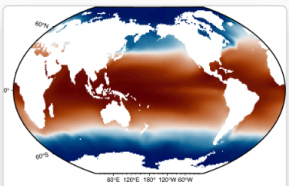
Welcome to the EGU22 PyGMT short course 🥳

This Jupyter book 📖 contains PyGMT tutorials for producing maps 🗺️ and doing geospatial data processing 🌐



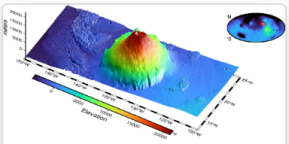
Anatomy of a PyGMT figure

by Leonardo Uieda



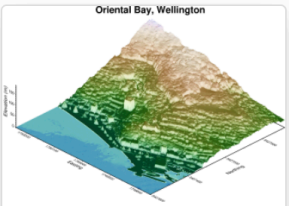
Integration with the scientific Python ecosystem 🐍

by Max Jones



Making some Mars maps with PyGMT

by André Belém



LiDAR Point clouds to 3D surfaces ✨➡️🏔️

by Wei Ji Leong

Wei Ji Leong, Leonardo Uieda, Max Jones, Andre Belem

https://www.generic-mapping-tools.org/egu22pygmt/ 2021 PyGMT course at the UAF Geophysical Institute

2021 GI PyGMT short course (lecture notebook)

Liam Toney

Geophysical Institute,
University of Alaska Fairbanks

1 March 2021

---

1. Basics

First, let's import the PyGMT Python package:

[1]:
```
import pygmt
```

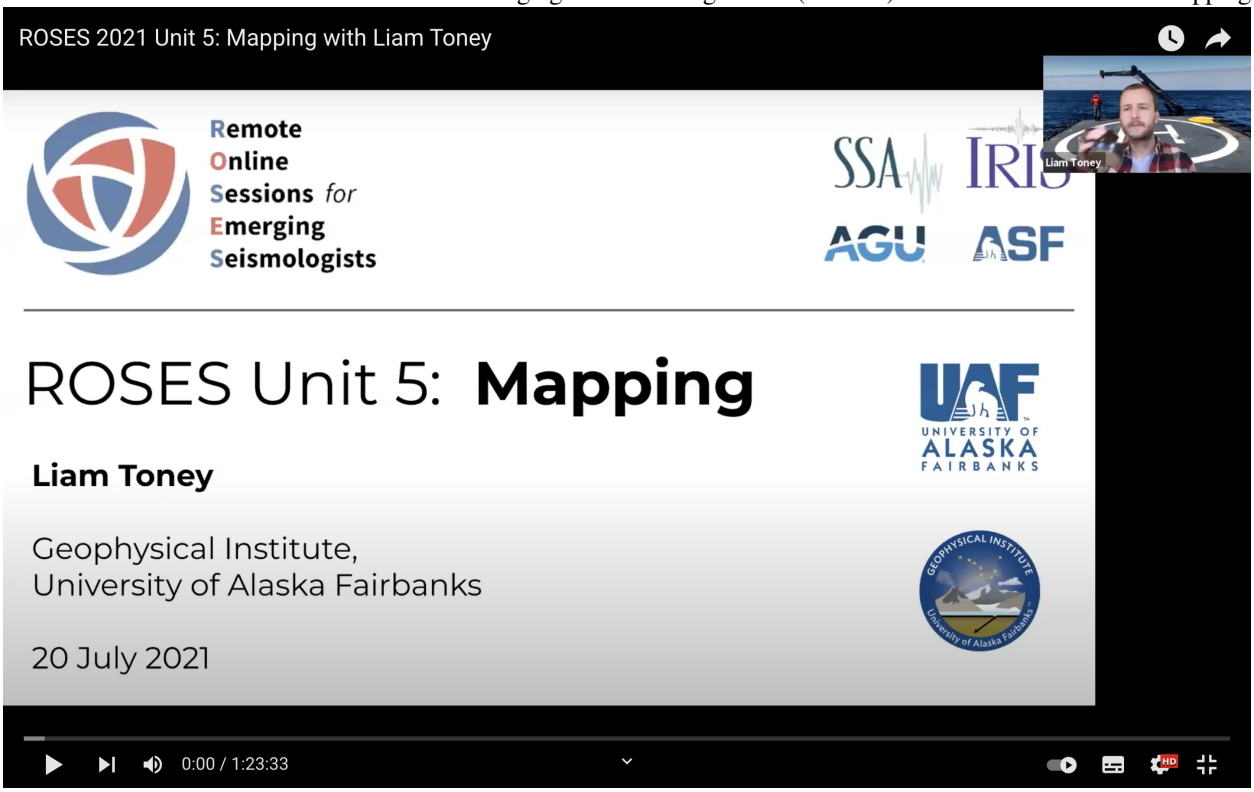
In PyGMT, every plot or map must start with the creation of a `pygmt.Figure` instance. This is similar to the Matplotlib command

```
fig = plt.figure()
```

which creates a `matplotlib.figure.Figure` instance. (Throughout this lecture, I'll try to connect PyGMT commands to Matplotlib commands.)

Liam Toney

https://github.com/liamtoney/gi-pygmt-2021 2021
Remote Online Sessions for Emerging Seismologists (ROSES): Unit 5 - Mapping



Liam Toney

https://www.youtube.com/watch?v=Zvcy7VDuhiw 2020
Remote Online Sessions for Emerging Seismologists (ROSES): Unit 8 - PyGMT



Liam Toney

https://www.iris.edu/hq/inclass/lesson/728 PyGMT Tutorial in 2021

PyGMT Tutorial in 2021

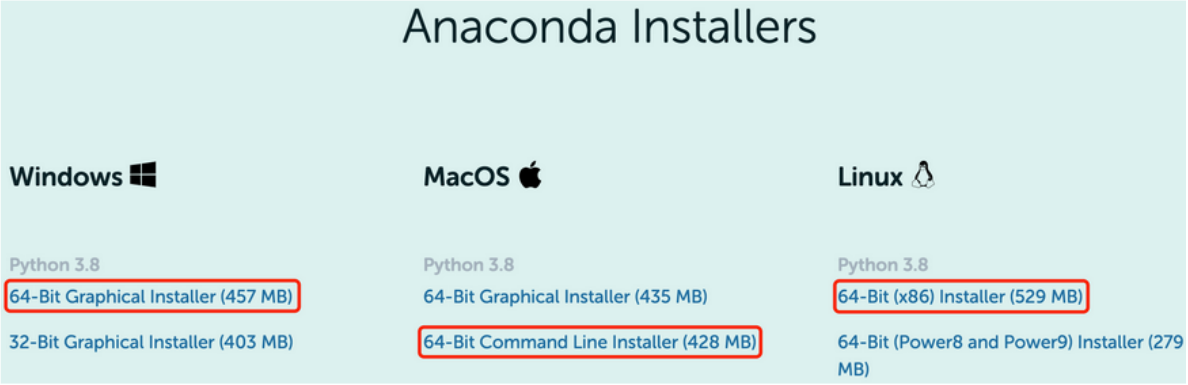
A preliminary introduction to PyGMT for MIG members.

Installation instructions

Anaconda is a Python distribution for scientific computing and is **strongly recommended**.

1. Download Anaconda

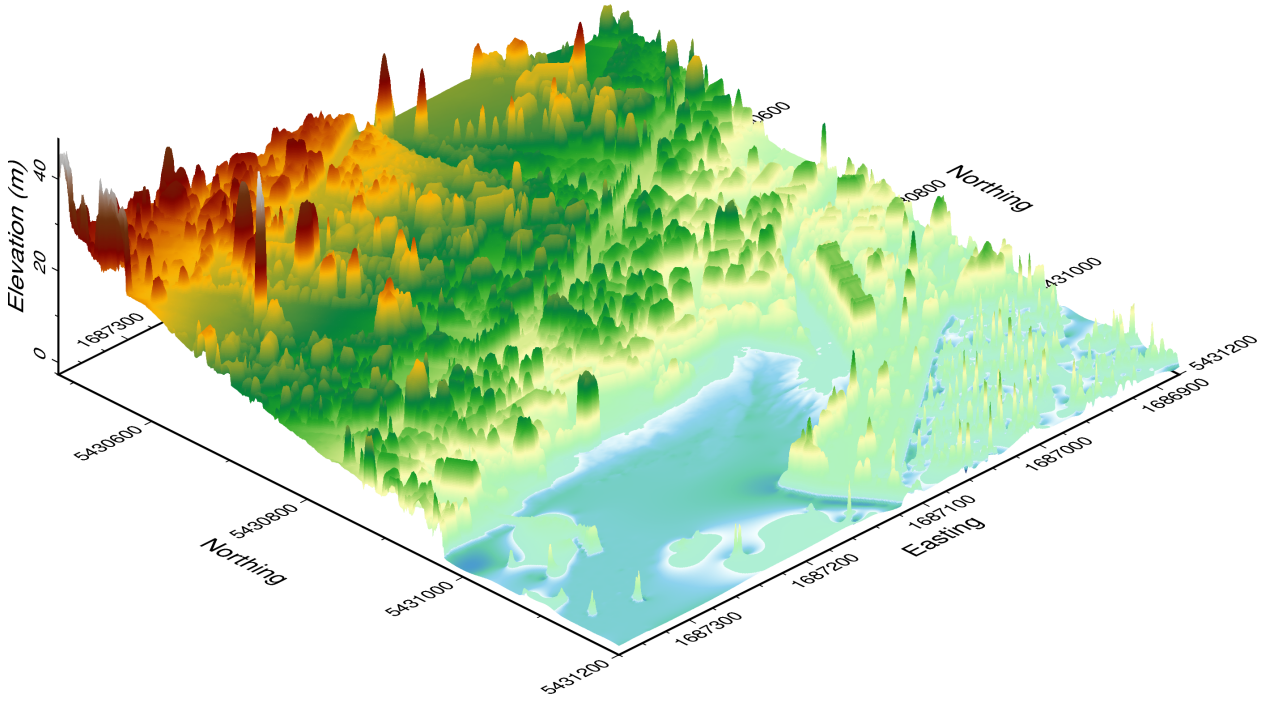
   Go to Anaconda download, you can see a downloading page similar to the below snapshot. Choose the corresponding package based on your operating system. Those indicated in red boxes are usually used.



MIGG-NTU

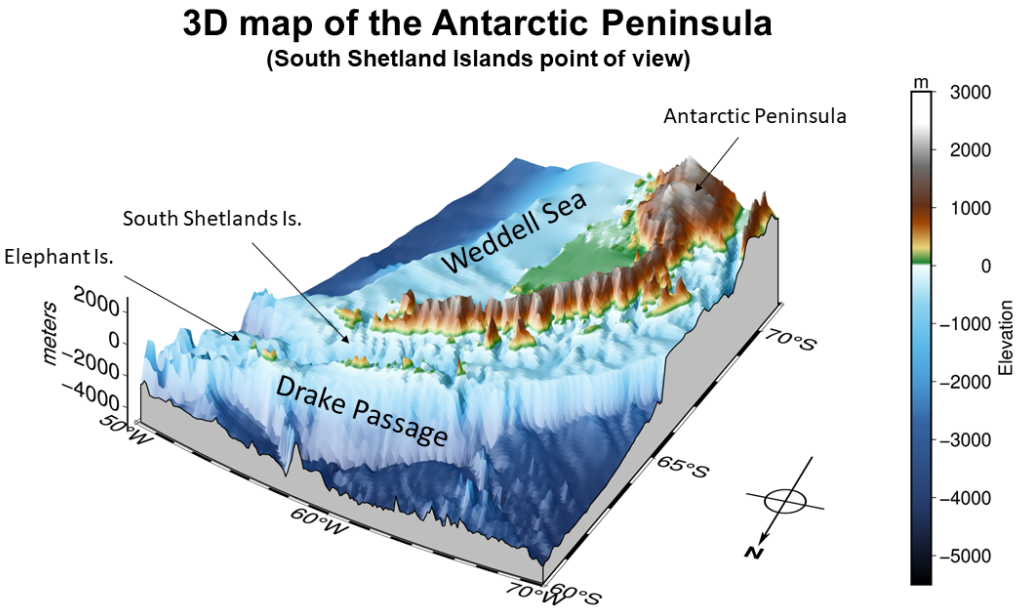
https://github.com/MIGG-NTU/PyGMT2021 PyGMT Workshop at FOSS4G Oceania 2019

Picton Harbour



Wei Ji Leong

https://github.com/GenericMappingTools/foss4g2019oceania Crafting 3D maps of Antarctica with PyGMT

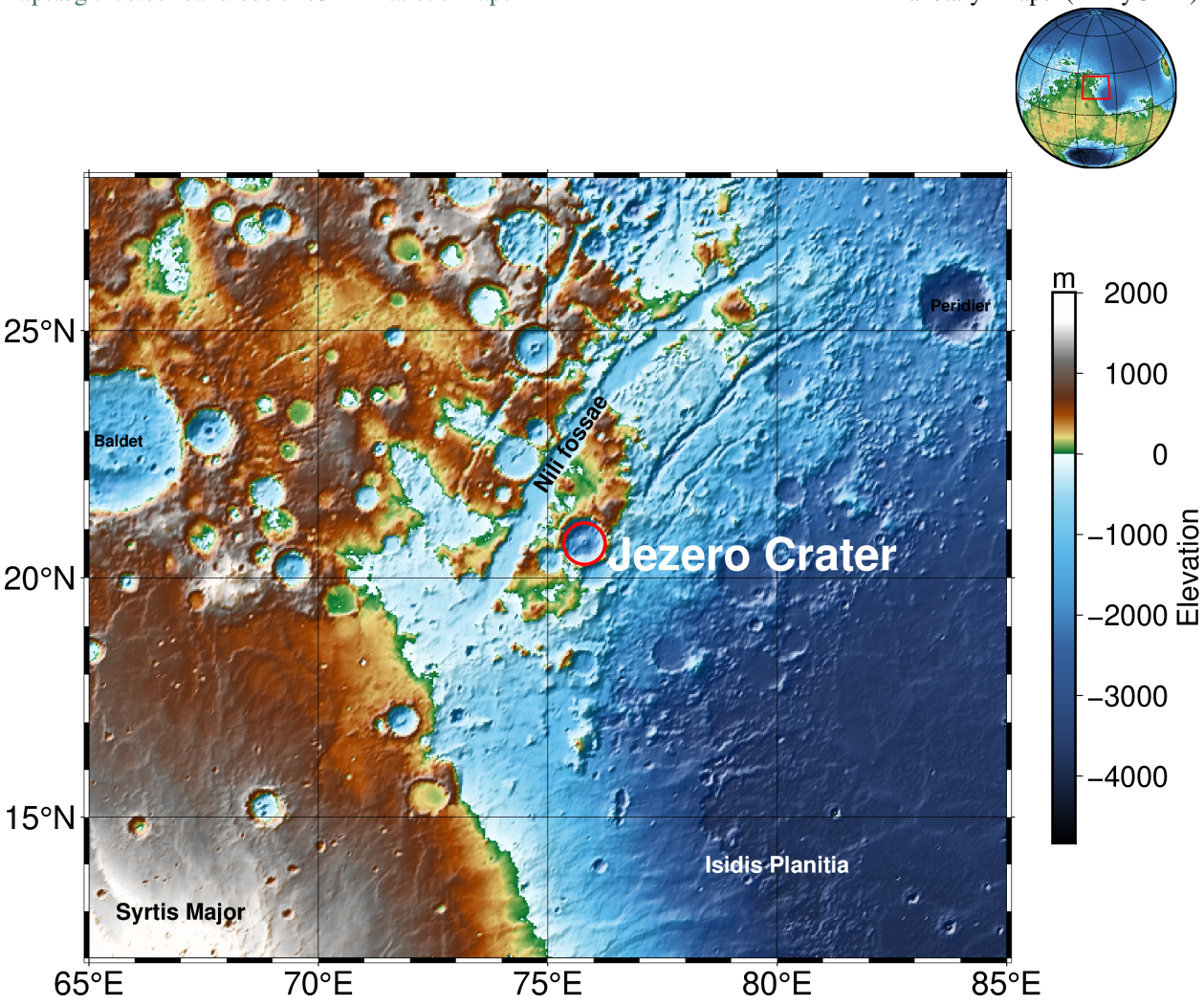


and the IBCSO V2

Andre Belem

https://github.com/andrebelem/3D-Antarctic-maps

Planetary Maps (in PyGMT)

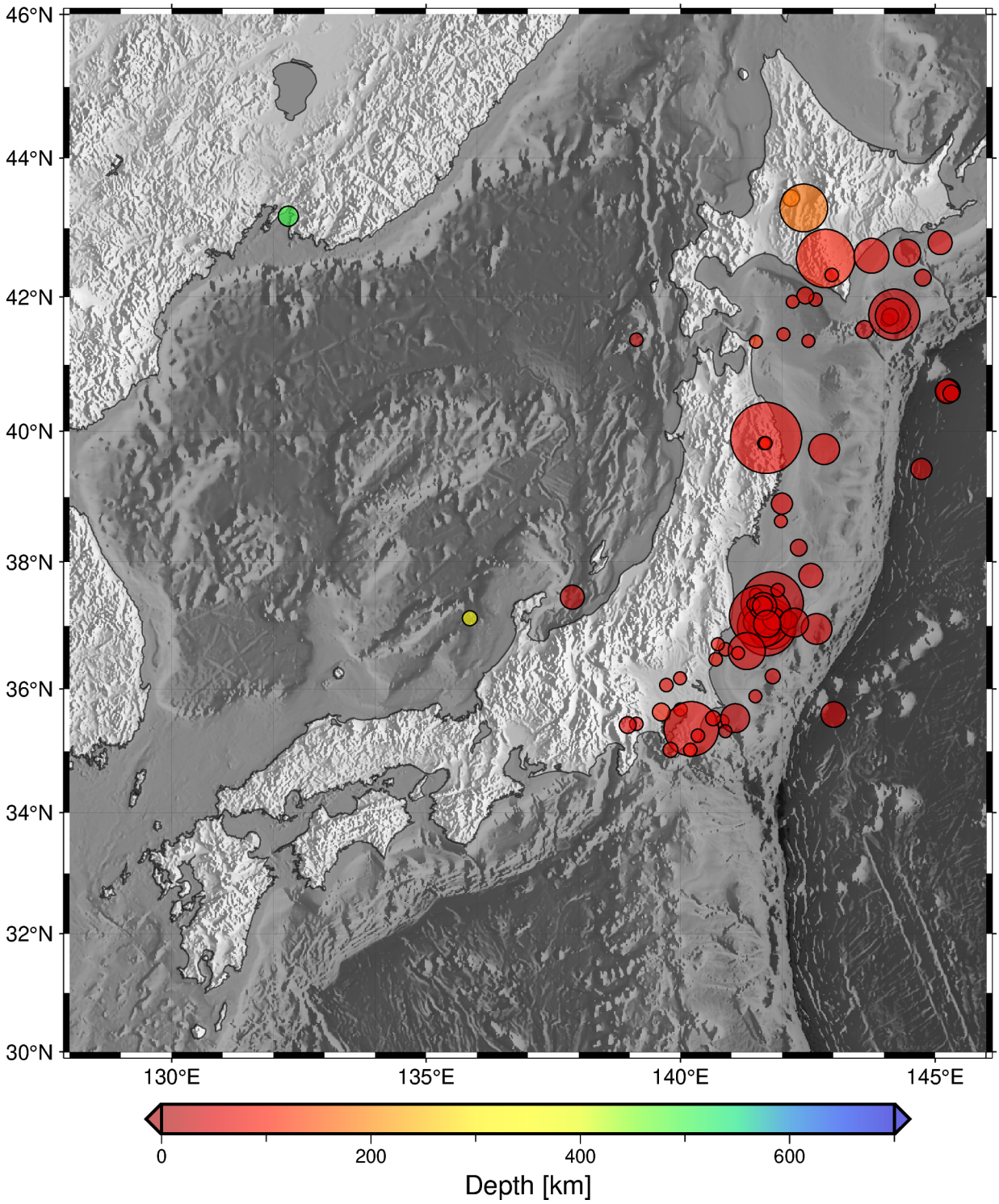


Andre Belem

https://github.com/andrebelem/PlanetaryMaps

PyGMT-HOWTO

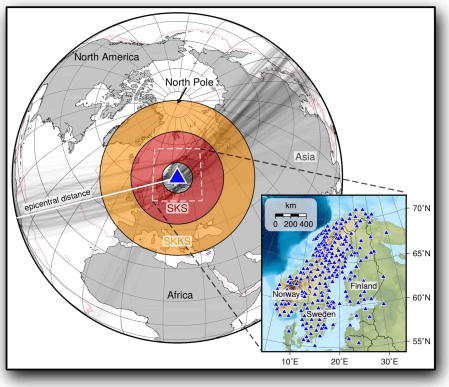
Seismicity around Japanese Archipelago

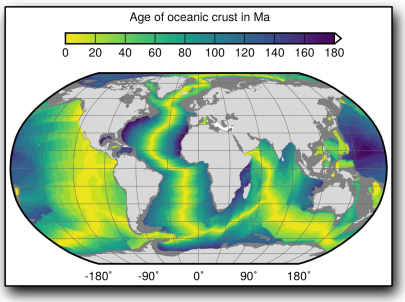


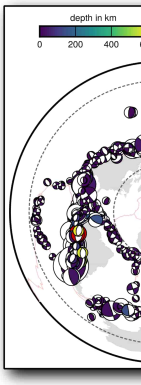
Takuto Maeda

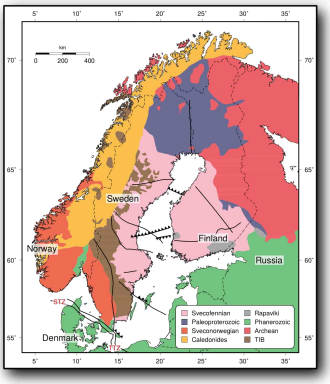
https://tktmyd.github.io/pygmt-howto-jp/pygmt

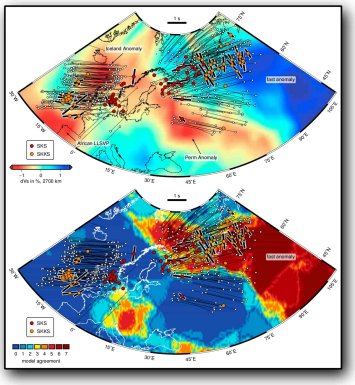
7.2 Examples from Publications and Posters

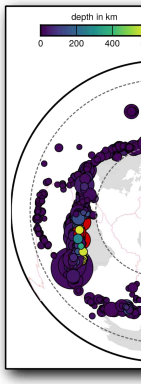










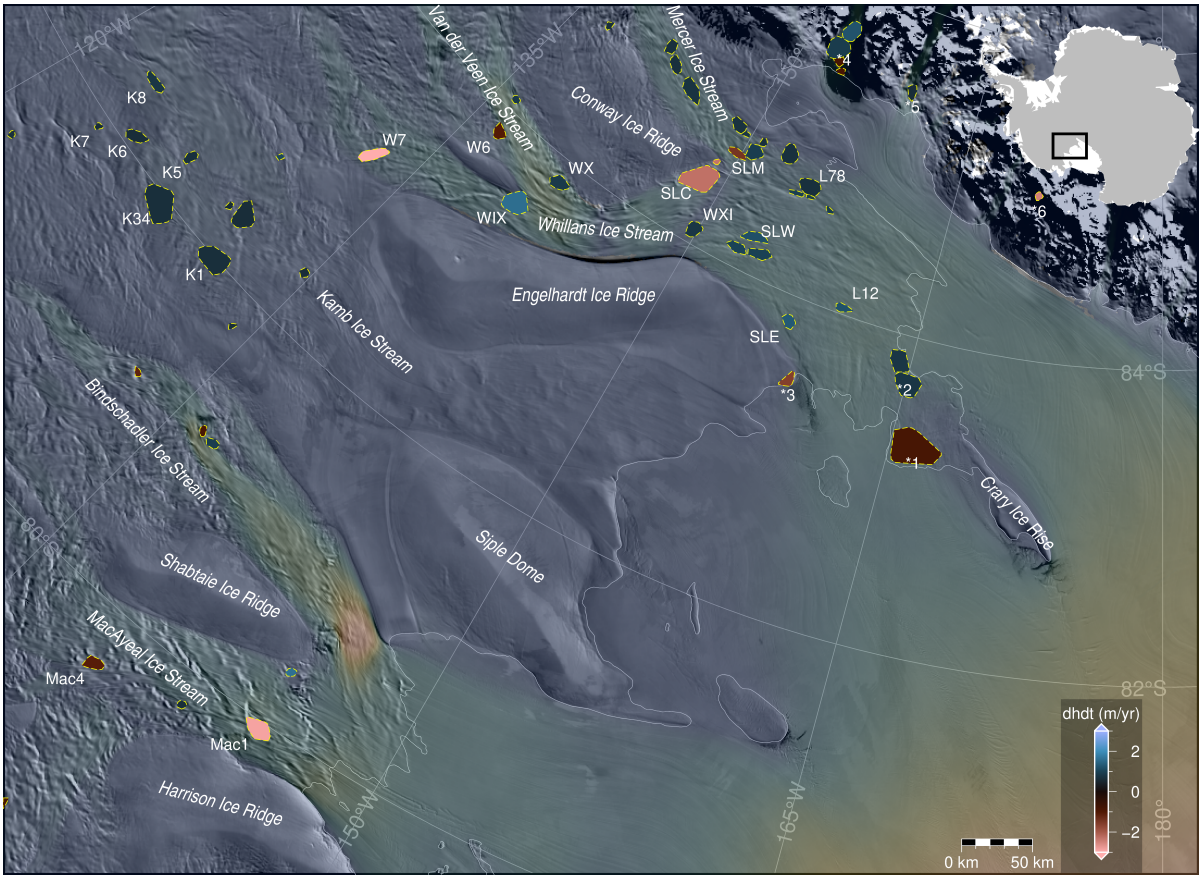


GMT and PyGMT plotting examples

Michael Grund

https://github.com/michaelgrund/GMT-plotting

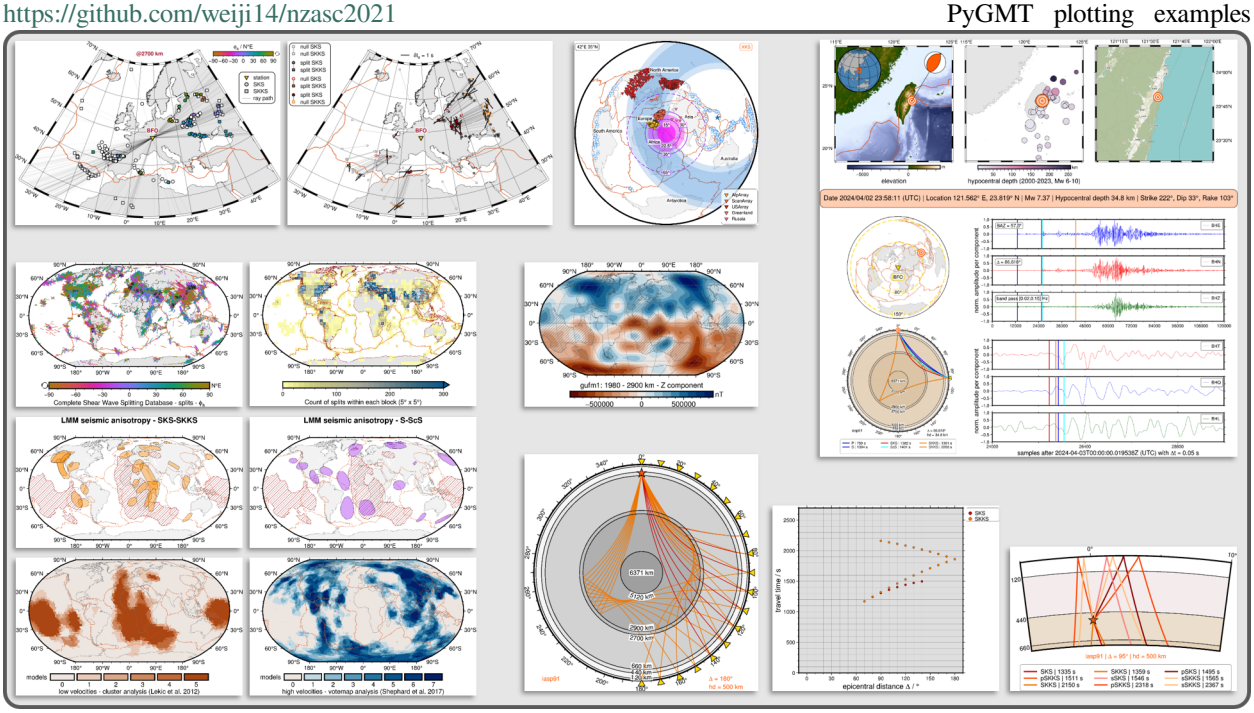
NZ Antarctic Science Conference 2021 poster



Wei Ji Leong

https://github.com/weiji14/nzasc2021

PyGMT plotting examples



Yvonne Fröhlich

https://github.com/yvonnefroehlich/gmt-pygmt-plotting

CHAPTER

EIGHT

API REFERENCE

This page gives an overview of all public PyGMT objects, functions and methods.

8.1 Plotting

8.1.1 Figure class overview

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*.

Examples

```
>>> import pygmt
>>> fig = pygmt.Figure()
>>> fig.basemap(region=[0, 360, -90, 90], projection="W15c", frame=True)
>>> fig.savefig("my-figure.png")
>>> # Make sure the figure file is generated and clean it up
>>> from pathlib import Path
>>> assert Path("my-figure.png").exists()
>>> Path("my-figure.png").unlink()
```

The plot region can be specified through ISO country codes (for example, `"JP"` for Japan):

```
>>> import pygmt
>>> fig = pygmt.Figure()
>>> fig.basemap(region="JP", projection="M7c", frame=True)
>>> # 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
```

Attributes

property Figure.**region**: **ndarray**

The geographic WESN bounding box for the current figure.

Methods Summary

Figure.basemap(*[, region, projection, ...])	Plot base maps and frames.
Figure.coast(*[, area_thresh, frame, lakes, ...])	Plot continents, countries, shorelines, rivers, and borders.
Figure.colorbar(*[, frame, cmap, position, ...])	Plot gray scale or color scale bar.
Figure.contour([data, x, y, z])	Contour table data by direct triangulation.
Figure.grdcontour(grid, **kwargs)	Make contour map using a grid.
Figure.grdimage(grid, **kwargs)	Project and plot grids or images.
Figure.grdview(grid, **kwargs)	Create 3-D perspective image or surface mesh from a grid.
Figure.histogram(data, **kwargs)	Calculate and plot histograms.
Figure.hlines(y[, xmin, xmax, pen, label, ...])	Plot one or multiple horizontal line(s).
Figure.image(imagefile, **kwargs)	Plot raster or EPS images.
Figure.inset(*[, position, box, projection, ...])	Manage figure inset setup and completion.
Figure.legend([spec, position, box])	Plot a legend.
Figure.logo(*[, region, projection, ...])	Plot the GMT logo.
Figure.meca(spec, scale[, convention, ...])	Plot focal mechanisms.
Figure.plot([data, x, y, size, symbol, ...])	Plot lines, polygons, and symbols in 2-D.
Figure.plot3d([data, x, y, z, size, symbol, ...])	Plot lines, polygons, and symbols in 3-D.
Figure.psconvert(*[, crop, gs_option, dpi, ...])	Convert [E]PS file(s) to other formats using Ghostscript.
Figure.rose([data, length, azimuth])	Plot a polar histogram (rose, sector, windrose diagrams).
Figure.savefig(fname[, transparent, crop, ...])	Save the figure to an image file.
Figure.set_panel([panel, fixedlabel, ...])	Set the current subplot panel to plot on.
Figure.shift_origin([xshift, yshift])	Shift the plot origin in x and/or y directions.
Figure.show([method, dpi, width, waiting])	Display a preview of the figure.
Figure.solar([terminator, terminator_datetime])	Plot day-night terminators and other sunlight parameters.
Figure.subplot([nrows, ncols, figsize, ...])	Manage figure subplot configuration and selection.
Figure.ternary(data[, alabel, blabel, clabel])	Plot data on ternary diagrams.
Figure.text([textfiles, x, y, position, ...])	Plot or typeset text.
Figure.tilemap(region[, zoom, source, ...])	Plot an XYZ tile map.
Figure.timestamp([text, label, justify, ...])	Plot the GMT timestamp logo.
Figure.velo([data])	Plot velocity vectors, crosses, anisotropy bars, and wedges.
Figure.vlines(x[, ymin, ymax, pen, label, ...])	Plot one or multiple vertical line(s).
Figure.wiggle([data, x, y, z, fillpositive, ...])	Plot z=f(x,y) anomalies along tracks.

pygmt.Figure.basemap

Figure.**basemap**(** (Keyword-only parameters separator (PEP 3102)), region=None, projection=None, zscale=None, zsize=None, frame=None, map_scale=None, box=None, rose=None, compass=None, verbose=None, panel=None, coltypes=None, perspective=None, transparency=None, **kwargs*)

Plot base maps and frames.

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 if not in subplot mode.

Full option list at https://docs.generic-mapping-tools.org/6.5/basemap.html

Aliases:

- B = frame
- F = box
- J = projection
- JZ = zsize
- Jz = zscale
- L = map_scale
- R = region
- Td = rose
- Tm = compass
- V = verbose
- c = panel
- f = coltypes
- p = perspective
- t = transparency

Parameters

- **projection** (*str*) – *projcode*[*projparams*/]*width*|*scale*. 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, str, or list*) – Set map boundary *frame and axes attributes*.
- **map_scale** (*str*) – [**g**|**j**|**J**|**n**|**x**]*refpoint***+w***length*. Draw a simple map scale centered on the reference point specified.
- **box** (*bool or str*) – [**+c***clearances*][**+g***fill*][**+i**[[*gap*/]*pen*]][**+p**[*pen*]][**+r**[*radius*]][**+s**[[*dx*/*dy*/][*shade*]]]. If set to `True`, draw a rectangular border around the map scale or rose. 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 [Default is `"4p/-4p"`] and shade sets the fill style to use for shading [Default is `"gray50"`].
- **rose** (*str*) – Draw a map directional rose on the map at the location defined by the reference and anchor points.
- **compass** (*str*) – Draw a map magnetic rose on the map at the location defined by the reference and anchor points.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **panel** (*bool, 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/6.5/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/6.5/gmt.html#perspective-full.
- **transparency** (*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`

pygmt.Figure.coast

`Figure.`**`coast`**(*, *area_thresh=None*, *frame=None*, *lakes=None*, *resolution=None*, *dcw=None*, *box=None*, *land=None*, *rivers=None*, *projection=None*, *map_scale=None*, *borders=None*, *region=None*, *water=None*, *verbose=None*, *shorelines=None*, *panel=None*, *perspective=None*, *transparency=None*, ***kwargs*)

Plot continents, countries, shorelines, rivers, and borders.

Plots grayshaded, colored, or textured land masses [or water masses] on maps and [optionally] draws coastlines, rivers, and political boundaries. 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/6.5/coast.html

Aliases:

- A = area_thresh
- B = frame
- C = lakes
- D = resolution
- E = dcw
- F = box
- G = land
- I = rivers
- J = projection
- L = map_scale
- N = borders
- R = region
- S = water
- V = verbose
- W = shorelines
- c = panel
- p = perspective
- t = transparency

Parameters

- **`projection`** (*`str`*) – *projcode*[*projparams*/]*width*|*scale*. 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`** (*`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, 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**. Select the resolution of the data set to: (**f**)ull, (**h**)igh, (**i**)ntermediate, (**l**)ow, and (**c**)rude.
- **`land`** (*`str`*) – Select filling of "dry" areas.
- **`rivers`** (*`int, str, or list`*) – *river*[/*pen*]. Draw rivers. Specify the type of rivers and [optionally] append pen attributes [Default is `"0.25p,black,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"`: rivers and canals (`0 - 10`)
  - `"A"`: rivers and canals except river-lakes (`1 - 10`)
  - `"r"`: permanent rivers (`0 - 4`)
  - `"R"`: permanent rivers except river-lakes (`1 - 4`)
  - `"i"`: intermittent rivers (`5 - 7`)
  - `"c"`: canals (`8 - 10`)

- **map_scale** (*str*) – [**g**|**j**|**J**|**n**|**x**]*refpoint*+**w***length*. Draw a simple map scale centered on the reference point specified.
- **box** (*bool or str*) – [**+c***clearances*][**+g***fill*][**+i**[[*gap*/]*pen*]][**+p**[*pen*]][**+r**[*radius*]][**+s**[[*dx*/*dy*/][*shade*]]]. If set to `True`, draw a rectangular border around the map scale or rose. 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 [Default is `"4p/-4p"`] and shade sets the fill style to use for shading [Default is `"gray50"`].
- **borders** (*int, str, or list*) – *border*[/*pen*]. Draw political boundaries. Specify the type of boundary and [optionally] append pen attributes [Default is `"0.25p,black,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 "wet" areas.
- **shorelines** (*int, str, or list*) – [*level*/]*pen*. Draw shorelines [Default is no shorelines]. Append pen attributes [Default is `"0.25p,black,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,...*[**+g***fill*][**+p***pen*][**+z**]. Select painting 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].
- **panel** (*bool, 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/6.5/gmt.html#perspective-full.
- **transparency** (*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 [*Full usage*].

Example

```
>>> import pygmt
>>> # Create a new plot with pygmt.Figure()
>>> fig = pygmt.Figure()
>>> # Call the coast method for the plot
>>> fig.coast(
...     # Set the projection to Mercator, and the plot width to 10 centimeters
...     projection="M10c",
...     # Set the region of the plot
...     region=[-10, 30, 30, 60],
...     # Set the frame of the plot, here annotations and major ticks
...     frame="a",
...     # Set the color of the land to "darkgreen"
...     land="darkgreen",
...     # Set the color of the water to "lightblue"
...     water="lightblue",
...     # Draw national borders with a 1-point black line
...     borders="1/1p,black",
... )
>>> # Show the plot
>>> fig.show()
```

Examples using `pygmt.Figure.coast`

pygmt.Figure.colorbar

Figure.**colorbar**(*, *frame=None*, *cmap=None*, *position=None*, *box=None*, *truncate=None*, *shading=None*, *projection=None*, *equalsize=None*, *log=None*, *region=None*, *verbose=None*, *scale=None*, *zfile=None*, *panel=None*, *perspective=None*, *transparency=None*, ***kwargs*)

Plot gray scale or color scale bar.

Both horizontal and vertical colorbars 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/6.5/colorbar.html

Aliases:

- B = frame
- C = cmap
- D = position
- F = box
- G = truncate
- I = shading
- J = projection
- L = equalsize
- Q = log
- R = region
- V = verbose
- W = scale
- Z = zfile
- c = panel
- p = perspective
- t = transparency

Parameters

- **frame** (*str or list*) – Set colorbar 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*]][**+o***d*
  Define 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-character 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 colorbar. 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-character 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`, draw 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 [Default is `"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, 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. Alter-

natively, set `shading=[low, high]` to specify an asymmetric intensity range from *low* to *high*. [Default is no illumination].

- **equalsize** (*float or str*) – [**i**][*gap*]. Equal-sized color rectangles. By default, the rectangles are scaled according to the z-range in the CPT (see also `zfile`). If *gap* is appended and the CPT is discrete each annotation is centered on each rectangle, using the lower boundary z-value for the annotation. If **i** is prepended the interval range is annotated instead. If `shading` is used each rectangle will have its constant color modified by the specified intensity.
- **log** (*bool*) – Select logarithmic scale and power of ten annotations. All z-values in the CPT will be converted to p = log10(z) and only integer p-values will be annotated using the 10^p format [Default is linear scale].
- **zfile** (*str*) – File with colorbar-width per color entry. By default, the width of the entry is scaled to the color range, i.e., z = 0-100 gives twice the width as z = 100-150 (see also `equalsize`). **Note**: The widths may be in plot distance units or given as relative fractions and will be automatically scaled so that the sum of the widths equals the requested colorbar length.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **panel** (*bool, 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/6.5/gmt.html#perspective-full.
- **transparency** (*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).

Example

```
>>> import pygmt
>>> # Create a new figure instance with pygmt.Figure()
>>> fig = pygmt.Figure()
>>> # Create a basemap
>>> fig.basemap(region=[0, 10, 0, 3], projection="X10c/3c", frame=True)
>>> # Call the colorbar method for the plot
>>> fig.colorbar(
...     # Set cmap to the "roma" CPT
...     cmap="roma",
...     # Label the x-axis "Velocity" and the y-axis "m/s"
...     frame=["x+lVelocity", "y+lm/s"],
... )
>>> # Show the plot
>>> fig.show()
```

Examples using `pygmt.Figure.colorbar`

pygmt.Figure.contour

`Figure.`**`contour`**(*data=None*, *x=None*, *y=None*, *z=None*, ***kwargs*)

Contour table data by direct triangulation.

Takes a matrix, (x, y, z) triplets, 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`, and `z`.

Full option list at https://docs.generic-mapping-tools.org/6.5/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
- V = verbose
- W = pen
- b = binary
- c = panel
- d = nodata
- e = find
- f = coltypes
- h = header
- i = incols
- l = label
- p = perspective
- t = transparency

Parameters

- **data** (`str` | `PathLike` | `dict` | `ndarray` | `DataFrame` | `Dataset` | `GeoDataFrame` | `None`, default: `None`) – Pass in (x, y, z) or (longitude, latitude, elevation) values by providing a file name to an ASCII data table, a 2-D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1-D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **x/y/z** (*1-D arrays*) – Arrays of x and y coordinates and values z of the data points.
- **projection** (*str*) – *projcode*[*projparams*/]*width*|*scale*. Select map *projection*.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **annotation** (*float, list, or str*) – Specify or disable annotated contour levels, modifies annotated contours specified in `levels`.
  - Specify a fixed annotation interval.
  - Specify a list of annotation levels.
  - Disable all annotations by setting `annotation="n"`.
  - Adjust the appearance by appending different modifiers, e.g., `"annot_int+f10p+gred"` gives annotations with a font size of 10 points and a red filled box. For all available modifiers see https://docs.generic-mapping-tools.org/6.5/contour.html#a.
- **frame** (*bool, str, or list*) – Set map boundary *frame and axes attributes*.
- **levels** (*float, list, or str*) – Specify the contour lines to generate.
  - The file name of a CPT file where the color boundaries will be used as contour levels.
  - The file name of a 2 (or 3) column file containing the contour levels (col 0), (**C**)ontour or (**A**)nnotate (col 1), and optional angle (col 2).
  - A fixed contour interval.
  - A list of contour levels.
- **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/6.5/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 frame boundaries [Default is `False` 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 or list*) – [*type*]*pen*[**+c**[**l**|**f**]]. *type*, if present, can be **a** for annotated contours or **c** for regular contours [Default]. The pen sets the attributes for the particular line. Default pen for annotated contours is `"0.75p,black"` and for regular contours `"0.25p,black"`. Normally, all contours are drawn with a fixed color determined by the pen setting. If **+cl** is appended the colors of the contour lines are taken from the CPT (see `levels`). If **+cf** is appended the colors from the CPT file are applied to the contour annotations. Select **+c** for both effects.
- **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.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **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/6.5/gmt.html#bi-full.
- **panel** (*bool, 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, `nodata="-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/6.5/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 1-D 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 *1-D 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/6.5/gmt.html#perspective-full.

- **transparency** (*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`

pygmt.Figure.grdcontour

Figure.**grdcontour**(*grid*, ***kwargs*)

Make contour map using a grid.

Takes a grid file name or an `xarray.DataArray` object as input.

Full option list at https://docs.generic-mapping-tools.org/6.5/grdcontour.html

Aliases:

- A = annotation
- B = frame
- C = levels
- G = label_placement
- J = projection
- L = limit
- Q = cut
- R = region
- S = resample
- V = verbose
- W = pen
- c = panel
- f = coltypes
- l = label
- p = perspective
- t = transparency

Parameters

- **grid** (`str` | `PathLike` | `DataArray`) – Name of the input grid file or the grid loaded as a `xarray.DataArray` object.

  For reading a specific grid file format or applying basic data operations, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **levels** (*float, list, or str*) – Specify the contour lines to generate.
  - The file name of a CPT file where the color boundaries will be used as contour levels.
  - The file name of a 2 (or 3) column file containing the contour levels (col 0), (**C**)ontour or (**A**)nnotate (col 1), and optional angle (col 2).
  - A fixed contour interval.
  - A list of contour levels.
- **annotation** (*float, list, or str*) – Specify or disable annotated contour levels, modifies annotated contours specified in `levels`.
  - Specify a fixed annotation interval.
  - Specify a list of annotation levels.
  - Disable all annotations by setting `annotation="n"`.
  - Adjust the appearance by appending different modifiers, e.g., `"annot_int+f10p+gred"` gives annotations with a font size of 10 points and a red filled box. For all available modifiers see https://docs.generic-mapping-tools.org/6.5/grdcontour.html#a.
- **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*|*scale*. Select map *projection*.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **frame** (*bool, 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/6.5/grdcontour.html#g for details.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **pen** (*str or list*) – [*type*]*pen*[**+c**[**l**|**f**]]. *type*, if present, can be **a** for annotated contours or **c** for regular contours [Default]. The pen sets the attributes for the particular line. Default pen for annotated contours is `"0.75p,black"` and for regular contours `"0.25p,black"`. Normally, all contours are drawn with a fixed color determined by the pen setting. If **+cl** is appended the colors of the contour lines are taken from the CPT (see `levels`). If **+cf** is appended the colors from the CPT file are applied to the contour annotations. Select **+c** for both effects.
- **panel** (*bool, 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/6.5/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/6.5/gmt.html#perspective-full.
- **transparency** (*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).

Example

```
>>> import pygmt
>>> # Load the 15 arc-minutes grid with "gridline" registration in the
>>> # specified region
>>> grid = pygmt.datasets.load_earth_relief(
...     resolution="15m",
...     region=[-92.5, -82.5, -3, 7],
...     registration="gridline",
... )
>>> # Create a new plot with pygmt.Figure()
>>> fig = pygmt.Figure()
>>> # Create the contour plot
>>> fig.grdcontour(
...     # Pass in the grid downloaded above
```

(continues on next page)

(continued from previous page)

```
...     grid=grid,
...     # Set the interval for contour lines at 250 meters
...     levels=250,
...     # Set the interval for annotated contour lines at 1,000 meters
...     annotation=1000,
...     # Add a frame for the plot
...     frame="a",
...     # Set the projection to Mercator for the 10 cm figure
...     projection="M10c",
... )
>>> # Show the plot
>>> fig.show()
```

Examples using `pygmt.Figure.grdcontour`

pygmt.Figure.grdimage

`Figure.`**`grdimage`**(*grid*, ***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 https://docs.generic-mapping-tools.org/6.5/grdmath.html or *pygmt.grdhisteq*. Alternatively, pass *image* which 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 option list at https://docs.generic-mapping-tools.org/6.5/grdimage.html

Aliases:

- B = frame
- C = cmap
- D = img_in
- E = dpi
- G = bitcolor
- I = shading
- J = projection
- M = monochrome
- N = no_clip
- Q = nan_transparent
- R = region
- V = verbose
- c = panel
- f = coltypes
- n = interpolation
- p = perspective
- t = transparency
- x = cores

Parameters

- **`grid`** (`str` | `PathLike` | `DataArray`) – Name of the input grid file or the grid loaded as a `xarray.DataArray` object.

For reading a specific grid file format or applying basic data operations, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **frame** (*bool, 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*]. Set the resolution of the projected grid that will be created if a map projection other than Linear or Mercator was selected [Default is `100` dpi]. 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.
- **bitcolor** (*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 function, 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 represent an *image* then an *intensfile* or constant *intensity* must be provided.
- **projection** (*str*) – *projcode*[*projparams*/]*width*|*scale*. 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 frame boundaries (only relevant for non-rectangular maps) [Default is `False`].
- **nan_transparent** (*bool or str*) – [**+z***value*][*color*] Make grid nodes with z = NaN transparent, using the color-masking feature in PostScript Level 3 (the PS device must support PS Level 3). If the input is a grid, use **+z** to select another grid value than NaN. If input is instead an image, append an alternate *color* to select another pixel value to be transparent [Default is `"black"`].
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **panel** (*bool, 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/6.5/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/6.5/gmt.html#perspective-full.
- **transparency** (*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*) – Specify the number of active cores to be used in any OpenMP-enabled multi-threaded algorithms. By default, all available cores are used. Set a positive number *n* to use *n* cores (if too large it will be truncated to the maximum cores available); or set a negative number *-n* to select (all - *n*) cores (or at least 1 if *n* equals or exceeds all).

Example

```
>>> import pygmt
>>> # load the 30 arc-minutes grid with "gridline" registration
>>> grid = pygmt.datasets.load_earth_relief("30m", registration="gridline")
>>> # create a new plot with pygmt.Figure()
>>> fig = pygmt.Figure()
>>> # pass in the grid and set the CPT to "geo"
>>> # set the projection to Mollweide and the size to 10 cm
>>> fig.grdimage(grid=grid, cmap="geo", projection="W10c", frame="ag")
>>> # show the plot
>>> fig.show()
```

Examples using `pygmt.Figure.grdimage`

pygmt.Figure.grdview

Figure.**grdview**(*grid*, ***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/6.5/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
- c = panel
- f = coltypes
- n = interpolation
- p = perspective
- t = transparency

Parameters

- **grid** (str | PathLike | DataArray) – Name of the input grid file or the grid loaded as a xarray.DataArray object.

  For reading a specific grid file format or applying basic data operations, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **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 given by the input grid].

- **projection** (*str*) – *projcode*[*projparams*/]*width*|*scale*. Select map *projection*.

- **zscale/zsize** (*float or str*) – Set z-axis scaling or z-axis size.

- **frame** (*bool, 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 xarray.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 refer to grid. 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*]. Draw 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*) – Specify 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*) – Set the pen attributes used for the mesh. You must also select surftype of **m** or **sm** for meshlines to be drawn.

- **facadepen** (*str*) – Set 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 main input data grid by using *pygmt.grdgradient* first; append **+a***azimuth*, **+n***args*, and **+m***ambient* to specify azimuth, intensity, and ambient arguments for that function, or just give **+d** to select the default arguments [Default is "+a-45+nt1+m0"].

- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].

- **panel** (*bool, 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/6.5/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/6.5/gmt.html#perspective-full.
- **transparency** (*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).

Example

```
>>> import pygmt
>>> # Load the 30 arc-minutes grid with "gridline" registration in a given region
>>> grid = pygmt.datasets.load_earth_relief(
...     resolution="30m",
...     region=[-92.5, -82.5, -3, 7],
...     registration="gridline",
... )
>>> # Create a new figure instance with pygmt.Figure()
>>> fig = pygmt.Figure()
>>> # Create the contour plot
>>> fig.grdview(
...     # Pass in the grid downloaded above
...     grid=grid,
...     # Set the perspective to an azimuth of 130° and an elevation of 30°
...     perspective=[130, 30],
...     # Add a frame to the x- and y-axes
...     # Specify annotations on the south and east borders of the plot
...     frame=["xa", "ya", "wSnE"],
...     # Set the projection of the 2-D map to Mercator with a 10 cm width
...     projection="M10c",
...     # Set the vertical scale (z-axis) to 2 cm
...     zsize="2c",
...     # Set "surface plot" to color the surface via a CPT
...     surftype="s",
...     # Specify CPT to "geo"
...     cmap="geo",
```

(continues on next page)

(continued from previous page)

```
... )
>>> # Show the plot
>>> fig.show()
```

Examples using `pygmt.Figure.grdview`

pygmt.Figure.histogram

`Figure.`**`histogram`**(*data*, ***kwargs*)

Calculate and plot histograms.

Full option list at https://docs.generic-mapping-tools.org/6.5/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
- V = verbose
- W = pen
- 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` | `PathLike` | `dict` | `ndarray` | `DataFrame` | `Dataset` | `GeoDataFrame`) – Pass in either a file name to an ASCII data table, a Python list, a 2-D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1-D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **`projection`** (*`str`*) – *projcode*[*projparams*/]*width*|*scale*. Select map *projection*.
- **`region`** (*`str or list`*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **`frame`** (*`bool, 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.
- **`fill`** (*`str`*) – Set color or pattern for filling bars [Default is no fill].
- **`pen`** (*`str`*) – Set pen attributes for lines or the outline of symbols.
- **`panel`** (*`bool, 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 [Default is `"6p"`]; use **+r** to rotate the labels from horizontal to vertical.
- **`barwidth`** (*`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, float, or str*) – [*mode*][**+p***pen*]. Draw the equivalent normal distribution; append desired *pen* [Default is `"0.25p,black,solid"`]. 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*) – Draw a stairs-step diagram which does not include the internal bars of the default histogram.
- **horizontal** (*bool*) – Plot the histogram horizontally from x = 0 [Default is vertically from y = 0]. The plot dimensions remain the same, but the two axes are flipped, i.e., the x-axis is plotted vertically and the y-axis is plotted horizontally.
- **series** (*int, 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**.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **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/6.5/gmt.html#bi-full.

- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `nodata="-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 1-D 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 *1-D 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/6.5/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/6.5/gmt.html#perspective-full.
- **transparency** (*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/6.5/gmt.html#w-full.

Examples using `pygmt.Figure.histogram`

pygmt.Figure.hlines

`Figure.`**`hlines`**(*y*, *xmin=None*, *xmax=None*, *pen=None*, *label=None*, *no_clip=False*, *perspective=None*)

Plot one or multiple horizontal line(s).

This method is a high-level wrapper around *pygmt.Figure.plot* that focuses on plotting horizontal lines at Y-coordinates specified by the `y` parameter. The `y` parameter can be a single value (for a single horizontal line) or a sequence of values (for multiple horizontal lines).

By default, the X-coordinates of the start and end points of the lines are set to be the X-limits of the current plot, but this can be overridden by specifying the `xmin` and `xmax` parameters. `xmin` and `xmax` can be either a single value or a sequence of values. If a single value is provided, it is applied to all lines. If a sequence is provided, the length of `xmin` and `xmax` must match the length of `y`.

The term "horizontal" lines can be interpreted differently in different coordinate systems:

- **Cartesian**: lines are plotted as straight lines.
- **Polar**: lines are plotted as arcs along a constant radius.
- **Geographic**: lines are plotted as arcs along parallels (i.e., constant latitude).

Parameters

- **y** (float | Sequence[float]) – Y-coordinates to plot the lines. It can be a single value (for a single line) or a sequence of values (for multiple lines).
- **xmin/xmax** – X-coordinates of the start/end point(s) of the line(s). If None, defaults to the X-limits of the current plot. xmin and xmax can be either a single value or a sequence of values. If a single value is provided, it is applied to all lines. If a sequence is provided, the length of xmin and xmax must match the length of y.
- **pen** (str | None, default: None) – Pen attributes for the line(s), in the format of *width,color,style*.
- **label** (str | None, default: None) – Label for the line(s), to be displayed in the legend.
- **no_clip** (bool, default: False) – If True, do not clip lines outside the plot region. Only makes sense in the Cartesian coordinate system.
- **perspective** (str | bool | None, default: None) – Select perspective view and set the azimuth and elevation angle of the viewpoint. Refer to *pygmt.Figure.plot* for details.

Examples

```
>>> import pygmt
>>> fig = pygmt.Figure()
>>> fig.basemap(region=[0, 10, 0, 10], projection="X10c/10c", frame=True)
>>> fig.hlines(y=1, pen="1p,black", label="Line at y=1")
>>> fig.hlines(y=2, xmin=2, xmax=8, pen="1p,red,-", label="Line at y=2")
>>> fig.hlines(y=[3, 4], xmin=3, xmax=7, pen="1p,black,.", label="Lines at y=3,4")
>>> fig.hlines(y=[5, 6], xmin=4, xmax=9, pen="1p,red", label="Lines at y=5,6")
>>> fig.hlines(
...     y=[7, 8], xmin=[0, 1], xmax=[7, 8], pen="1p,blue", label="Lines at y=7,8"
... )
>>> fig.legend()
>>> fig.show()
```

Examples using `pygmt.Figure.hlines`

pygmt.Figure.image

Figure.**image**(*imagefile*, ***kwargs*)

Plot raster or EPS images.

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/6.5/image.html

Aliases:

- D = position
- F = box
- G = bitcolor
- J = projection
- M = monochrome
- R = region
- V = verbose
- 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*|*scale*. 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*]]. Set 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*]]]. If set to `True`, draw a rectangular border around the image using MAP_FRAME_PEN.
- **bitcolor** (*str or list*) – [*color*][**+b**|**f**|**t**]. Change certain pixel values to another color or make them transparent. For 1-bit images you can specify an alternate *color* for the background (**+b**) or the foreground (**+f**) pixels, or give no color to make those pixels transparent. Can be repeated with different settings. Alternatively, for color images you can select a single *color* that should be made transparent instead (**+t**).
- **monochrome** (*bool*) – Convert color image to monochrome grayshades using the (television) YIQ-transformation.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **panel** (*bool, 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/6.5/gmt.html#perspective-full.
- **transparency** (*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`

pygmt.Figure.inset

`Figure.`**inset**(*, *position=None*, *box=None*, *projection=None*, *margin=None*, *no_clip=None*, *region=None*, *verbose=None*, ***kwargs*)

Manage figure inset setup and completion.

This method sets the position, frame, and margins for a smaller figure inside of the larger figure. Plotting methods that are called within the context manager are added to the inset figure.

Full option list at https://docs.generic-mapping-tools.org/6.5/inset.html

Aliases:

- D = position
- F = box
- J = projection
- M = margin
- N = no_clip
- R = region
- 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-character 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, centimeters, 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 meters (**e**)].

Append **+w***width*[/*height*] of bounding rectangle or box in plot coordinates (inches, centimeters, 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-character 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 set to `True`, draw a rectangular box around the map inset using the default pen; specify a different pen with **+p***pen*. Add **+g***fill* to fill the inset 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-directions, or individual side spacings between map embellishment and border. Append **+i** to draw a secondary, inner border as well. We use a uniform *gap* between borders of 2p 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 [Default is `"4p/-4p"`] and *shade* sets the fill style to use for shading [Default is `"gray50"`].
- **margin** (*float, 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 the inset frame boundaries [Default is `False`].
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **projection** (*str*) – *projcode*[*projparams*/]*width*|*scale*. Select map *projection*.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].

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
```

(continues on next page)

(continued from previous page)

```
>>> 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" statement are plotted in the main
>>> # figure
>>> fig.logo(position="jBR+o0.2c+w3c")
>>> fig.show()
```

Examples using `pygmt.Figure.inset`

pygmt.Figure.legend

`Figure.`**`legend`**(*spec=None*, *position='JTR+jTR+o0.2c'*, *box='+gwhite+p1p'*, ***kwargs*)

Plot a legend.

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/6.5/legend.html

Aliases:

- D = position
- F = box
- J = projection
- R = region
- V = verbose
- c = panel
- p = perspective
- t = transparency

Parameters

- **spec** (`str` | `PathLike` | `StringIO` | `None`, default: `None`) – The legend specification. It can be:
  - `None` which means using the automatically generated legend specification file
  - Path to the legend specification file
  - A `io.StringIO` object containing the legend specification

  See https://docs.generic-mapping-tools.org/6.5/legend.html for the definition of the legend specification.
- **projection** (*str*) – *projcode*[*projparams*/]*width*|*scale*. 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*]]. Define 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*]]]. If set to `True`, draw 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.

- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **panel** (*bool, 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/6.5/gmt.html#perspective-full.
- **transparency** (*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`

pygmt.Figure.logo

`Figure.`**`logo`**(*, *region=None*, *projection=None*, *position=None*, *box=None*, *style=None*, *verbose=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/6.5/gmtlogo.html.

Aliases:

- D = position
- F = box
- J = projection
- R = region
- S = style
- V = verbose
- c = panel
- t = transparency

Parameters

- **projection** (*str*) – *projcode*[*projparams*/]*width*|*scale*. 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*]]. Set reference point on the map for the image.
- **box** (*bool or str*) – If set to `True`, draw 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
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **panel** (*bool, 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`** (*`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`

pygmt.Figure.meca

`Figure.`**`meca`**(*spec*, *scale*, *convention=None*, *component='full'*, *longitude=None*, *latitude=None*, *depth=None*, *plot_longitude=None*, *plot_latitude=None*, *event_name=None*, ***kwargs*)

Plot focal mechanisms.

The following focal mechanism conventions are supported:

Table 3: Supported focal mechanism conventions.

Convention	Description	Focal parameters	Remark
`"aki"`	Aki and Richard	*strike*, *dip*, *rake*, *magnitude*	angles in degrees
`"gcmt"`	global centroid moment tensor	*strike1*, *dip1*, *rake1*, *strike2*, *dip2*, *rake2*, *mantissa*, *exponent*	angles in degrees; seismic moment is $mantissa * 10^{exponent}$ in dyn cm
`"mt"`	seismic moment tensor	*mrr*, *mtt*, *mff*, *mrt*, *mrf*, *mtf*, *exponent*	moment components in $10^{exponent}$ dyn cm
`"partial"`	partial focal mechanism	*strike1*, *dip1*, *strike2*, *fault_type*, *magnitude*	angles in degrees; *fault_type* means +1/-1 for normal/reverse fault
`"principal_axis"`	principal axis	*t_value*, *t_azimuth*, *t_plunge*, *n_value*, *n_azimuth*, *n_plunge*, *p_value*, *p_azimuth*, *p_plunge*, *exponent*	values in $10^{exponent}$ dyn cm; azimuths and plunges in degrees

Full option list at https://docs.generic-mapping-tools.org/6.5/supplements/seis/meca.html

Aliases:

- A = offset
- B = frame
- C = cmap
- E = extensionfill
- Fr = labelbox
- G = compressionfill
- J = projection
- L = outline
- N = no_clip
- R = region
- T = nodal
- V = verbose
- W = pen
- c = panel
- p = perspective
- t = transparency

Parameters

- **spec** (*str, 1-D numpy array, 2-D numpy array, dict, or pandas.DataFrame*) – Data that contain focal mechanism parameters.

  `spec` can be specified in either of the following types:

  – *str*: a file name containing focal mechanism parameters as columns. The meaning of each column is:

    * Columns 1 and 2: event longitude and latitude
    * Column 3: event depth (in kilometers)
    * Columns 4 to 3+n: focal mechanism parameters. The number of columns *n* depends on the choice of `convention` (see the table above for the supported conventions).
    * Columns 4+n and 5+n: longitude and latitude at which to place the beachball. `0 0` plots the beachball at the longitude and latitude given in the columns 1 and 2. [optional; requires `offset=True`].
    * Last Column: text string to appear near the beachball [optional].

  – *1-D np.array*: focal mechanism parameters of a single event. The meanings of columns are the same as above.

  – *2-D np.array*: focal mechanism parameters of multiple events. The meanings of columns are the same as above.

  – *dict* or `pandas.DataFrame`: The dict keys or `pandas.DataFrame` column names determine the focal mechanism convention. For the different conventions, the combination of keys / column names as given in the table above are required.

  A dict may contain values for a single focal mechanism or lists of values for multiple focal mechanisms.

  Both dict and `pandas.DataFrame` may optionally contain the keys / column names: `latitude`, `longitude`, `depth`, `plot_longitude`, `plot_latitude`, and/or `event_name`.

  If `spec` is either a str or a 1-D or 2-D numpy array, the `convention` parameter is required to interpret the columns. If `spec` is a dict or a `pandas.DataFrame`, `convention` is not needed and ignored if specified.

- **scale** (*float or str*) – *scale*[**+a***angle*][**+f***font*][**+j***justify*][**+l**][**+m**][**+o***dx*[/*dy*]][**+s***reference*]. Adjust scaling of the radius of the beachball, which is proportional to the magnitude. By default, *scale* defines the size for magnitude = 5 (i.e., scalar seismic moment M0 = 4.0E23 dyn cm). If **+l** is used the radius will be proportional to the seismic moment instead. Use **+s** and give a *reference* to change the reference magnitude (or moment), and use **+m** to plot all beachballs with the same size. A text string can be specified to appear near the beachball (corresponding to column or parameter `event_name`). Append **+a***angle* to change the angle of the text string; append **+f***font* to change its font (size,fontname,color); append **+j***justify* to change the text location relative to the beachball [Default is `"TC"`, i.e., Top Center]; append **+o** to offset the text string by *dx*/*dy*.

- **convention** (Optional[Literal['aki', 'gcmt', 'mt', 'partial', 'principal_axis']], default: None) – Specify the focal mechanism convention of the input data. Ignored if spec is a dict or pandas.DataFrame. See the table above for the supported conventions.
- **component** (Literal['full', 'dc', 'deviatoric'], default: 'full') – The component of the seismic moment tensor to plot. Valid values are:
  - "full": the full seismic moment tensor
  - "dc": the closest double couple defined from the moment tensor (zero trace and zero determinant)
  - "deviatoric": deviatoric part of the moment tensor (zero trace)
- **longitude/latitude/depth** – Longitude(s), latitude(s), and depth(s) of the event(s). The length of each must match the number of events. These parameters are only used if spec is a dictionary or a pandas.DataFrame, and they override any existing longitude, latitude, or depth values in spec.
- **plot_longitude/plot_latitude** – Longitude(s) and latitude(s) at which to place the beachball(s). The length of each must match the number of events. These parameters are only used if spec is a dictionary or a pandas.DataFrame, and they override any existing plot_longitude or plot_latitude values in spec.
- **event_name** (str | Sequence[str] | None, default: None) – Text string(s), such as event name(s), to appear near the beachball(s). The length must match the number of events. This parameter is only used if spec is a dictionary or a pandas.DataFrame, and it overrides any existing event_name labels in spec.
- **labelbox** (*bool or str*) – [*fill*]. Draw a box behind the label if given via event_name. Use *fill* to give a fill color [Default is "white"].
- **offset** (*bool or str*) – [**+p***pen*][**+s***size*]. Offset beachball(s) to the longitude(s) and latitude(s) specified in the last two columns of the input file or array, or by plot_longitude and plot_latitude if provided. A line from the beachball to the initial location is drawn. Use **+s***size* to plot a small circle at the initial location and to set the diameter of this circle [Default is no circle]. Use **+p***pen* to set the pen attributes for this feature [Default is set via pen]. The fill of the circle is set via compressionfill or cmap, i.e., corresponds to the fill of the compressive quadrants.
- **compressionfill** (*str*) – Set color or pattern for filling compressive quadrants [Default is "black"]. This setting also applies to the fill of the circle defined via offset.
- **extensionfill** (*str*) – Set color or pattern for filling extensive quadrants [Default is "white"].
- **pen** (*str*) – Set (default) pen attributes for all lines related to the beachball [Default is "0.25p,black,solid"]. This setting applies to outline, nodal, and offset, unless overruled by arguments passed to those parameters. Draws the circumference of the beachball.
- **outline** (*bool or str*) – [*pen*]. Draw circumference and nodal planes of the beachball. Use *pen* to set the pen attributes for this feature [Default is set via pen].
- **nodal** (*bool, int, or str*) – [*nplane*][/*pen*]. Plot the nodal planes and outline the bubble which is transparent. If *nplane* is
  - 0 or True: both nodal planes are plotted [Default].
  - 1: only the first nodal plane is plotted.
  - 2: only the second nodal plane is plotted.

Use */pen* to set the pen attributes for this feature [Default is set via `pen`]. For double couple mechanisms, `nodal` renders the beachball transparent by drawing only the nodal planes and the circumference. For non-double couple mechanisms, `nodal=0` overlays best double couple transparently.

- **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. The color of the compressive quadrants is determined by the z-value (i.e., event depth or the third column for an input file). This setting also applies to the fill of the circle defined via `offset`.
- **no_clip** (*bool*) – Do **not** skip symbols that fall outside the frame boundaries [Default is `False`, i.e., plot symbols inside the frame boundaries only].
- **projection** (*str*) – *projcode*[*projparams*/]*width*|*scale*. Select map *projection*.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **frame** (*bool, str, or list*) – Set map boundary *frame and axes attributes*.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **panel** (*bool, 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/6.5/gmt.html#perspective-full.
- **transparency** (*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`

pygmt.Figure.plot

`Figure.`**`plot`**(*data=None*, *x=None*, *y=None*, *size=None*, *symbol=None*, *direction=None*, *straight_line=False*, ***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`, `fill` can be a 1-D 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 `fill`. If `fill` is set, `pen` will control whether the polygon outline is drawn or not. If a symbol is selected, `fill` and `pen` determine the fill and outline/no outline, respectively.

Full option list at https://docs.generic-mapping-tools.org/6.5/plot.html

Aliases:

- A = straight_line
- B = frame
- C = cmap
- D = offset
- E = error_bar
- F = connection
- G = fill
- I = intensity
- J = projection
- L = close
- N = no_clip
- R = region
- S = style
- V = verbose
- W = pen
- 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 | PathLike | dict | ndarray | DataFrame | Dataset | GeoDataFrame | None, default: None) – Pass in either a file name to an ASCII data table, a 2-D numpy.ndarray, a pandas.DataFrame, an xarray.Dataset made up of 1-D xarray.DataArray data variables, or a geopandas.GeoDataFrame containing the tabular data. Use parameter incols to choose which columns are x, y, fill, and size, respectively.
- **x/y** (*float or 1-D arrays*) – The x and y coordinates, or arrays of x and y coordinates of the data points
- **size** (*1-D array*) – The size of the data points in units specified using style. Only valid if using x/y.
- **symbol** (*1-D array*) – The symbols of the data points. Only valid if using x/y.
- **direction** (*list of two 1-D arrays*) – If plotting vectors (using style="V" or style="v"), then should be a list of two 1-D 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*|*scale*. Select map *projection*.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **straight_line** (Union[bool, Literal['x', 'y']], default: False) – By default, line segments are drawn as straight lines in the Cartesian and polar coordinate systems, and as great circle arcs (by resampling coarse input data along such arcs) in the geographic coordinate system. The straight_line parameter can control the drawing of line segments. Valid values are:

  – True: Draw line segments as straight lines in geographic coordinate systems.

  – "x": Draw line segments by first along *x*, then along *y*.

  – "y": Draw line segments by first along *y*, then along *x*.

  Here, *x* and *y* have different meanings depending on the coordinate system:

  – **Cartesian** coordinate system: *x* and *y* are the X- and Y-axes.

  – **Polar** coordinate system: *x* and *y* are theta and radius.

  – **Geographic** coordinate system: *x* and *y* are parallels and meridians.

  Attention

  There exits a bug in GMT<=6.5.0 that, in geographic coordinate systems, the meaning of *x* and *y* is reversed, i.e., *x* means meridians and *y* means parallels. The bug is fixed by

upstream PR #8648.

- **frame** (*bool, 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*) – [**x**|**y**|**X**|**Y**][**+a**|**A**][**+cl**|**f**][**+n**][**+w***cap*][**+p***pen*]. Draw error bars. Full documentation is at https://docs.generic-mapping-tools.org/6.5/plot.html#e.
- **connection** (*str*) – [**c**|**n**|**p**][**a**|**r**|**s**|**t**|*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].
  - **n**: Draw networks of line segments between all points in each group.
  - **p**: Draw line segments from a reference point reset for 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 the reference point to the very first point of the first file.
  - **r**: Segment headers are honored so each segment is a group; the group reference point is reset after each record to the previous point (this method is only available with the `connection="p"` scheme).
  - **s**: Same as **r**, but the group reference point is reset to the first point of each incoming segment [Default].
  - **t**: 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.

  Instead of the codes **a**|**r**|**s**|**t** you may append the coordinates of a *refpoint* which will serve as a fixed external reference point for all groups.
- **fill** (*str*) – Set color or pattern for filling symbols or polygons [Default is no fill]. *fill* can be a 1-D array, but it is only valid if using `x`/`y` and `cmap=True` is also required.
- **intensity** (*float, bool, or 1-D 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 1-D 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/6.5/plot.html#l.
- **no_clip** (*bool or str*) – [**c**|**r**]. Do **not** clip symbols that fall outside the frame boundaries [Default plots points whose coordinates are strictly inside the frame boundaries 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.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **zvalue** (*str*) – *value*|*file*. Instead of specifying a symbol or polygon fill and outline color via `fill` 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 `fill="+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/6.5/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/6.5/gmt.html#bi-full.
- **panel** (*bool, 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, `nodata="-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/6.5/gmt.html#f-full.
- **gap** (*str or list*) – **x**|**y**|**z**|**d**|**X**|**Y**|**D***gap*[**u**][**+a**][**+c***col*][**+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.
  - **x**|**X**: define a gap when there is a large enough change in the x coordinates (uppercase to use projected coordinates).
  - **y**|**Y**: define a gap when there is a large enough change in the y coordinates (uppercase to use projected coordinates).
  - **d**|**D**: define a gap when there is a large enough distance between coordinates (uppercase to use projected coordinates).
  - **z**: define a gap when there is a large enough change in the z data. Use **+c***col* to change the z data column [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**(egrees), **m**(inutes), and **s**(econds) , or (m)**e**(ters), **f**(eet), **k**(ilometers), **M**(iles), or **n**(autical miles) [Default is (m)**e**(ters)].
  - For projected data (**X**|**Y**|**D**), the unit may be **i**(nches), **c**(entimeters), or **p**(oints).

  Append modifier **+a** to specify that *all* the criteria must be met [default imposes breaks if any one criterion is met].

  One of the following modifiers can be appended:
  - **+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 1-D 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 *1-D 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/6.5/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/6.5/gmt.html#perspective-full.
- **transparency** (*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 1-D 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/6.5/gmt.html#w-full.

Examples using `pygmt.Figure.plot`

pygmt.Figure.plot3d

`Figure.`**`plot3d`**(*data=None*, *x=None*, *y=None*, *z=None*, *size=None*, *symbol=None*, *direction=None*, *straight_line=False*, ***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`, and `z`.

If providing data through `x`, `y`, and `z`, `fill` can be a 1-D 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 `fill`. If `fill` is set, `pen` will control whether the polygon outline is drawn or not. If a symbol is selected, `fill` and `pen` determine the fill and outline/no outline, respectively.

Full option list at https://docs.generic-mapping-tools.org/6.5/plot3d.html

Aliases:

- A = straight_line
- B = frame
- C = cmap
- D = offset
- G = fill
- I = intensity
- J = projection
- JZ = zsize
- Jz = zscale
- L = close
- N = no_clip
- Q = no_sort
- R = region
- S = style
- V = verbose
- W = pen
- 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` | `PathLike` | `dict` | `ndarray` | `DataFrame` | `Dataset` | `GeoDataFrame` | `None`, default: `None`) – Either a data file name, a 2-D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1-D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data. Optionally, use parameter `incols` to specify which columns are x, y, z, fill, and size, respectively.
- **x/y/z** (*float or 1-D arrays*) – The x, y, and z coordinates, or arrays of x, y and z coordinates of the data points.
- **size** (*1-D array*) – The size of the data points in units specified in `style`. Only valid if using `x`/`y`/`z`.
- **symbol** (*1-D array*) – The symbols of the data points. Only valid if using `x`/`y`.
- **direction** (*list of two 1-D arrays*) – If plotting vectors (using `style="V"` or `style="v"`), then should be a list of two 1-D 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*|*scale*. 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** (`Union`[`bool`, `Literal`[`'x'`, `'y'`]], default: `False`) – By default, line segments are drawn as straight lines in the Cartesian and polar coordinate systems, and as great circle arcs (by resampling coarse input data along such arcs) in the geographic coordinate system. The `straight_line` parameter can control the drawing of line segments. Valid values are:
  - `True`: Draw line segments as straight lines in geographic coordinate systems.
  - `"x"`: Draw line segments by first along *x*, then along *y*.

– `"y"`: Draw line segments by first along *y*, then along *x*.

Here, *x* and *y* have different meanings depending on the coordinate system:

– **Cartesian** coordinate system: *x* and *y* are the X- and Y-axes.

– **Polar** coordinate system: *x* and *y* are theta and radius.

– **Geographic** coordinate system: *x* and *y* are parallels and meridians.

NOTE: The `straight_line` parameter requires constant *z*-coordinates.

Attention

There exits a bug in GMT<=6.5.0 that, in geographic coordinate systems, the meaning of *x* and *y* is reversed, i.e., *x* means meridians and *y* means parallels. The bug is fixed by upstream PR #8648.

- **frame** (*bool, 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].
- **fill** (*str*) – Set color or pattern for filling symbols or polygons [Default is no fill]. *fill* can be a 1-D array, but it is only valid if using `x`/`y` and `cmap=True` is also required.
- **intensity** (*float, bool, or 1-D 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 1-D 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/6.5/plot3d.html#l.
- **no_clip** (*bool or str*) – [**c**|**r**]. Do **not** clip symbols that fall outside the frame boundaries [Default plots points whose coordinates are strictly inside the frame boundaries 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/6.5/plot3d.html#s.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **pen** (*str*) – Set pen attributes for lines or the outline of symbols.
- **zvalue** (*str*) – *value*|*file*. Instead of specifying a symbol or polygon fill and outline color via `fill` 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 `fill="+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/6.5/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/6.5/gmt.html#bi-full.
- **panel** (*bool, 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, `nodata="-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/6.5/gmt.html#f-full.
- **gap** (*str or list*) – **x|y|z|d|X|Y|D***gap*[**u**][**+a**][**+c***col*][**+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.

- **x**|**X**: define a gap when there is a large enough change in the x coordinates (uppercase to use projected coordinates).
- **y**|**Y**: define a gap when there is a large enough change in the y coordinates (uppercase to use projected coordinates).
- **d**|**D**: define a gap when there is a large enough distance between coordinates (uppercase to use projected coordinates).
- **z**: define a gap when there is a large enough change in the z data. Use **+c***col* to change the z data column [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**(egrees), **m**(inutes), and **s**(econds) , or (m)**e**(ters), **f**(eet), **k**(ilometers), **M**(iles), or **n**(autical miles) [Default is (m)**e**(ters)].
- For projected data (**X**|**Y**|**D**), the unit may be **i**(nches), **c**(entimeters), or **p**(oints).

Append modifier **+a** to specify that *all* the criteria must be met [default imposes breaks if any one criterion is met].

One of the following modifiers can be appended:

- **+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 1-D 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 *1-D 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/6.5/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/6.5/gmt.html#perspective-full.
- **transparency** (*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 1-D 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/6.5/gmt.html#w-full.

Examples using `pygmt.Figure.plot3d`

pygmt.Figure.psconvert

`Figure.`**psconvert**(*, *crop=None*, *gs_option=None*, *dpi=None*, *prefix=None*, *gs_path=None*, *resize=None*, *bb_style=None*, *fmt=None*, *anti_aliasing=None*, *verbose=None*, ***kwargs*)

Convert [E]PS file(s) to other formats using Ghostscript.

Converts one or more PostScript files to other formats (BMP, EPS, JPEG, PDF, PNG, PPM, 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/6.5/psconvert.html

Aliases:

- A = crop
- C = gs_option
- E = dpi
- F = prefix
- G = gs_path
- I = resize
- N = bb_style
- 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. Default is `True`. Append **+u** to first remove any GMT-produced time-stamps. Append **+r** to *round* the HighResBoundingBox instead of using the `ceil` function. This is going against Adobe Law but can be useful when creating very small images where the difference of one pixel might matter. If `verbose` is used we also report the dimensions of the final illustration.
- **gs_path** (*str*) – Full path to the Ghostscript executable.
- **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 is 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.
- **resize** (*str*) – [**+m***margins*][**+s**[**m**]*width*[*/height*]][**+S***scale*]. Adjust the BoundingBox and HiResBoundingBox by scaling and/or adding margins. Append **+m** to specify extra margins to extend the bounding box. Give either one (uniform), two (x and y) or four (individual sides) margins; append unit [Default is set by PROJ_LENGTH_UNIT]. Append **+s***width* to resize the output image to exactly *width* units. The default unit is set by PROJ_LENGTH_UNIT but you can append a new unit and/or impose different width and height (**Note**: This may change the image aspect ratio). What happens here is that Ghostscript will do the re-interpolation work and the final image will retain the DPI resolution set by `dpi`. Append **+sm** to set a maximum size and the new *width* is only imposed if the original figure width exceeds it. Append */height* to also impose a maximum height in addition to the width. Alternatively, append **+S***scale* to scale the image by a constant factor.
- **bb_style** (*str*) – Set optional BoundingBox fill color, fading, or draw the outline of the BoundingBox. Append **+f***fade* to fade the entire plot towards black (100%) [no fading, 0]. Append **+g***paint* to paint the BoundingBox behind the illustration and append **+p**[*pen*] to draw the BoundingBox outline (append a pen or accept the default pen of `"0.25p,black,solid"`). **Note**: If both **+g** and **+f** are used then we use *paint* as the fade color instead of black. Append **+i** to 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) [Default is `"4"`]. [Default is no anti-aliasing (same as bits = 1).]
- **fmt** (*str*) – Set 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, 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 [*Full usage*].

pygmt.Figure.rose

`Figure.`**`rose`**(*data=None*, *length=None*, *azimuth=None*, ***kwargs*)

Plot a polar histogram (rose, sector, windrose diagrams).

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/6.5/rose.html

Aliases:

- A = sector
- B = frame
- C = cmap
- D = shift
- Em = vectors
- F = no_scale
- G = fill
- I = inquire
- JX = diameter
- L = labels
- M = vector_params
- Q = alpha
- R = region
- S = norm
- T = orientation
- V = verbose
- W = pen
- Z = scale
- b = binary
- c = panel
- d = nodata
- e = find
- h = header
- i = incols
- p = perspective
- t = transparency
- w = wrap

Parameters

- **`data`** (`str` | `PathLike` | `dict` | `ndarray` | `DataFrame` | `Dataset` | `GeoDataFrame` | `None`, default: `None`) – Pass in either a file name to an ASCII data table, a 2-D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1-D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data. Use parameter `incols` to choose which columns are length and azimuth, respectively. If a file with only azimuths is given, use `incols` 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 1-D arrays`*) – Length and azimuth values, or arrays of length and azimuth values.
- **`orientation`** (*`bool`*) – Specify 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 `region` is given as (-90, 90) or (0, 180).
- **`region`** (*`str or list`*) – *r0/r1/az0/az1* or [*r0*, *r1*, *az0*, *az1*]. *Required if this is the first plot command*. Specify the `region` of interest in (*r*, *azimuth*) space. Here, *r0* is 0 and *r1* is the maximal 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`*) – Set the diameter of the rose diagram. If not given, then we default to a diameter of 7.5 cm.
- **`sector`** (*`float or str`*) – Give 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` 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 y label 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].
- **fill** (*str*) – Set color or pattern for filling 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*) – *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/6.5/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*) – Set the confidence level used to determine if the mean resultant is significant (i.e., Lord Rayleigh test for uniformity) [Default is `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.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **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/6.5/gmt.html#bi-full.

- **panel** (*bool, 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, `nodata="-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 1-D 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 *1-D 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/6.5/gmt.html#perspective-full.
- **transparency** (*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/6.5/gmt.html#w-full.

Examples using `pygmt.Figure.rose`

pygmt.Figure.savefig

`Figure.`**savefig**(*fname*, *transparent=False*, *crop=True*, *anti_alias=True*, *show=False*, *worldfile=False*, ***kwargs*)

Save the figure to an image file.

Supported image formats and their extensions:

Raster image formats

- BMP (`.bmp`)

- JPEG (`.jpg` or `.jpeg`)
- GeoTIFF (`.tiff`)
- PNG (`.png`)
- PPM (`.ppm`)
- TIFF (`.tif`)

Vector image formats

- EPS (`.eps`)
- PDF (`.pdf`)

Besides the above formats, you can also save the figure to a KML file (`.kml`), with a companion PNG file generated automatically. The KML file can be viewed in Google Earth.

You can pass in any keyword arguments that *pygmt.Figure.psconvert* accepts.

Parameters

- **fname** (`str` | `PathLike`) – The desired figure file name, including the extension. See the list of supported formats and their extensions above.
- **transparent** (`bool`, default: `False`) – Use a transparent background for the figure. Only valid for PNG format and the PNG file asscoiated with KML format.
- **crop** (`bool`, default: `True`) – Crop the figure canvas (page) to the plot area.
- **anti_alias** (`bool`, default: `True`) – Use anti-aliasing when creating raster images. Ignored if creating vector images. More specifically, it passes the arguments `"t2"` and `"g2"` to the `anti_aliasing` parameter of *pygmt.Figure.psconvert*.
- **show** (`bool`, default: `False`) – Display the figure in an external viewer.
- **worldfile** (`bool`, default: `False`) – Create a companion world file for the figure. The world file will have the same name as the figure file but with different extension (e.g., `.tfw` for `.tif`). See https://en.wikipedia.org/wiki/World_file#Filename_extension for the convention of world file extensions. This parameter only works for raster image formats (except GeoTIFF).
- ****kwargs** (*dict*) – Additional keyword arguments passed to *pygmt.Figure.psconvert*. Valid parameters are `dpi`, `gs_path`, `gs_option`, `resize`, `bb_style`, and `verbose`.

Return type

`None`

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 methods 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 [*Full usage*].

Examples using `pygmt.Figure.set_panel`

pygmt.Figure.shift_origin

`Figure.`**`shift_origin`**(*xshift=None*, *yshift=None*)

Shift the plot origin in x and/or y directions.

The shifts can be permanent or temporary. If used as a standalone method, the shifts are permanent and apply to all subsequent plots. If used as a context manager, the shifts are temporary and only apply to the block of code within the context manager.

1. Use as a standalone method to shift the plot origin permanently:

```
fig.shift_origin(...)
...  # Other plot commands
```

2. Use as a context manager to shift the plot origin temporarily:

```
with fig.shift_origin(...):
    ...  # Other plot commands
    ...
```

The shifts *xshift* and *yshift* in x and y directions are relative to the current plot origin. The default unit for shifts is centimeters (**c**) but can be changed to other units via PROJ_LENGTH_UNIT. Optionally, append the length unit (**c** for centimeters, **i** for inches, or **p** for points) to the shifts.

For *xshift*, a special character **w** can also be used, which represents the bounding box **width** of the previous plot. The full syntax is [[±][*f*]**w**[/*d*]±]*xoff*, where optional signs, factor *f* and divisor *d* can be used to compute an offset that may be adjusted further by ±*xoff*. Assuming that the previous plot has a width of 10 centimeters, here are some example values for *xshift*:

- `"w"`: x-shift is 10 cm

- `"w+2c"`: x-shift is 10+2=12 cm
- `"2w+3c"`: x-shift is 2*10+3=23 cm
- `"w/2-2c"`: x-shift is 10/2-2=3 cm

Similarly, for *yshift*, a special character **h** can also be used, which is the bounding box **height** of the previous plot.

Note: The previous plot bounding box refers to the last object plotted, which may be a basemap, image, logo, legend, colorbar, etc.

Parameters

- **xshift** (`float` | `str` | `None`, default: `None`) – Shift plot origin in x direction.
- **yshift** (`float` | `str` | `None`, default: `None`) – Shift plot origin in y direction.

Examples

Shifting the plot origin permanently:

```
>>> import pygmt
>>> fig = pygmt.Figure()
>>> fig.basemap(region=[0, 5, 0, 5], projection="X5c/5c", frame=True)
>>> # Shift the plot origin in x direction by 6 cm
>>> fig.shift_origin(xshift=6)
<contextlib._GeneratorContextManager object at ...>
>>> fig.basemap(region=[0, 7, 0, 5], projection="X7c/5c", frame=True)
>>> # Shift the plot origin in x direction based on the previous plot width.
>>> # Here, the width is 7 cm, and xshift is 8 cm.
>>> fig.shift_origin(xshift="w+1c")
<contextlib._GeneratorContextManager object at ...>
>>> fig.basemap(region=[0, 10, 0, 5], projection="X10c/5c", frame=True)
>>> fig.show()
```

Shifting the plot origin temporarily:

```
>>> fig = pygmt.Figure()
>>> fig.basemap(region=[0, 5, 0, 5], projection="X5c/5c", frame=True)
>>> # Shift the plot origin in x direction by 6 cm temporarily. The plot origin␣
↪will
>>> # revert back to the original plot origin after the block of code is executed.
>>> with fig.shift_origin(xshift=6):
...     fig.basemap(region=[0, 5, 0, 5], projection="X5c/5c", frame=True)
>>> # Shift the plot origin in y direction by 6 cm temporarily.
>>> with fig.shift_origin(yshift=6):
...     fig.basemap(region=[0, 5, 0, 5], projection="X5c/5c", frame=True)
>>> # Shift the plot origin in x and y directions by 6 cm temporarily.
>>> with fig.shift_origin(xshift=6, yshift=6):
...     fig.basemap(region=[0, 5, 0, 5], projection="X5c/5c", frame=True)
>>> fig.show()
```

Examples using `pygmt.Figure.shift_origin`

pygmt.Figure.show

Figure.**show**(*method=None*, *dpi=300*, *width=500*, *waiting=0.5*, ***kwargs*)

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).

Use *pygmt.set_display* to select the default display method (`"notebook"`, `"external"`, `"none"` or `None`).

The `method` parameter allows to override the default display method for the current figure. The parameters `dpi` and `width` can be used to control the resolution and dimension of the figure in the notebook.

The external viewer can be disabled by setting the environment variable *PYGMT_USE_EXTERNAL_DISPLAY* to `"false"`. This is useful when running tests and building the documentation to avoid popping up windows.

The external viewer does not block the current process, thus it's necessary to suspend the execution of the current process for a short while after launching the external viewer, so that the preview image won't be deleted before the external viewer tries to open it. Set the `waiting` parameter to a larger number if the image viewer on your computer is slow to open the figure.

Parameters

- **method** (`Literal['external', 'notebook', 'none', None]`, default: `None`) – The method to display the current image preview. Choose from:
  - `"external"`: External PDF preview using the default PDF viewer
  - `"notebook"`: Inline PNG preview in the current notebook
  - `"none"`: Disable image preview
  - `None`: Reset to the default display method

  The default display method is `"external"` in Python consoles and `"notebook"` in Jupyter notebooks, but can be changed by *pygmt.set_display*.
- **dpi** (`int`, default: `300`) – The image resolution (dots per inch) in Jupyter notebooks.
- **width** (`int`, default: `500`) – The image width (in pixels) in Jupyter notebooks.
- **waiting** (`float`, default: `0.5`) – Suspend the execution of the current process for a given number of seconds after launching an external viewer. Only works if `method="external"`.
- ****kwargs** (*dict*) – Additional keyword arguments passed to *pygmt.Figure.psconvert*. Valid parameters are `gs_path`, `gs_option`, `resize`, `bb_style`, and `verbose`.

Return type

`None`

Examples using `pygmt.Figure.show`

pygmt.Figure.solar

Figure.**solar**(*terminator='day_night'*, *terminator_datetime=None*, ***kwargs*)

Plot day-night terminators and other sunlight parameters.

This function plots the day-night terminator. Alternatively, it can plot the terminators for civil twilight, nautical twilight, or astronomical twilight.

Full option list at https://docs.generic-mapping-tools.org/6.5/solar.html

Aliases:

- B = frame
- G = fill
- J = projection
- R = region
- V = verbose
- W = pen
- c = panel
- p = perspective
- t = transparency

Parameters

- **terminator** (Literal['astronomical', 'civil', 'day_night', 'nautical'], default: 'day_night') – Set the type of terminator displayed, which can be set with either the full name or the first letter of the name. Available options are:
  - "astronomical": Astronomical twilight
  - "civil": Civil twilight
  - "day_night": Day-night terminator
  - "nautical": Nautical twilight

  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 [Default is the current UTC date and time]. It can be passed as a string or Python datetime object.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **projection** (*str*) – *projcode*[*projparams*/]*width*|*scale*. Select map *projection*.
- **frame** (*bool, str, or list*) – Set map boundary *frame and axes attributes*.
- **fill** (*str*) – Set color or pattern for filling terminators [Default is no fill].
- **pen** (*str*) – Set pen attributes for lines [Default is "0.25p,black,solid"].
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **panel** (*bool, 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/6.5/gmt.html#perspective-full.
- **transparency** (*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).

Example

```
>>> # import the Python module "datetime"
>>> import datetime
>>> import pygmt
>>> # create a datetime object at 8:52:18 on June 24, 1997 (time in UTC)
>>> date = datetime.datetime(
...     year=1997, month=6, day=24, hour=8, minute=52, second=18
... )
>>> # create a new plot with pygmt.Figure()
>>> fig = pygmt.Figure()
```

(continues on next page)

(continued from previous page)

```
>>> # create a map of the Earth with the coast method
>>> fig.coast(land="darkgreen", water="lightblue", projection="W10c", region="d")
>>> fig.solar(
...     # set the terminator to "day_night"
...     terminator="day_night",
...     # pass the datetime object
...     terminator_datetime=date,
...     # fill the night-section with navyblue at 75% transparency
...     fill="navyblue@75",
...     # draw the terminator with a 1-point black line
...     pen="1p,black",
... )
>>> # show the plot
>>> fig.show()
```

Examples using `pygmt.Figure.solar`

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*, ***kwargs*)

Manage figure subplot configuration and selection.

This method 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/6.5/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

Parameters

- **`nrows`** (*int*) – Number of vertical rows of the subplot grid.
- **`ncols`** (*int*) – Number of horizontal columns of the subplot grid.
- **`figsize`** (*list*) – Specify the final figure dimensions as [*width*, *height*].
- **`subsize`** (*list*) – 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, 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 methods that plot scales, bars, text, etc.
- **projection** (*str*) – *projcode*[*projparams*/]*width*|*scale*. 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 [*Full usage*].

Examples using `pygmt.Figure.subplot`

pygmt.Figure.ternary

`Figure.`**`ternary`**(*data*, *alabel=None*, *blabel=None*, *clabel=None*, ***kwargs*)

Plot data on ternary diagrams.

Reads (*a*,*b*,*c*[,*z*]) records from *data* and plots symbols at those locations on a ternary diagram. If a symbol is selected and no symbol size given, then we 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 specified then we instead plot lines or polygons.

Full option list at https://docs.generic-mapping-tools.org/6.5/ternary.html

Aliases:

- B = frame
- C = cmap
- G = fill
- JX = width
- R = region
- S = style
- V = verbose
- W = pen
- c = panel
- p = perspective
- t = transparency

Parameters

- **data** (`str` | `PathLike` | `dict` | `ndarray` | `DataFrame` | `Dataset` | `GeoDataFrame`) – Pass in either a file name to an ASCII data table, a Python list, a 2-D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1-D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **width** (*str*) – Set the width of the figure by passing a number, followed by a unit (**i** for inches, **c** for centimeters). Use a negative width to indicate that positive axes directions be clock-wise [Default lets the a, b, c axes be positive in a counter-clockwise direction].
- **region** (*str or list*) – [*amin*, *amax*, *bmin*, *bmax*, *cmin*, *cmax*]. Give the min and max limits for each of the three axes **a**, **b**, and **c**.
- **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.
- **fill** (*str*) – Set color or pattern for filling symbols or polygons [Default is no fill].
- **alabel** (`str` | `None`, default: `None`) – Set the label for the *a* vertex where the component is 100%. The label is placed at a distance of three times the MAP_LABEL_OFFSET setting from the corner.
- **blabel** (`str` | `None`, default: `None`) – Same as `alabel` but for the *b* vertex.
- **clabel** (`str` | `None`, default: `None`) – Same as `alabel` but for the *c* vertex.
- **style** (*str*) – *symbol*[*size*]. Plot individual symbols in a ternary diagram.
- **pen** (*str*) – Set pen attributes for lines or the outline of symbols.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **panel** (*bool, 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/6.5/gmt.html#perspective-full.
- **transparency** (*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.ternary`

pygmt.Figure.text

`Figure.`**`text`**(*textfiles=None*, *x=None*, *y=None*, *position=None*, *text=None*, *angle=None*, *font=None*, *justify=None*, ***kwargs*)

Plot or typeset text.

Must provide at least one of the following combinations as input:

- `textfiles`
- `x`/`y`, and `text`
- `position` and `text`

The text strings passed via the `text` parameter can contain ASCII characters and non-ASCII characters defined in the Adobe ISOLatin1+, Adobe Symbol, Adobe ZapfDingbats and ISO-8859-x (x can be 1-11, 13-16) encodings. Refer to *Supported Encodings and Non-ASCII Characters* for the full list of supported non-ASCII characters.

Full option list at https://docs.generic-mapping-tools.org/6.5/text.html.

Aliases:

- B = frame
- C = clearance
- D = offset
- G = fill
- J = projection
- N = no_clip
- R = region
- V = verbose
- W = pen
- a = aspatial
- c = panel
- e = find
- f = coltypes
- h = header
- it = use_word
- p = perspective
- t = transparency
- w = wrap

Parameters

- **textfiles** (*str or list*) – A file name or a list of file names containing one or more records. Each record has the following columns:
  - *x*: X coordinate or longitude
  - *y*: Y coordinate or latitude
  - *angle*: Angle in degrees counter-clockwise from horizontal
  - *font*: Text size, font, and color
  - *justify*: Two-character justification code
  - *text*: The text string to typeset

  The *angle*, *font*, and *justify* columns are optional and can be set by using the `angle`, `font`, and `justify` parameters, respectively. If these parameters are set to `True`, then the corresponding columns must be present in the input file(s) and the columns must be in the order mentioned above.

- **x/y** (*float or 1-D arrays*) – The x and y coordinates, or an array of x and y coordinates to plot the text.
- **position** (Optional[Literal['TL', 'TC', 'TR', 'ML', 'MC', 'MR', 'BL', 'BC', 'BR']], default: None) – Set 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:
  - Vertical: **T**(op), **M**(iddle), **B**(ottom)
  - Horizontal: **L**(eft), **C**(entre), **R**(ight)

  For example, `position="TL"` plots the text at the Top Left corner of the map.
- **text** (str | Sequence[str] | ndarray | StringArray | None, default: None) – The text string, or an array of strings to plot on the figure.
- **angle** (*float, str, bool or list*) – 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, bool or list of str*) – 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** (Union[bool, None, Literal['TL', 'TC', 'TR', 'ML', 'MC', 'MR', 'BL', 'BC', 'BR'], Sequence[Literal['TL', 'TC', 'TR', 'ML', 'MC', 'MR', 'BL', 'BC', 'BR']]], default: None) – Set the alignment which refers to the part of the text string that will be mapped onto the (x, y) point. Choose a two-letter combination of **L**, **C**, **R** (for left, center, or right) and **T**, **M**, **B** (for top, middle, or bottom). E.g., **BL** for bottom 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*|*scale*. 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 centimeters, **i** for inches, or **p** for points; 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 text box when using `fill` and/or `pen`. Append lowercase **o** to get a straight rectangle [Default is **o**]. Append uppercase **O** to get a rounded rectangle. In paragraph mode (*paragraph*) you can also append lowercase **c** to get a concave rectangle or append uppercase **C** to get a convex rectangle.
- **fill** (*str*) – Set color for filling text boxes [Default is no fill].
- **offset** (*str*) – [**j**|**J**]*dx*[/*dy*][**+v**[*pen*]]. Offset the text from the projected (x, y) point by *dx*/*dy* [Default is `"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*) – Set the pen used to draw a rectangle around the text string (see `clearance`) [Default is `"0.25p,black,solid"`].

- **no_clip** (*bool*) – Do **not** clip text at the frame boundaries [Default is `False`].
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **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/6.5/gmt.html#aspatial-full.
- **panel** (*bool, 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/6.5/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.
- **use_word** (*int*) – Select a specific word from the trailing text, with the first word being 0 [Default is the entire trailing text]. No numerical columns can be specified.
- **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/6.5/gmt.html#perspective-full.
- **transparency** (*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 1-D array to set varying transparency for texts, but this option is only valid if using `x`/`y` and `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/6.5/gmt.html#w-full.

Examples using `pygmt.Figure.text`

pygmt.Figure.tilemap

`Figure.`**`tilemap`**(*region*, *zoom='auto'*, *source=None*, *lonlat=True*, *wait=0*, *max_retries=2*, *zoom_adjust=None*, ***kwargs*)

Plot an XYZ tile map.

This method loads XYZ tile maps from a tile server or local file using *`pygmt.datasets.load_tile_map`* into a georeferenced form, and plots the tiles as a basemap or overlay using *`pygmt.Figure.grdimage`*.

Note: By default, standard web map tiles served in a Spherical Mercator (EPSG:3857) Cartesian format will be reprojected to a geographic coordinate reference system (OGC:CRS84) and plotted with longitude/latitude bounds when `lonlat=True`. If reprojection is not desired, please set `lonlat=False` and provide Spherical Mercator (EPSG:3857) coordinates to the `region` parameter.

Aliases:

- B = frame
- E = dpi
- I = shading
- J = projection
- M = monochrome
- N = no_clip
- Q = nan_transparent
- V = verbose
- c = panel
- p = perspective
- t = transparency

Parameters

- **`region`** (`list`) – The bounding box of the map in the form of a list [*xmin*, *xmax*, *ymin*, *ymax*]. These coordinates should be in longitude/latitude if `lonlat=True` or Spherical Mercator (EPSG:3857) if `lonlat=False`.
- **`zoom`** (`Union[int, Literal['auto']]`, default: `'auto'`) – Level of detail. Higher levels (e.g. `22`) mean a zoom level closer to the Earth's surface, with more tiles covering a smaller geographical area and thus more detail. Lower levels (e.g. `0`) mean a zoom level further from the Earth's surface, with less tiles covering a larger geographical area and thus less detail. Default is `"auto"` to automatically determine the zoom level based on the bounding box region extent.

  Note

  The maximum possible zoom level may be smaller than `22`, and depends on what is supported by the chosen web tile provider source.

- **`source`** (`TileProvider | str | None`, default: `None`) – The tile source: web tile provider or path to a local file. Provide either:

  - A web tile provider in the form of a `xyzservices.TileProvider` object. See Contextily providers for a list of tile providers. Default is `xyzservices.providers.OpenStreetMap.HOT`, i.e. OpenStreetMap Humanitarian web tiles.
  - A web tile provider in the form of a URL. The placeholders for the XYZ in the URL need to be `{x}`, `{y}`, `{z}`, respectively. E.g. `https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png`.
  - A local file path. The file is read with rasterio and all bands are loaded into the basemap. See Working with local files.

Important

Tiles are assumed to be in the Spherical Mercator projection (EPSG:3857).

- **lonlat** (`bool`, default: `True`) – If `False`, coordinates in `region` are assumed to be Spherical Mercator as opposed to longitude/latitude.
- **wait** (`int`, default: `0`) – If the tile API is rate-limited, the number of seconds to wait between a failed request and the next try.
- **max_retries** (`int`, default: `2`) – Total number of rejected requests allowed before contextily will stop trying to fetch more tiles from a rate-limited API.
- **zoom_adjust** (`int` | `None`, default: `None`) – The amount to adjust a chosen zoom level if it is chosen automatically. Values outside of -1 to 1 are not recommended as they can lead to slow execution.
- **kwargs** (*dict*) – Extra keyword arguments to pass to *pygmt.Figure.grdimage*.

Examples using `pygmt.Figure.tilemap`

pygmt.Figure.timestamp

`Figure.`**`timestamp`**(*text=None*, *label=None*, *justify='BL'*, *offset=('-54p', '-54p')*, *font='Helvetica,black'*, *timefmt='%Y %b %d %H:%M:%S'*)

Plot the GMT timestamp logo.

Add the GMT timestamp logo with an optional label at the bottom-left corner of a plot with an offset of `("-54p", "-54p")`. The timestamp will be in the locale set by the environment variable *TZ* (generally local time but can be changed via `os.environ["TZ"]`) and its format is controlled by the `timefmt` parameter. It can also be replaced with any custom text string using the `text` parameter.

Parameters

- **text** (`str` | `None`, default: `None`) – If `None`, the current UNIX timestamp is shown in the GMT timestamp logo. Set this parameter to replace the UNIX timestamp with a custom text string instead. The text must be no longer than 64 characters.
- **label** (`str` | `None`, default: `None`) – The text string shown after the GMT timestamp logo.
- **justify** (`Literal['TL', 'TC', 'TR', 'ML', 'MC', 'MR', 'BL', 'BC', 'BR']`, default: `'BL'`) – Justification of the timestamp box relative to the plot's bottom-left corner (i.e., the plot origin). Give a two-character code that is a combination of a horizontal (**L**(eft), **C**(enter), or **R**(ight)) and a vertical (**T**(op), **M**(iddle), or **B**(ottom)) code. For example, `justify="TL"` means choosing the **T**op **L**eft point of the timestamp as the anchor point.

- **offset** (float | str | Sequence[float | str], default: ('-54p', '-54p')) – *offset* or (*offset_x*, *offset_y*). Offset the anchor point of the timestamp box by *offset_x* and *offset_y*. If a single value *offset* is given, *offset_y* = *offset_x* = *offset*.
- **font** (str, default: 'Helvetica,black') – Font of the timestamp and the optional label. Since the GMT logo has a fixed height, the font sizes are fixed to be 8-point for the timestamp and 7-point for the label. The parameter can't change the font color for GMT<=6.4.0, only the font style.
- **timefmt** (str, default: '%Y %b %d %H:%M:%S') – Format string for the UNIX timestamp. The format string is parsed by the C function strftime, so that virtually any text can be used (even not containing any time information).

Examples

Plot the GMT timestamp logo.

```
>>> import pygmt
>>> fig = pygmt.Figure()
>>> fig.timestamp()
>>> fig.show()
```

Plot the GMT timestamp logo with a custom label.

```
>>> fig = pygmt.Figure()
>>> fig.timestamp(label="Powered by PyGMT")
>>> fig.show()
```

Examples using pygmt.Figure.timestamp

pygmt.Figure.velo

Figure.**velo**(*data=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/6.5/supplements/geodesy/velo.html

Aliases:

- A = vector
- B = frame
- C = cmap
- D = rescale
- E = uncertaintyfill
- G = fill
- H = scale
- I = shading
- J = projection
- L = line
- N = no_clip
- R = region
- S = spec
- V = verbose
- W = pen
- Z = zvalue
- c = panel
- d = nodata
- e = find
- h = header
- i = incols
- p = perspective
- t = transparency

Parameters

- **data** (str | PathLike | dict | ndarray | DataFrame | Dataset | GeoDataFrame | None, default: None) – Pass in either a file name to an ASCII data table, a 2-D numpy.ndarray, a pandas.DataFrame, an xarray.Dataset made up of 1-D 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*) – Select 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 parameter and the arrow-head can be colored via fill. The ellipse will be filled with the color or pattern specified by the uncertaintyfill parameter [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 parameter and the arrow-head can be colored via fill. The ellipse will be filled with the color or pattern specified by the uncertaintyfill parameter [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 `fill` to set the fill color or pattern for the wedge, and `uncertaintyfill` to set the color or pattern 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*|*scale*. 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/6.5/supplements/geodesy/velo.html#vector-attributes for specifying additional attributes.
- **frame** (*bool, 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.
- **uncertaintyfill** (*str*) – Set color or pattern for filling uncertainty wedges (`spec="w"`) or velocity error ellipses (`spec="e"` or `spec="r"`). If `uncertaintyfill` 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` [Default is magnitude error]. More details at https://docs.generic-mapping-tools.org/6.5/reference/features.html#gfill-attrib.
- **fill** (*str*) – Set color or pattern for filling symbols [Default is no fill]. **Note**: Using `cmap` (and optionally `zvalue`) will update the symbol fill color based on the selected measure in `zvalue` [Default is magnitude]. More details at https://docs.generic-mapping-tools.org/6.5/reference/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*) – Do **not** skip symbols that fall outside the frame boundaries [Default is `False`, i.e., plot symbols inside the frame boundaries only].
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **pen** (*str*) – [*pen*][**+c**[**f**|**l**]]. Set pen attributes for velocity arrows, ellipse circumference and fault plane edges [Default is `"0.25p,black,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.
- **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, 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, `nodata="-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 1-D 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 *1-D 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/6.5/gmt.html#perspective-full.
- **transparency** (*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`

pygmt.Figure.vlines

`Figure.`**vlines**(*x*, *ymin=None*, *ymax=None*, *pen=None*, *label=None*, *no_clip=False*, *perspective=None*)

Plot one or multiple vertical line(s).

This method is a high-level wrapper around *pygmt.Figure.plot* that focuses on plotting vertical lines at X-coordinates specified by the `x` parameter. The `x` parameter can be a single value (for a single vertical line) or a sequence of values (for multiple vertical lines).

By default, the Y-coordinates of the start and end points of the lines are set to be the Y-limits of the current plot, but this can be overridden by specifying the `ymin` and `ymax` parameters. `ymin` and `ymax` can be either a single value or a sequence of values. If a single value is provided, it is applied to all lines. If a sequence is provided, the length of `ymin` and `ymax` must match the length of `x`.

The term "vertical" lines can be interpreted differently in different coordinate systems:

- **Cartesian**: lines are plotted as straight lines.
- **Polar**: lines are plotted as straight lines along a constant azimuth.
- **Geographic**: lines are plotted as arcs along meridians (i.e., constant longitude).

Parameters

- **x** (float | Sequence[float]) – X-coordinates to plot the lines. It can be a single value (for a single line) or a sequence of values (for multiple lines).
- **ymin/ymax** – Y-coordinates of the start/end point(s) of the line(s). If None, defaults to the Y-limits of the current plot. ymin and ymax can either be a single value or a sequence of values. If a single value is provided, it is applied to all lines. If a sequence is provided, the length of ymin and ymax must match the length of x.
- **pen** (str | None, default: None) – Pen attributes for the line(s), in the format of *width,color,style*.
- **label** (str | None, default: None) – Label for the line(s), to be displayed in the legend.
- **no_clip** (bool, default: False) – If True, do not clip lines outside the plot region. Only makes sense in the Cartesian coordinate system.
- **perspective** (str | bool | None, default: None) – Select perspective view and set the azimuth and elevation angle of the viewpoint. Refer to *pygmt.Figure.plot* for details.

Examples

```
>>> import pygmt
>>> fig = pygmt.Figure()
>>> fig.basemap(region=[0, 10, 0, 10], projection="X10c/10c", frame=True)
>>> fig.vlines(x=1, pen="1p,black", label="Line at x=1")
>>> fig.vlines(x=2, ymin=2, ymax=8, pen="1p,red,-", label="Line at x=2")
>>> fig.vlines(x=[3, 4], ymin=3, ymax=7, pen="1p,black,.", label="Lines at x=3,4")
>>> fig.vlines(x=[5, 6], ymin=4, ymax=9, pen="1p,red", label="Lines at x=5,6")
>>> fig.vlines(
...     x=[7, 8], ymin=[0, 1], ymax=[7, 8], pen="1p,blue", label="Lines at x=7,8"
... )
>>> fig.legend()
>>> fig.show()
```

Examples using `pygmt.Figure.vlines`

pygmt.Figure.wiggle

Figure.**wiggle**(*data=None*, *x=None*, *y=None*, *z=None*, *fillpositive=None*, *fillnegative=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, and z.

Full option list at https://docs.generic-mapping-tools.org/6.5/wiggle.html

Aliases:

- B = frame
- D = position
- J = projection
- R = region
- T = track
- V = verbose
- W = pen
- 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** (*1-D arrays*) – The arrays of x and y coordinates and z data points.
- **data** (str | PathLike | dict | ndarray | DataFrame | Dataset | GeoDataFrame | None, default: None) – Pass in either a file name to an ASCII data table, a 2-D numpy.ndarray, a pandas.DataFrame, an xarray.Dataset made up of 1-D 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*|*scale*. Select map *projection*.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **scale** (*str or float*) – Give anomaly scale in data-units/distance-unit. Append **c**, **i**, or **p** to indicate the distance unit (centimeters, inches, or points); if no unit is given we use the default unit that is controlled by PROJ_LENGTH_UNIT.
- **frame** (*bool, 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*]]. Define the reference point on the map for the vertical scale bar.
- **fillpositive** (*str*) – Set color or pattern for filling positive wiggles [Default is no fill].
- **fillnegative** (*str*) – Set color or pattern for filling negative wiggles [Default is no fill].
- **track** (*str*) – Draw track [Default is no track]. Append pen attributes to use [Default is "0.25p,black,solid"].
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **pen** (*str*) – Specify outline pen attributes [Default is no outline].
- **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/6.5/gmt.html#bi-full.

- **panel** (*bool, 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, `nodata="-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/6.5/gmt.html#f-full.
- **gap** (*str or list*) – **x**|**y**|**z**|**d**|**X**|**Y**|**D***gap*[**u**][**+a**][**+c***col*][**+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.
  - **x**|**X**: define a gap when there is a large enough change in the x coordinates (uppercase to use projected coordinates).
  - **y**|**Y**: define a gap when there is a large enough change in the y coordinates (uppercase to use projected coordinates).
  - **d**|**D**: define a gap when there is a large enough distance between coordinates (uppercase to use projected coordinates).
  - **z**: define a gap when there is a large enough change in the z data. Use **+c***col* to change the z data column [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**(egrees), **m**(inutes), and **s**(econds) , or (m)**e**(ters), **f**(eet), **k**(ilometers), **M**(iles), or **n**(autical miles) [Default is (m)**e**(ters)].
  - For projected data (**X**|**Y**|**D**), the unit may be **i**(nches), **c**(entimeters), or **p**(oints).

  Append modifier **+a** to specify that *all* the criteria must be met [default imposes breaks if any one criterion is met].

  One of the following modifiers can be appended:
  - **+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 1-D 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 *1-D 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/6.5/gmt.html#perspective-full.
- **transparency** (*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/6.5/gmt.html#w-full.

Examples using `pygmt.Figure.wiggle`

Examples using `pygmt.Figure`

8.1.2 Plotting map elements

`Figure.basemap`(*[, region, projection, ...])	Plot base maps and frames.
`Figure.coast`(*[, area_thresh, frame, lakes, ...])	Plot continents, countries, shorelines, rivers, and borders.
`Figure.colorbar`(*[, frame, cmap, position, ...])	Plot gray scale or color scale bar.
`Figure.hlines`(y[, xmin, xmax, pen, label, ...])	Plot one or multiple horizontal line(s).
`Figure.inset`(*[, position, box, projection, ...])	Manage figure inset setup and completion.
`Figure.legend`([spec, position, box])	Plot a legend.
`Figure.logo`(*[, region, projection, ...])	Plot the GMT logo.
`Figure.solar`([terminator, terminator_datetime])	Plot day-night terminators and other sunlight parameters.
`Figure.text`([textfiles, x, y, position, ...])	Plot or typeset text.
`Figure.timestamp`([text, label, justify, ...])	Plot the GMT timestamp logo.
`Figure.vlines`(x[, ymin, ymax, pen, label, ...])	Plot one or multiple vertical line(s).

8.1.3 Plotting tabular data

`Figure.contour`([data, x, y, z])	Contour table data by direct triangulation.
`Figure.histogram`(data, **kwargs)	Calculate and plot histograms.
`Figure.meca`(spec, scale[, convention, ...])	Plot focal mechanisms.
`Figure.plot`([data, x, y, size, symbol, ...])	Plot lines, polygons, and symbols in 2-D.
`Figure.plot3d`([data, x, y, z, size, symbol, ...])	Plot lines, polygons, and symbols in 3-D.
`Figure.rose`([data, length, azimuth])	Plot a polar histogram (rose, sector, windrose diagrams).
`Figure.ternary`(data[, alabel, blabel, clabel])	Plot data on ternary diagrams.
`Figure.velo`([data])	Plot velocity vectors, crosses, anisotropy bars, and wedges.
`Figure.wiggle`([data, x, y, z, fillpositive, ...])	Plot z=f(x,y) anomalies along tracks.

8.1.4 Plotting raster data

`Figure.grdcontour`(grid, **kwargs)	Make contour map using a grid.
`Figure.grdimage`(grid, **kwargs)	Project and plot grids or images.
`Figure.grdview`(grid, **kwargs)	Create 3-D perspective image or surface mesh from a grid.
`Figure.image`(imagefile, **kwargs)	Plot raster or EPS images.
`Figure.tilemap`(region[, zoom, source, ...])	Plot an XYZ tile map.

8.1.5 Configuring layout

`Figure.set_panel`([panel, fixedlabel, ...])	Set the current subplot panel to plot on.
`Figure.shift_origin`([xshift, yshift])	Shift the plot origin in x and/or y directions.
`Figure.subplot`([nrows, ncols, figsize, ...])	Manage figure subplot configuration and selection.

8.1.6 Saving and displaying the figure

Figure.savefig(fname[, transparent, crop, ...])	Save the figure to an image file.
Figure.show([method, dpi, width, waiting])	Display a preview of the figure.
Figure.psconvert(*[, crop, gs_option, dpi, ...])	Convert [E]PS file(s) to other formats using Ghostscript.

8.1.7 Configuring the display settings

The following function is provided directly through the `pygmt` top level package.

set_display([method])	Set the display method when calling *pygmt.Figure.show*.

pygmt.set_display

pygmt.**set_display**(*method=None*)

Set the display method when calling *pygmt.Figure.show*.

Parameters

method (Literal['external', 'notebook', 'none', None], default: None) – The method to display an image preview. Choose from:

- "external": External PDF preview using the default PDF viewer
- "notebook": Inline PNG preview in the current notebook
- "none": Disable image preview
- None: Reset to the default display method, which is either "external" in Python consoles or "notebook" in Jupyter notebooks.

Return type

None

Examples

Let's assume that you're using a Jupyter Notebook:

```
>>> import pygmt
>>> fig = pygmt.Figure()
>>> fig.basemap(region=[0, 10, 0, 10], projection="X10c/5c", frame=True)
>>> fig.show()  # Will display a PNG image in the current notebook
>>>
>>> # Set the display method to "external"
>>> pygmt.set_display(method="external")
>>> fig.show()  # Will display a PDF image using the default PDF viewer
>>>
>>> # Set the display method to "none"
>>> pygmt.set_display(method="none")
>>> fig.show()  # Will not show any image
>>>
>>> # Reset to the default display method
```

(continues on next page)

(continued from previous page)

```
>>> pygmt.set_display(method=None)
>>> fig.show()  # Again, will show a PNG image in the current notebook
```

8.1.8 Color palette table generation

The following functions are provided directly through the `pygmt` top level package.

grd2cpt(grid, **kwargs)	Make linear or histogram-equalized color palette table from grid.
makecpt(*[, transparency, cmap, background, ...])	Make GMT color palette tables.

pygmt.grd2cpt

pygmt.**grd2cpt**(*grid*, ***kwargs*)

Make linear or histogram-equalized color palette table from grid.

This function will help you to make static color palette tables (CPTs). By default, the CPT will be saved as the current CPT of the session, figure, subplot, panel, or inset depending on which level *pygmt.grd2cpt* is called (for details on how GMT modern mode maintains different levels of colormaps please see https://docs.generic-mapping-tools.org/6.5/reference/features.html#gmt-modern-mode-hierarchical-levels). You can use `output` to save the CPT 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 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 *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` parameter will be used.

Full option list at https://docs.generic-mapping-tools.org/6.5/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` | `PathLike` | `DataArray`) – Name of the input grid file or the grid loaded as a `xarray.DataArray` object.

For reading a specific grid file format or applying basic data operations, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **transparency** (*int or float or str*) – Set a constant level of transparency (0-100) for all color slices. Append **+a** to also affect the foreground, background, and NaN colors [Default is no transparency, i.e., `0` (opaque)].
- **cmap** (*str*) – Select 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/6.5/reference/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** (*str*) – [**R**|**r**|**h**|**c**][**+c**[*label*|*start*[**-**]]]. 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). Additionally append **-** to build ranges *start-start+1* as labels instead.
- **nlevels** (*bool, 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*]. Define 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/6.5/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/6.5/reference/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 current CPT of the session, figure, subplot, panel, or inset depending on which level *pygmt.grd2cpt* is called.
- **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/6.5/reference/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. 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 [*Full usage*].

Example

```
>>> import pygmt
>>> # load the 30 arc-minutes grid with "gridline" registration
>>> grid = pygmt.datasets.load_earth_relief("30m", registration="gridline")
>>> # create a plot
>>> fig = pygmt.Figure()
>>> # create a CPT from the grid object with grd2cpt
>>> pygmt.grd2cpt(grid=grid)
>>> # plot the grid object, the CPT will be automatically used
>>> fig.grdimage(grid=grid)
>>> # show the plot
>>> fig.show()
```

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 function will help you to make static color palette tables (CPTs). By default, the CPT will be saved as the current CPT of the session, figure, subplot, panel, or inset depending on which level *pygmt.makecpt* is called (for details on how GMT modern mode maintains different levels of colormaps please see https://docs.generic-mapping-tools.org/6.5/reference/features.html#gmt-modern-mode-hierarchical-levels). You can use `output` to save the CPT 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 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 *pygmt.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 will be used.

Full option list at https://docs.generic-mapping-tools.org/6.5/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*) – Set a constant level of transparency (0-100) for all color slices. Append **+a** to also affect the foreground, background, and NaN colors [Default is no transparency, i.e., `0` (opaque)].
- **cmap** (*str*) – Select 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/6.5/reference/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** (*str*) – [**R**|**r**|**h**|**c**][**+c**[*label*|*start*[**-**]]]. 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). Additionally append **-** to build ranges *start-start+1* as labels instead.
- **series** (*list or str*) – [*min/max/inc*[**+b**|**l**|**n**]|*file*|*list*]. Define 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/6.5/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/6.5/reference/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 current CPT of the session, figure, subplot, panel, or inset depending on which level *pygmt.makecpt* is called.
- **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/6.5/reference/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. 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 [*Full usage*].
- **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`

8.2 Data Processing

8.2.1 Operations on tabular data

binstats(data[, outgrid])	Bin spatial data and determine statistics per bin.
blockmean([data, x, y, z, output_type, outfile])	Block average (x, y, z) data tables by mean estimation.
blockmedian([data, x, y, z, output_type, ...])	Block average (x, y, z) data tables by median estimation.
blockmode([data, x, y, z, output_type, outfile])	Block average (x, y, z) data tables by mode estimation.
filter1d(data[, output_type, outfile])	Time domain filtering of 1-D data tables.
nearneighbor([data, x, y, z, outgrid])	Grid table data using a "Nearest neighbor" algorithm.
project([data, x, y, z, output_type, outfile])	Project data onto lines or great circles, or generate tracks.
select([data, output_type, outfile])	Select data table subsets based on multiple spatial criteria.
sph2grd(data[, outgrid])	Compute grid from spherical harmonic coefficients.
sphdistance([data, x, y, outgrid])	Create Voronoi distance, node, or natural nearest-neighbor grid on a sphere.
sphinterpolate(data[, outgrid])	Spherical gridding in tension of data on a sphere.
surface([data, x, y, z, outgrid])	Grid table data using adjustable tension continuous curvature splines.
triangulate()	Delaunay triangulation or Voronoi partitioning and gridding of Cartesian data.
triangulate.regular_grid([data, x, y, z, ...])	Delaunay triangle based gridding of Cartesian data.
triangulate.delaunay_triples([data, x, y, ...])	Delaunay triangle based gridding of Cartesian data.
xyz2grd([data, x, y, z, outgrid])	Convert data table to a grid.

pygmt.binstats

pygmt.**binstats**(*data*, *outgrid=None*, ***kwargs*)

Bin spatial data and determine statistics per bin.

Reads arbitrarily located (x,y[,z][,w]) points (2-4 columns) from `data` and for each node in the specified grid layout determines which points are within the given radius. These points are then used in the calculation of the

specified statistic. The results may be presented as is or may be normalized by the circle area to perhaps give density estimates.

Full option list at https://docs.generic-mapping-tools.org/6.5/gmtbinstats.html

Aliases:

- C = statistic
- E = empty
- I = spacing
- N = normalize
- R = region
- S = search_radius
- V = verbose
- W = weight
- a = aspatial
- b = binary
- h = header
- i = incols
- r = registration

Parameters

- **data** (`str` | `PathLike` | `dict` | `ndarray` | `DataFrame` | `Dataset` | `GeoDataFrame`) – A file name of an ASCII data table or a 2-D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1-D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **outgrid** (`str` | `PathLike` | `None`, default: `None`) – Name of the output netCDF grid file. If not specified, will return an `xarray.DataArray` object. For writing a specific grid file format or applying basic data operations to the output grid, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.
- **statistic** (*str*) – **a**|**d**|**g**|**i**|**l**|**L**|**m**|**n**|**o**|**p**|**q**[*quant*]|**r**|**s**|**u**|**U**|**z**. Choose the statistic that will be computed per node based on the points that are within *radius* distance of the node. Select one of:
  - **a**: mean (average)
  - **d**: median absolute deviation (MAD)
  - **g**: full (max-min) range
  - **i**: 25-75% interquartile range
  - **l**: minimum (low)
  - **L**: minimum of positive values only
  - **m**: median
  - **n**: number of values
  - **o**: LMS scale
  - **p**: mode (maximum likelihood)
  - **q**: selected quantile (append desired quantile in 0-100% range [50])
  - **r**: root mean square (RMS)
  - **s**: standard deviation
  - **u**: maximum (upper)
  - **U**: maximum of negative values only
  - **z**: sum
- **empty** (*float*) – Set the value assigned to empty nodes [Default is NaN].
- **normalize** (*bool*) – Normalize the resulting grid values by the area represented by the search *radius* [Default is no normalization].

- **search_radius** (*float or str*) – Set the *search_radius* that determines which data points are considered close to a node. Append the distance unit. Not compatible with `tiling`.
- **weight** (*str*) – Input data have an extra column containing observation point weight. If weights are given then weighted statistical quantities will be computed while the count will be the sum of the weights instead of number of points. If the weights are actually uncertainties (one sigma) then append **+s** and weight = 1/sigma.
- **spacing** (*float, str, or list*) – *x_inc*[**+e**|**n**][/*y_inc*[**+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 [*Full usage*].
- **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/6.5/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/6.5/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 1-D 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 *1-D 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)].

Return type

`DataArray` | `None`

Returns

ret – 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 the file set by `outgrid`)

pygmt.blockmean

`pygmt.`**`blockmean`**(*data=None*, *x=None*, *y=None*, *z=None*, *output_type='pandas'*, *outfile=None*, ***kwargs*)

Block average (x, y, z) data tables by mean estimation.

Reads arbitrarily located (x, y, z) triplets [or optionally weighted quadruplets (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, (x, y, z) 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/6.5/blockmean.html

Aliases:

- I = spacing
- R = region
- S = summary
- 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` | `PathLike` | `dict` | `ndarray` | `DataFrame` | `Dataset` | `GeoDataFrame` | `None`, default: `None`) – Pass in (x, y, z) or (longitude, latitude, elevation) values by providing a file name to an ASCII data table, a 2-D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1-D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **`x/y/z`** (*`1-D arrays`*) – Arrays of x and y coordinates and values z of the data points.
- **`output_type`** (`Literal['pandas', 'numpy', 'file']`, default: `'pandas'`) – Desired output type of the result data.
  - `pandas` will return a `pandas.DataFrame` object.
  - `numpy` will return a `numpy.ndarray` object.
  - `file` will save the result to the file specified by the `outfile` parameter.
- **`outfile`** (`str` | `PathLike` | `None`, default: `None`) – File name for saving the result data. Required if `output_type="file"`. If specified, `output_type` will be forced to be `"file"`.
- **`spacing`** (*`float, str, or list`*) – *x_inc*[**+e**|**n**][/*y_inc*[**+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.

- **summary** (*str*) – [**m**|**n**|**s**|**w**]. Type of summary values calculated by blockmean.
  - **m**: reports mean value [Default]
  - **n**: report the number of input points inside each block
  - **s**: report the sum of all z-values inside a block
  - **w**: report the sum of weights
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **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/6.5/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/6.5/gmt.html#bi-full.
- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `nodata="-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 1-D 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 *1-D 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/6.5/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 1-D 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 *1-D 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/6.5/gmt.html#w-full.

Return type
`DataFrame` | `ndarray` | `None`

Returns

ret – 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`)

Example

```
>>> import pygmt
>>> # Load a table of ship observations of bathymetry off Baja California
>>> data = pygmt.datasets.load_sample_data(name="bathymetry")
>>> # Calculate block mean values within 5 by 5 arc-minute bins
>>> data_bmean = pygmt.blockmean(data=data, region=[245, 255, 20, 30], spacing="5m
↪")
```

Examples using `pygmt.blockmean`

pygmt.blockmedian

`pygmt.`**`blockmedian`**(*data=None*, *x=None*, *y=None*, *z=None*, *output_type='pandas'*, *outfile=None*, ***kwargs*)

Block average (x, y, z) data tables by median estimation.

Reads arbitrarily located (x, y, z) triplets [or optionally weighted quadruplets (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, (x, y, z) 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/6.5/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` | `PathLike` | `dict` | `ndarray` | `DataFrame` | `Dataset` | `GeoDataFrame` | `None`, default: `None`) – Pass in (x, y, z) or (longitude, latitude, elevation) values by providing a file name to an ASCII data table, a 2-D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1-D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **x/y/z** (*1-D arrays*) – Arrays of x and y coordinates and values z of the data points.
- **output_type** (`Literal['pandas', 'numpy', 'file']`, default: `'pandas'`) – Desired output type of the result data.
  - `pandas` will return a `pandas.DataFrame` object.
  - `numpy` will return a `numpy.ndarray` object.
  - `file` will save the result to the file specified by the `outfile` parameter.
- **outfile** (`str` | `PathLike` | `None`, default: `None`) – File name for saving the result data. Required if `output_type="file"`. If specified, `output_type` will be forced to be `"file"`.
- **spacing** (*float, str, or list*) – *x_inc*[**+e**|**n**][/*y_inc*[**+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 [*Full usage*].

- **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/6.5/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/6.5/gmt.html#bi-full.
- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `nodata="-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/6.5/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 1-D 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 *1-D 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 1-D 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 *1-D 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/6.5/gmt.html#w-full.

Return type
`DataFrame` | `ndarray` | `None`

Returns

ret – Return type depends on `outfile` and `output_type`:

- `None` if `outfile` is set (output will be stored in the file set by `outfile`)
- `pandas.DataFrame` or `numpy.ndarray` if `outfile` is not set (depends on `output_type`)

Example

```
>>> import pygmt
>>> # Load a table of ship observations of bathymetry off Baja California
>>> data = pygmt.datasets.load_sample_data(name="bathymetry")
>>> # Calculate block median values within 5 by 5 arc-minute bins
>>> data_bmedian = pygmt.blockmedian(
...     data=data, region=[245, 255, 20, 30], spacing="5m"
... )
```

pygmt.blockmode

`pygmt.`**`blockmode`**(*data=None*, *x=None*, *y=None*, *z=None*, *output_type='pandas'*, *outfile=None*, ***kwargs*)

Block average (x, y, z) data tables by mode estimation.

Reads arbitrarily located (x, y, z) triplets [or optionally weighted quadruplets (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, (x, y, z) 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/6.5/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` | `PathLike` | `dict` | `ndarray` | `DataFrame` | `Dataset` | `GeoDataFrame` | `None`, default: `None`) – Pass in (x, y, z) or (longitude, latitude, elevation) values by providing a file name to an ASCII data table, a 2-D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1-D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.

- **x/y/z** (*1-D arrays*) – Arrays of x and y coordinates and values z of the data points.
- **output_type** (Literal['pandas', 'numpy', 'file'], default: 'pandas') – Desired output type of the result data.
  - pandas will return a pandas.DataFrame object.
  - numpy will return a numpy.ndarray object.
  - file will save the result to the file specified by the outfile parameter.
- **outfile** (str | PathLike | None, default: None) – File name for saving the result data. Required if output_type="file". If specified, output_type will be forced to be "file".
- **spacing** (*float, str, or list*) – *x_inc*[**+e**|**n**][/*y_inc*[**+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 [*Full usage*].
- **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/6.5/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/6.5/gmt.html#bi-full.

- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `nodata="-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/6.5/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 1-D 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 *1-D 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 1-D 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 *1-D 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/6.5/gmt.html#w-full.

Return type
`DataFrame` | `ndarray` | `None`

Returns

ret – Return type depends on `outfile` and `output_type`:

- `None` if `outfile` is set (output will be stored in the file set by `outfile`)
- `pandas.DataFrame` or `numpy.ndarray` if `outfile` is not set (depends on `output_type`)

Example

```
>>> import pygmt
>>> # Load a table of ship observations of bathymetry off Baja California
>>> data = pygmt.datasets.load_sample_data(name="bathymetry")
>>> # Calculate block mode values within 5 by 5 arc-minute bins
>>> data_bmode = pygmt.blockmode(data=data, region=[245, 255, 20, 30], spacing="5m
↪")
```

pygmt.filter1d

pygmt.**filter1d**(*data*, *output_type='pandas'*, *outfile=None*, ***kwargs*)

Time domain filtering of 1-D data tables.

A general time domain filter for multiple column time series data. The user specifies which column is the time (i.e., the independent variable) via `time_col`. The fastest operation occurs when the input time series are equally spaced and have no gaps or outliers and the special options are not needed. Read a table and output as a `numpy.ndarray`, `pandas.DataFrame`, or ASCII file.

Full option list at https://docs.generic-mapping-tools.org/6.5/filter1d.html

Aliases:

- E = end
- F = filter_type
- N = time_col

Parameters

- **output_type** (`Literal['pandas', 'numpy', 'file']`, default: `'pandas'`) – Desired output type of the result data.
  - `pandas` will return a `pandas.DataFrame` object.
  - `numpy` will return a `numpy.ndarray` object.
  - `file` will save the result to the file specified by the `outfile` parameter.
- **outfile** (`str` | `PathLike` | `None`, default: `None`) – File name for saving the result data. Required if `output_type="file"`. If specified, `output_type` will be forced to be `"file"`.
- **filter_type** (*str*) – **type***width*[**+h**]. Set the filter **type**. Choose among convolution and non-convolution filters. Append the filter code followed by the full filter *width* in same units as time column. By default, this performs a low-pass filtering; append **+h** to select high-pass filtering. Some filters allow for optional arguments and a modifier.

  Available convolution filter types are:
  - **b**: boxcar. All weights are equal.
  - **c**: cosine arch. Weights follow a cosine arch curve.
  - **g**: Gaussian. Weights are given by the Gaussian function.
  - **f**: custom. Instead of *width* give name of a one-column file with your own weight coefficients.

  Non-convolution filter types are:
  - **m**: median. Returns median value.
  - **p**: maximum likelihood probability (a mode estimator). Return modal value. If more than one mode is found we return their average value. Append **+l** or **+u** if you rather want to return the lowermost or uppermost of the modal values.

- **l**: lower (absolute). Return the minimum of all values.
- **L**: lower. Return minimum of all positive values only.
- **u**: upper (absolute). Return maximum of all values.
- **U**: upper. Return maximum of all negative values only.

Uppercase type **B**, **C**, **G**, **M**, **P**, **F** will use robust filter versions: i.e., replace outliers (2.5 L1 scale off median, using 1.4826 * median absolute deviation [MAD]) with median during filtering.

In the case of **L**|**U** it is possible that no data passes the initial sign test; in that case the filter will return 0.0. Apart from custom coefficients (**f**), the other filters may accept variable filter widths by passing *width* as a two-column time-series file with filter widths in the second column. The filter-width file does not need to be co-registered with the data as we obtain the required filter width at each output location via interpolation. For multi-segment data files the filter file must either have the same number of segments or just a single segment to be used for all data segments.

- **end** (*bool*) – Include ends of time series in output. The default [False] loses half the filter-width of data at each end.
- **time_col** (*int*) – Indicate which column contains the independent variable (time). The left-most column is 0, while the right-most is (*n_cols* - 1) [Default is 0].

Return type
DataFrame | ndarray | None

Returns

ret – Return type depends on `outfile` and `output_type`:

- `None` if `outfile` is set (output will be stored in the file set by `outfile`)
- `pandas.DataFrame` or `numpy.ndarray` if `outfile` is not set (depends on `output_type`)

pygmt.nearneighbor

pygmt.**nearneighbor**(*data=None*, *x=None*, *y=None*, *z=None*, *outgrid=None*, ***kwargs*)

Grid table data using a "Nearest neighbor" algorithm.

nearneighbor reads arbitrarily located (*x*, *y*, *z*[, *w*]) triplets [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 $w_i = 1$.

Takes a matrix, (x, y, z) 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/6.5/nearneighbor.html

Aliases:

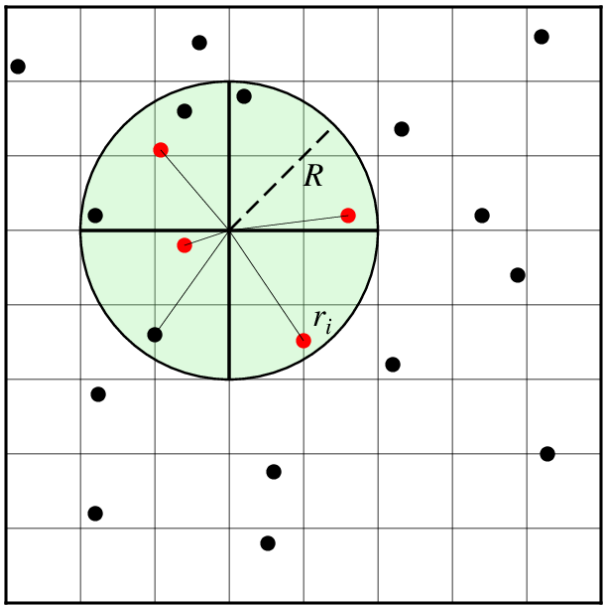


Fig. 1: Search geometry includes the search radius (R) which limits the points considered and the number of sectors (here 4), which restricts how points inside the search radius contribute to the value at the node. Only the closest point in each sector (red circles) contribute to the weighted estimate.

- E = empty
- 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` | `PathLike` | `dict` | `ndarray` | `DataFrame` | `Dataset` | `GeoDataFrame` | `None`, default: `None`) – Pass in (x, y, z) or (longitude, latitude, elevation) values by providing a file name to an ASCII data table, a 2-D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1-D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **x/y/z** (*1-D arrays*) – Arrays of x and y coordinates and values z of the data points.
- **spacing** (*float, str, or list*) – *x_inc*[**+e**|**n**][/*y_inc*[**+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*) – Set the search radius that determines which data points are considered close to a node.
- **outgrid** (`str` | `PathLike` | `None`, default: `None`) – Name of the output netCDF grid file. If not specified, will return an `xarray.DataArray` object. For writing a specific grid file format or applying basic data operations to the output grid, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.
- **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 [*Full usage*].
- **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/6.5/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/6.5/gmt.html#bi-full.

- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `nodata="-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/6.5/gmt.html#f-full.

- **gap** (*str or list*) – **x**|**y**|**z**|**d**|**X**|**Y**|**D***gap*[**u**][**+a**][**+c***col*][**+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.

  - **x**|**X**: define a gap when there is a large enough change in the x coordinates (uppercase to use projected coordinates).

  - **y**|**Y**: define a gap when there is a large enough change in the y coordinates (uppercase to use projected coordinates).

  - **d**|**D**: define a gap when there is a large enough distance between coordinates (uppercase to use projected coordinates).

  - **z**: define a gap when there is a large enough change in the z data. Use **+c***col* to change the z data column [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**(egrees), **m**(inutes), and **s**(econds) , or (m)**e**(ters), **f**(eet), **k**(ilometers), **M**(iles), or **n**(autical miles) [Default is (m)**e**(ters)].

  - For projected data (**X**|**Y**|**D**), the unit may be **i**(nches), **c**(entimeters), or **p**(oints).

  Append modifier **+a** to specify that *all* the criteria must be met [default imposes breaks if any one criterion is met].

  One of the following modifiers can be appended:

  - **+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 1-D 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 *1-D 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/6.5/gmt.html#w-full.

Return type

`DataArray` | `None`

Returns

ret – 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 the file set by `outgrid`)

Example

```
>>> import pygmt
>>> # Load a sample dataset of bathymetric x, y, and z values
>>> data = pygmt.datasets.load_sample_data(name="bathymetry")
>>> # Create a new grid with 5 arc-minutes spacing in the designated region
>>> # Set search_radius to only take points within 10 arc-minutes of a node
>>> output = pygmt.nearneighbor(
...     data=data,
...     spacing="5m",
...     region=[245, 255, 20, 30],
...     search_radius="10m",
... )
```

pygmt.project

`pygmt.`**`project`**(*data=None*, *x=None*, *y=None*, *z=None*, *output_type='pandas'*, *outfile=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) triplets 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` parameter to set the pole. Then the equator of the projection is already determined and the `center` parameter 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` parameters. 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` parameter 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 `unit` 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/6.5/mapproject as project is strictly spherical.

Full option list at https://docs.generic-mapping-tools.org/6.5/project.html

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` | `PathLike` | `dict` | `ndarray` | `DataFrame` | `Dataset` | `GeoDataFrame` | `None`, default: `None`) – Pass in (x, y, z) or (longitude, latitude, elevation) values by providing a file name to an ASCII data table, a 2-D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1-D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **output_type** (`Literal['pandas', 'numpy', 'file']`, default: `'pandas'`) – Desired output type of the result data.
  - `pandas` will return a `pandas.DataFrame` object.
  - `numpy` will return a `numpy.ndarray` object.
  - `file` will save the result to the file specified by the `outfile` parameter.
- **outfile** (`str` | `PathLike` | `None`, default: `None`) – File name for saving the result data. Required if `output_type="file"`. If specified, `output_type` will be forced to be `"file"`.
- **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 lowercase. 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` parameter). 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|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 to 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 [*Full usage*].
- **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., 3-D 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*.
- **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/6.5/gmt.html#f-full.

Return type

`DataFrame` | `ndarray` | `None`

Returns

ret – 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`)

Examples using `pygmt.project`

pygmt.select

`pygmt.`**`select`**(*data=None*, *output_type='pandas'*, *outfile=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:

1. are inside a rectangular region (`region` [and `projection`])
2. are within *dist* km of any point in *pointfile* (`dist2pt`)
3. are within *dist* km of any line in *linefile* (`dist2line`)

4. are inside one of the polygons in *polygonfile* (`polygon`)
5. are inside geographical features (based on coastlines)
6. have z-values within a given range
7. are 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` parameter.

Full option list at https://docs.generic-mapping-tools.org/6.5/gmtselect.html

Aliases:

- A = area_thresh
- C = dist2pt
- D = resolution
- F = polygon
- G = gridmask
- I = reverse
- J = projection
- L = dist2line
- 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
- s = skiprows
- w = wrap

Parameters

- **data** (`str` | `PathLike` | `dict` | `ndarray` | `DataFrame` | `Dataset` | `GeoDataFrame` | `None`, default: `None`) – Pass in either a file name to an ASCII data table, a 2-D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1-D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **output_type** (`Literal['pandas', 'numpy', 'file']`, default: `'pandas'`) – Desired output type of the result data.
  - `pandas` will return a `pandas.DataFrame` object.
  - `numpy` will return a `numpy.ndarray` object.
  - `file` will save the result to the file specified by the `outfile` parameter.
- **outfile** (`str` | `PathLike` | `None`, default: `None`) – File name for saving the result data. Required if `output_type="file"`. If specified, `output_type` will be forced to be `"file"`.
- **area_thresh** (*`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)].
- **dist2pt** (*`str`*) – *pointfile*|*lon*/*lat***+d***dist*. Pass all records whose locations are within *dist* of any of the points in the ASCII file *pointfile*. If *dist* is zero, the 3rd column of *pointfile* must have each point's individual radius of influence. If you only have a single point, you can specify *lon*/*lat* instead of *pointfile*. Distances are Cartesian and in user units. Alternatively, if `region` and `projection` are used, the geographic coordinates are projected to map coordinates (in centimeters, inches, meters, or points, as determined by PROJ_LENGTH_UNIT) before Cartesian distances are compared to *dist*.
- **dist2line** (*`str`*) – *linefile***+d***dist*[**+p**]. Pass all records whose locations are within *dist* of any of the line segments in the ASCII multiple-segment file *linefile*. If *dist* is zero, we will scan each sub-header in *linefile* for an embedded **-D***dist* setting that sets each line's individual distance value. Distances are Cartesian and in user units. Alternatively, if `region` and `projection` are used, the geographic coordinates are projected to map coordinates (in centimeters, inches, meters, or points, as determined by PROJ_LENGTH_UNIT) before Cartesian distances are compared to *dist*. Append **+p** to ensure only points whose orthogonal projections onto the

nearest line-segment fall within the segment's endpoints [Default considers points "beyond" the line's endpoints].

- **polygon** (*str*) – *polygonfile*. Pass all records whose locations are within one of the closed polygons in the ASCII multiple-segment file *polygonfile*. For spherical polygons (lon, lat), make sure no consecutive points are separated by 180 degrees or more in longitude.
- **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**]. Reverse 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*|*scale*. 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 [*Full usage*].
- **z_subregion** (*str or list*) – *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; https://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 pass a list of arguments as you have columns to test. **Note**: When more than one `z_subregion` argument is given then the `reverse="z"` 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` argument, 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/6.5/gmt.html#bi-full.
- **nodata** (*str*) – **i|o***nodata*. Substitute specific values with NaN (for tabular data). For example, `nodata="-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/6.5/gmt.html#f-full.
- **gap** (*str or list*) – **x|y|z|d|X|Y|D***gap*[**u**][**+a**][**+c***col*][**+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.
  - **x|X**: define a gap when there is a large enough change in the x coordinates (uppercase to use projected coordinates).
  - **y|Y**: define a gap when there is a large enough change in the y coordinates (uppercase to use projected coordinates).
  - **d|D**: define a gap when there is a large enough distance between coordinates (uppercase to use projected coordinates).
  - **z**: define a gap when there is a large enough change in the z data. Use **+c***col* to change the z data column [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**(egrees), **m**(inutes), and **s**(econds) , or (m)**e**(ters), **f**(eet), **k**(ilometers), **M**(iles), or **n**(autical miles) [Default is (m)**e**(ters)].
  - For projected data (**X**|**Y**|**D**), the unit may be **i**(nches), **c**(entimeters), or **p**(oints).

  Append modifier **+a** to specify that *all* the criteria must be met [default imposes breaks if any one criterion is met].

  One of the following modifiers can be appended:

  - **+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 1-D 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 *1-D 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 1-D 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 *1-D 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/6.5/gmt.html#w-full.

Return type
`DataFrame` | `ndarray` | `None`

Returns

ret – 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`)

Example

```
>>> import pygmt
>>> # Load a table of ship observations of bathymetry off Baja California
>>> ship_data = pygmt.datasets.load_sample_data(name="bathymetry")
>>> # Only return the data points that lie within the region between
```

(continues on next page)

(continued from previous page)

```
>>> # longitudes 246 and 247 and latitudes 20 and 21
>>> out = pygmt.select(data=ship_data, region=[246, 247, 20, 21])
```

pygmt.sph2grd

pygmt.**sph2grd**(*data*, *outgrid=None*, ***kwargs*)

Compute grid from spherical harmonic coefficients.

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/6.5/sph2grd.html

Aliases:

- I = spacing
- R = region
- V = verbose
- b = binary
- h = header
- i = incols
- r = registration
- x = cores

Parameters

- **data** (str | PathLike | dict | ndarray | DataFrame | Dataset | 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 2-D numpy.ndarray, a pandas.DataFrame, an xarray.Dataset made up of 1-D xarray.DataArray data variables, or a geopandas.GeoDataFrame containing the tabular data.
- **outgrid** (str | PathLike | None, default: None) – Name of the output netCDF grid file. If not specified, will return an xarray.DataArray object. For writing a specific grid file format or applying basic data operations to the output grid, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.
- **spacing** (*float, str, or list*) – *x_inc*[**+e**|**n**][/*y_inc*[**+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 [*Full usage*].

- **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/6.5/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 1-D 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 *1-D 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*) – Specify the number of active cores to be used in any OpenMP-enabled multi-threaded algorithms. By default, all available cores are used. Set a positive number *n* to use *n* cores (if too large it will be truncated to the maximum cores available); or set a negative number *-n* to select (all - *n*) cores (or at least 1 if *n* equals or exceeds all).

Return type
DataArray | None

Returns

ret – 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`)

Example

```
>>> import pygmt
>>> # Create a new grid from the remote file "EGM96_to_36.txt",
>>> # set the grid spacing to 1 arc-degree, and the region to global ("g")
>>> new_grid = pygmt.sph2grd(data="@EGM96_to_36.txt", spacing=1, region="g")
```

pygmt.sphdistance

pygmt.**sphdistance**(*data=None*, *x=None*, *y=None*, *outgrid=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 https://docs.generic-mapping-tools.org/6.5/sphdistance.html

Aliases:

- C = single_form
- D = duplicate
- E = quantity
- I = spacing
- L = unit
- N = node_table
- Q = voronoi
- R = region
- V = verbose

Parameters

- **data** (str | PathLike | dict | ndarray | DataFrame | Dataset | GeoDataFrame | None, default: None) – Pass in (x, y) or (longitude, latitude) values by providing a file name to an ASCII data table, a 2-D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset`

made up of 1-D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.

- **x/y** (*1-D arrays*) – Arrays of x and y coordinates.
- **outgrid** (`str` | `PathLike` | `None`, default: `None`) – Name of the output netCDF grid file. If not specified, will return an `xarray.DataArray` object. For writing a specific grid file format or applying basic data operations to the output grid, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.
- **spacing** (*float, str, or list*) – *x_inc*[**+e**|**n**][/*y_inc*[**+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 [*Full usage*].
- **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 degrees), **e** (meters), **f** (feet), **k** (kilometers), **M** (miles), **n** (nautical miles), or **u** (survey feet).
- **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].

Return type
DataArray | None

Returns

ret – 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`)

Example

```
>>> import numpy as np
>>> import pygmt
>>> # Create an array of longitude/latitude coordinates
>>> coords_list = [[85.5, 22.3], [82.3, 22.6], [85.8, 22.4], [86.5, 23.3]]
>>> coords_array = np.array(coords_list)
>>> # Perform a calculation of the distance to
>>> # each point from Voronoi polygons
>>> grid = pygmt.sphdistance(
...     data=coords_array, spacing=[1, 2], region=[82, 87, 22, 24]
... )
```

pygmt.sphinterpolate

pygmt.**sphinterpolate**(*data*, *outgrid=None*, ***kwargs*)

Spherical gridding in tension of data on a sphere.

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/6.5/sphinterpolate.html

Aliases:

- I = spacing
- R = region
- V = verbose

Parameters

- **data** (str | PathLike | dict | ndarray | DataFrame | Dataset | GeoDataFrame) – Pass in (x, y, z) or (longitude, latitude, elevation) values by providing a file name to an ASCII data table, a 2-D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1-D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **outgrid** (str | PathLike | None, default: None) – Name of the output netCDF grid file. If not specified, will return an `xarray.DataArray` object. For writing a specific grid file format or applying basic data operations to the output grid, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.
- **spacing** (*float, str, or list*) – *x_inc*[**+e**|**n**][/*y_inc*[**+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 [*Full usage*].

Return type
`DataArray` | `None`

Returns

ret – 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`)

Example

```
>>> import pygmt
>>> # Load a table of Mars with longitude/latitude/radius columns
>>> mars_shape = pygmt.datasets.load_sample_data(name="mars_shape")
>>> # Perform Delaunay triangulation on the table data
>>> # to produce a grid with a 1 arc-degree spacing
>>> grid = pygmt.sphinterpolate(data=mars_shape, spacing=1, region="g")
```

pygmt.surface

pygmt.**surface**(*data=None*, *x=None*, *y=None*, *z=None*, *outgrid=None*, ***kwargs*)

Grid table data using adjustable tension continuous curvature splines.

Surface reads randomly-spaced (x, y, z) triplets 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. Here, $t = 0$ gives the "minimum curvature" solution. Minimum curvature can cause undesired oscillations and false local maxima or minima (see Smith and Wessel, 1990), and you may wish to use $t > 0$ to suppress these effects. Experience suggests $t \sim 0.25$ usually looks good for potential field data and t should be larger ($t \sim 0.35$) for steep topography data. $t = 1$ gives a harmonic surface (no maxima or minima are possible except at control data points). It is recommended that the user preprocess the data with *pygmt.blockmean*, *pygmt.blockmedian*, or *pygmt.blockmode* to avoid spatial aliasing and eliminate redundant data. You may impose lower and/or upper bounds on the solution. These

may be entered in the form of a fixed value, a grid with values, or simply be the minimum/maximum input data values. Natural boundary conditions are applied at the edges, except for geographic data with 360-degree range where we apply periodic boundary conditions in the longitude direction.

Takes a matrix, (x, y, z) 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/6.5/surface.html

Aliases:

- C = convergence
- I = spacing
- Ll = lower
- Lu = upper
- M = maxradius
- R = region
- T = tension
- V = verbose
- a = aspatial
- b = binary
- d = nodata
- e = find
- f = coltypes
- h = header
- i = incols
- r = registration
- w = wrap

Parameters

- **data** (`str` | `PathLike` | `dict` | `ndarray` | `DataFrame` | `Dataset` | `GeoDataFrame` | `None`, default: `None`) – Pass in (x, y, z) or (longitude, latitude, elevation) values by providing a file name to an ASCII data table, a 2-D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1-D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **x/y/z** (*1-D arrays*) – Arrays of x and y coordinates and values z of the data points.
- **spacing** (*float, str, or list*) – *x_inc*[**+e**|**n**][/*y_inc*[**+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` | `PathLike` | `None`, default: `None`) – Name of the output netCDF grid file. If not specified, will return an `xarray.DataArray` object. For writing a specific grid file format or applying basic data operations to the output grid, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.
- **convergence** (*float*) – Optional. Convergence limit. Iteration is assumed to have converged when the maximum absolute change in any grid value is less than `convergence`. (Units same as data z units). Alternatively, give limit in percentage of root-mean-square (rms)

deviation by appending %. [Default is scaled to 10^{-4} of the rms deviation of the data from a best-fit (least-squares) plane.] This is the final convergence limit at the desired grid spacing; for intermediate (coarser) grids the effective convergence limit is divided by the grid spacing multiplier.

- **maxradius** (*float or str*) – Optional. After solving for the surface, apply a mask so that nodes farther than `maxradius` away from a data constraint are set to NaN [Default is no masking]. Append a distance unit (see Units) if needed. One can also select the nodes to mask by using the *n_cells***c** form. Here *n_cells* means the number of cells around the node is controlled by a data point. As an example `"0c"` means that only the cell where the point lies is filled, `"1c"` keeps one cell beyond that (i.e. makes a 3x3 square neighborhood), and so on.
- **lower** (*float or str*) – Optional. Impose limits on the output solution. Parameter `lower` sets the lower bound. `lower` can be the name of a grid file with lower bound values, a fixed value, **d** to set to minimum input value, or **u** for unconstrained [Default]. Grid files used to set the limits may contain NaNs. In the presence of NaNs, the limit of a node masked with NaN is unconstrained.
- **upper** (*float or str*) – Optional. Impose limits on the output solution. Parameter `upper` sets the upper bound and can be the name of a grid file with upper bound values, a fixed value, **d** to set to maximum input value, or **u** for unconstrained [Default]. Grid files used to set the limits may contain NaNs. In the presence of NaNs, the limit of a node masked with NaN is unconstrained.
- **tension** (*float or str*) – [**b**|**i**]. Optional. Tension factor[s]. These must be between 0 and 1. Tension may be used in the interior solution (above equation, where it suppresses spurious oscillations) and in the boundary conditions (where it tends to flatten the solution approaching the edges). Add **i***tension* to set interior tension, and **b***tension* to set boundary tension. If you do not prepend **i** or **b**, both will be set to the same value. [Default is 0 for both and gives minimum curvature solution.]
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **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/6.5/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/6.5/gmt.html#bi-full.

- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `nodata="-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/6.5/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 1-D 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 *1-D 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/6.5/gmt.html#w-full.

Return type
DataArray | None

Returns
ret – 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`)

Example

```
>>> import pygmt
>>> # Load a sample table of topography
>>> topography = pygmt.datasets.load_sample_data(name="notre_dame_topography")
>>> # Perform gridding of topography data
>>> grid = pygmt.surface(data=topography, spacing=1, region=[0, 4, 0, 8])
```

pygmt.triangulate

class pygmt.**triangulate**

Delaunay triangulation or Voronoi partitioning and gridding of Cartesian data.

Triangulate reads in x,y[,z] data and performs Delaunay triangulation, i.e., it finds how the points should be connected to give the most equilateral triangulation possible. If a map projection (give `region` and `projection`) is chosen then it is applied before the triangulation is calculated. By default, the output is triplets of point id numbers that make up each triangle. The id numbers refer to the points position (line number, starting at 0 for the first line) in the input file. If `outgrid` and `spacing` are set a grid will be calculated based on the surface defined by the planar triangles. The actual algorithm used in the triangulations is either that of Watson [1982] or Shewchuk [1996] [Default is Shewchuk if installed; type `gmt get GMT_TRIANGULATE` on the command line to see which method is selected]. Furthermore, if the Shewchuk algorithm is installed then you can also perform the calculation of Voronoi polygons and optionally grid your data via the natural nearest neighbor algorithm.

> **Note**
>
> For geographic data with global or very large extent you should consider sphtriangulate instead since `triangulate` is a Cartesian or small-geographic area operator and is unaware of periodic or polar boundary conditions.

Methods Summary

triangulate.delaunay_triples([data, x, y, ...])	Delaunay triangle based gridding of Cartesian data.
triangulate.regular_grid([data, x, y, z, ...])	Delaunay triangle based gridding of Cartesian data.

pygmt.triangulate.delaunay_triples

static triangulate.**delaunay_triples**(*data=None*, *x=None*, *y=None*, *z=None*, ***, *output_type='pandas'*, *outfile=None*, ***kwargs*)

Delaunay triangle based gridding of Cartesian data.

Reads in x,y[,z] data and performs Delaunay triangulation, i.e., it finds how the points should be connected to give the most equilateral triangulation possible. If a map projection (give `region` and `projection`) is chosen then it is applied before the triangulation is calculated. The actual algorithm used in the triangulations is either that of Watson [1982] or Shewchuk [1996] [Default if installed; type `gmt get GMT_TRIANGULATE` on the command line to see which method is selected).

Must provide either `data` or `x`, `y`, and `z`.

Full option list at https://docs.generic-mapping-tools.org/6.5/triangulate.html

Aliases:

- I = spacing
- J = projection
- R = region
- V = verbose
- b = binary
- d = nodata
- e = find
- f = coltypes
- h = header
- i = incols
- r = registration
- s = skiprows
- w = wrap

Parameters

- **x/y/z** (`numpy.ndarray`) – Arrays of x and y coordinates and values z of the data points.
- **data** (`str` | `PathLike` | `dict` | `ndarray` | `DataFrame` | `Dataset` | `GeoDataFrame` | `None`, default: `None`) – Pass in (x, y, z) or (longitude, latitude, elevation) values by providing a file name to an ASCII data table, a 2-D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1-D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **projection** (*str*) – *projcode*[*projparams*/]*width*|*scale*. Select map *projection*.
- **region** (*str or list*) – *xmin*/*xmax*/*ymin*/*ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **output_type** (`Literal['pandas', 'numpy', 'file']`, default: `'pandas'`) – Desired output type of the result data.
  - `pandas` will return a `pandas.DataFrame` object.
  - `numpy` will return a `numpy.ndarray` object.
  - `file` will save the result to the file specified by the `outfile` parameter.

- **outfile** (*str* | *PathLike* | *None*, default: `None`) – File name for saving the result data. Required if `output_type="file"`. If specified, `output_type` will be forced to be `"file"`.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **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/6.5/gmt.html#bi-full.
- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `nodata="-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/6.5/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 1-D 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 *1-D 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].
- **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/6.5/gmt.html#w-full.

Return type
DataFrame | ndarray | None

Returns

ret – Return type depends on `outfile` and `output_type`:

- `None` if `outfile` is set (output will be stored in the file set by `outfile`)
- `pandas.DataFrame` or `numpy.ndarray` if `outfile` is not set (depends on `output_type`)

Note

For geographic data with global or very large extent you should consider sphtriangulate instead since `triangulate` is a Cartesian or small-geographic area operator and is unaware of periodic or polar boundary conditions.

pygmt.triangulate.regular_grid

static triangulate.**regular_grid**(*data=None*, *x=None*, *y=None*, *z=None*, *outgrid=None*, ***kwargs*)

Delaunay triangle based gridding of Cartesian data.

Reads in x,y[,z] data and performs Delaunay triangulation, i.e., it finds how the points should be connected to give the most equilateral triangulation possible. If a map projection (give `region` and `projection`) is chosen then it is applied before the triangulation is calculated. By setting `outgrid` and `spacing`, a grid will be calculated based on the surface defined by the planar triangles. The actual algorithm used in the triangulations is either that of Watson [1982] or Shewchuk [1996] [Default is Shewchuk if installed; type `gmt get GMT_TRIANGULATE` on the command line to see which method is selected]. This choice is made during the GMT installation. Furthermore, if the Shewchuk algorithm is installed then you can also perform the calculation of Voronoi polygons and optionally grid your data via the natural nearest neighbor algorithm.

Must provide either `data` or `x`, `y`, and `z`.

Must provide `region` and `spacing`.

Full option list at https://docs.generic-mapping-tools.org/6.5/triangulate.html

Aliases:

- I = spacing
- J = projection
- R = region
- V = verbose
- b = binary
- d = nodata
- e = find
- f = coltypes
- h = header
- i = incols
- r = registration
- s = skiprows
- w = wrap

Parameters

- **x/y/z** (`numpy.ndarray`) – Arrays of x and y coordinates and values z of the data points.
- **data** (`str` | `PathLike` | `dict` | `ndarray` | `DataFrame` | `Dataset` | `GeoDataFrame` | `None`, default: `None`) – Pass in (x, y[, z]) or (longitude, latitude[, elevation]) values by providing a file name to an ASCII data table, a 2-D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1-D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.
- **projection** (*str*) – *projcode*[*projparams*/]*width*|*scale*. Select map *projection*.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.

- **spacing** (*float, str, or list*) – *x_inc*[**+e**|**n**][/*y_inc*[**+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.
- **outgrid** (`str` | `PathLike` | `None`, default: `None`) – Name of the output netCDF grid file. If not specified, will return an `xarray.DataArray` object. For writing a specific grid file format or applying basic data operations to the output grid, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

  The interpolation is performed in the original coordinates, so if your triangles are close to the poles you are better off projecting all data to a local coordinate system before using `triangulate` (this is true of all gridding routines) or instead select sphtriangulate.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **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/6.5/gmt.html#bi-full.

- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `nodata="-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/6.5/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 1-D 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 *1-D 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)].
- **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/6.5/gmt.html#w-full.

Return type
`DataArray` | `None`

Returns

ret – Return type depends on whether the `outgrid` parameter is set:

- `xarray.DataArray` if `outgrid` is `None` [Default]
- `None` if `outgrid` is a str (grid output is stored in `outgrid`)

Note

For geographic data with global or very large extent you should consider sphtriangulate instead since `triangulate` is a Cartesian or small-geographic area operator and is unaware of periodic or polar boundary conditions.

pygmt.xyz2grd

`pygmt.`**`xyz2grd`**(*data=None*, *x=None*, *y=None*, *z=None*, *outgrid=None*, ***kwargs*)

Convert data table to a grid.

Reads one or more tables with *x, y, z* columns and creates a binary grid file. *pygmt.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/6.5/xyz2grd.html

Aliases:

- A = duplicate
- 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 | PathLike | dict | ndarray | DataFrame | Dataset | GeoDataFrame | None, default: None) – Pass in (x, y, z) or (longitude, latitude, elevation) values by providing a file name to an ASCII data table, a 2-D numpy.ndarray, a pandas.DataFrame, an xarray.Dataset made up of 1-D xarray.DataArray data variables, or a geopandas.GeoDataFrame containing the tabular data.
- **x/y/z** (*1-D arrays*) – The arrays of x and y coordinates and z data points.
- **outgrid** (str | PathLike | None, default: None) – Name of the output netCDF grid file. If not specified, will return an xarray.DataArray object. For writing a specific grid file format or applying basic data operations to the output grid, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.
- **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 duplicate to change this behavior, except it is ignored if convention 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 minimum values 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** (*float, str, or list*) – *x_inc*[**+e**|**n**][/*y_inc*[**+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*|*scale*. 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 [*Full usage*].

- **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: **La**]. 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/6.5/gmt.html#bi-full.

- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `nodata="-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/6.5/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 1-D 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 *1-D 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/6.5/gmt.html#w-full.

Return type
DataArray | None

Returns

ret – 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`)

Example

```
>>> import numpy as np
>>> import pygmt
>>> # generate a grid for z=x**2+y**2, with an x-range of 0 to 3,
>>> # and a y-range of 10.5 to 12.5. The x- and y-spacings are 1.0 and 0.5.
>>> x, y = np.meshgrid([0, 1, 2, 3], [10.5, 11.0, 11.5, 12.0, 12.5])
>>> z = x**2 + y**2
>>> xx, yy, zz = x.flatten(), y.flatten(), z.flatten()
>>> grid = pygmt.xyz2grd(
...     x=xx, y=yy, z=zz, spacing=(1.0, 0.5), region=[0, 3, 10, 13]
... )
```

Examples using `pygmt.xyz2grd`

8.2.2 Operations on raster data

dimfilter(grid[, outgrid])	Directional filtering of grids in the space domain.
grd2xyz(grid[, output_type, outfile])	Convert grid to data table.

continues on next page

Table 13 – continued from previous page

grdclip(grid[, outgrid, above, below, ...])	Clip the range of grid values.
grdcut(grid[, kind, outgrid])	Extract subregion from a grid or image or a slice from a cube.
grdfill(grid[, outgrid, constantfill, ...])	Interpolate across holes in a grid.
grdfilter(grid[, outgrid])	Filter a grid in the space (or time) domain.
grdgradient(grid[, outgrid])	Compute directional gradients from a grid.
grdhisteq()	Perform histogram equalization for a grid.
grdhisteq.equalize_grid(grid[, outgrid])	Perform histogram equalization for a grid.
grdhisteq.compute_bins(grid[, output_type, ...])	Perform histogram equalization for a grid.
grdlandmask([outgrid])	Create a "wet-dry" mask grid from shoreline database.
grdproject(grid[, outgrid])	Forward and inverse map transformation of grids.
grdsample(grid[, outgrid])	Resample a grid onto a new lattice.
grdtrack(grid[, points, output_type, ...])	Sample one or more grids at specified locations.
grdvolume(grid[, output_type, outfile])	Calculate grid volume and area constrained by a contour.

pygmt.dimfilter

pygmt.**dimfilter**(*grid*, *outgrid=None*, ***kwargs*)

Directional filtering of grids in the space domain.

Filter a grid in the space (or time) domain by dividing the given filter circle into the given number of sectors, applying one of the selected primary convolution or non-convolution filters to each sector, and choosing the final outcome according to the selected secondary filter. It computes distances using Cartesian or Spherical geometries. The output grid can optionally be generated as a subregion of the input and/or with a new increment using `spacing`, which may add an "extra space" in the input data to prevent edge effects for the output grid. If the filter is low-pass, then the output may be less frequently sampled than the input. *pygmt.dimfilter* will not produce a smooth output as other spatial filters do because it returns a minimum median out of *N* medians of *N* sectors. The output can be rough unless the input data are noise-free. Thus, an additional filtering (e.g., Gaussian via *pygmt.grdfilter*) of the DiM-filtered data is generally recommended.

Full option list at https://docs.generic-mapping-tools.org/6.5/dimfilter.html

Aliases:

- D = distance
- F = filter
- I = spacing
- N = sectors
- R = region
- V = verbose

Parameters

- **grid** (`str` | `PathLike` | `DataArray`) – Name of the input grid file or the grid loaded as a `xarray.DataArray` object.

  For reading a specific grid file format or applying basic data operations, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **outgrid** (`str` | `PathLike` | `None`, default: `None`) – Name of the output netCDF grid file. If not specified, will return an `xarray.DataArray` object. For writing a specific grid file format or applying basic data operations to the output grid, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **distance** (*int or str*) – Distance flag tells how grid (x,y) relates to filter width, as follows:

  – **0**: grid (x,y) in 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 two 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.

- **filter** (*str*) – **x***width*[**+l**|**u**]. Set the primary filter type. Choose among convolution and non-convolution filters. Use the filter code **x** followed by the full diameter *width*. Available convolution filters are:
  - **b**: boxcar. Aall weights are equal.
  - **c**: cosine arch. Weights follow a cosine arch curve.
  - **g**: Gaussian. Weights are given by the Gaussian function.

  Non-convolution filters are:
  - **m**: median. Returns median value.
  - **p**: maximum likelihood probability (a mode estimator). Return modal value. If more than one mode is found we return their average value. Append **+l** or **+h** to the filter width if you want to return the smallest or largest of each sector's modal values.
- **sectors** (*str*) – **x***sectors*[**+l**|**u**] Set the secondary filter type **x** and the number of bow-tie sectors. *sectors* must be integer and larger than 0. When *sectors* is set to 1, the secondary filter is not effective. Available secondary filters **x** are:
  - **l**: lower. Return the minimum of all filtered values.
  - **u**: upper. Return the maximum of all filtered values.
  - **a**: average. Return the mean of all filtered values.
  - **m**: median. Return the median of all filtered values.
  - **p**: mode. Return the mode of all filtered values. If more than one mode is found we return their average value. Append **+l** or **+h** to the sectors if you rather want to return the smallest or largest of the modal values.
- **spacing** (*str or list*) – *x_inc* [and optionally *y_inc*] is the output increment. Append **m** to indicate minutes, or **c** to indicate seconds. If the new *x_inc*, *y_inc* are **not** integer multiples of the old ones (in the input data), filtering will be considerably slower. [Default is same as the input.]
- **region** (*str or list*) – [*xmin*, *xmax*, *ymin*, *ymax*]. Define the region of the output points [Default is the same as the input].
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].

Return type
DataArray | None

Returns

ret – 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 the file set by `outgrid`)

Example

```
>>> import pygmt
>>> # Load a grid of Earth relief data
>>> grid = pygmt.datasets.load_earth_relief()
>>> # Create a filtered grid from an input grid.
>>> filtered_grid = pygmt.dimfilter(
...     grid=grid,
...     # Set filter type to "median" and the diameter width to 600 km
...     filter="m600",
...     # Set grid in degrees, width in km
...     distance=4,
...     # Create 6 sectors and return the lowest values in the sector
...     sectors="l6",
...     # Set the region longitude range from 55W to 51W, and the
...     # latitude range from 24S to 19S
...     region=[-55, -51, -24, -19],
... )
```

pygmt.grd2xyz

pygmt.**grd2xyz**(*grid*, *output_type='pandas'*, *outfile=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/6.5/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` | `PathLike` | `DataArray`) – Name of the input grid file or the grid loaded as a `xarray.DataArray` object.

  For reading a specific grid file format or applying basic data operations, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **output_type** (`Literal['pandas', 'numpy', 'file']`, default: `'pandas'`) – Desired output type of the result data.

  – `pandas` will return a `pandas.DataFrame` object.

  – `numpy` will return a `numpy.ndarray` object.

  – `file` will save the result to the file specified by the `outfile` parameter.

- **outfile** (`str` | `PathLike` | `None`, default: `None`) – File name for saving the result data. Required if `output_type="file"`. If specified, `output_type` will be forced to be `"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 [*Full usage*].
- **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/6.5/gmt.html#bi-full.

- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `nodata="-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/6.5/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 1-D 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 *1-D 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].

Return type
`DataFrame` | `ndarray` | `None`

Returns

ret – Return type depends on `outfile` and `output_type`:

- `None` if `outfile` is set (output will be stored in the file set by `outfile`)
- `pandas.DataFrame` or `numpy.ndarray` if `outfile` is not set (depends on `output_type`)

Example

```
>>> import pygmt
>>> # Load a grid of @earth_relief_30m data, with a longitude range of
>>> # 10° E to 30° E, and a latitude range of 15° N to 25° N
>>> grid = pygmt.datasets.load_earth_relief(
...     resolution="30m", region=[10, 30, 15, 25]
... )
>>> # Create a pandas.DataFrame with the xyz data from an input grid
>>> xyz_dataframe = pygmt.grd2xyz(grid=grid, output_type="pandas")
>>> xyz_dataframe.head(n=2)
    lon   lat           z
0  10.0  25.0      965.5
1  10.5  25.0      876.5
```

Examples using `pygmt.grd2xyz`

pygmt.grdclip

pygmt.**grdclip**(*grid*, *outgrid=None*, *above=None*, *below=None*, *between=None*, *replace=None*, ***kwargs*)

Clip the range of grid values.

This function operates on the values of a grid. It can:

- Set values smaller than a threshold to a new value
- Set values larger than a threshold to a new value
- Set values within a range to a new value
- Replace individual values with a new value

Such operations are useful when you want all of a continent or an ocean to fall into one color or gray shade in image processing, when clipping the range of data values is required, or for reclassification of data values. The values can be any number or NaN (Not a Number).

Full option list at https://docs.generic-mapping-tools.org/6.5/grdclip.html

Aliases:

- R = region
- V = verbose

Parameters

- **grid** (str | PathLike | DataArray) – Name of the input grid file or the grid loaded as a xarray.DataArray object.

  For reading a specific grid file format or applying basic data operations, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **outgrid** (str | PathLike | None, default: None) – Name of the output netCDF grid file. If not specified, will return an xarray.DataArray object. For writing a specific grid file format or applying basic data operations to the output grid, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.

- **above** (Sequence[float] | None, default: None) – Pass a sequence of two values in the form of (*high*, *above*), to set all node values greater than *high* to *above*.

- **below** (Sequence[float] | None, default: None) – Pass a sequence of two values in the form of (*low*, *below*) to set all node values less than *low* to *below*.

- **between** (Sequence[float] | Sequence[Sequence[float]] | None, default: None) – Pass a sequence of three values in the form of (*low*, *high*, *between*) to set all node values between *low* and *high* to *between*. It can also accept a sequence of sequences (e.g., list of lists or 2-D numpy array) to set different values for different ranges.

- **replace** (Sequence[float] | Sequence[Sequence[float]] | None, default: None) – Pass a sequence of two values in the form of (*old*, *new*) to replace all node values equal to *old* with *new*. It can also accept a sequence of sequences (e.g., list of lists or 2-D numpy array) to replace different old values with different new values. This is mostly useful when your data are known to be integer values.

- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].

Return type

DataArray | None

Returns

ret – 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 the file set by outgrid)

Example

```
>>> import pygmt
>>> # Load a grid of @earth_relief_30m data, with a longitude range of
>>> # 10° E to 30° E, and a latitude range of 15° N to 25° N
>>> grid = pygmt.datasets.load_earth_relief(
...     resolution="30m", region=[10, 30, 15, 25]
... )
>>> # Report the minimum and maximum data values
>>> [grid.data.min(), grid.data.max()]
[183.5, 1807.0]
>>> # Create a new grid from an input grid. Set all values below 1,000 to
>>> # 0 and all values above 1,500 to 10,000
```

(continues on next page)

(continued from previous page)

```
>>> new_grid = pygmt.grdclip(grid=grid, below=[1000, 0], above=[1500, 10000])
>>> # Report the minimum and maximum data values
>>> [new_grid.data.min(), new_grid.data.max()]
[0.0, 10000.0]
```

Examples using `pygmt.grdclip`

pygmt.grdcut

pygmt.**grdcut**(*grid*, *kind='grid'*, *outgrid=None*, ***kwargs*)

Extract subregion from a grid or image or a slice from a cube.

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/6.5/grdcut.html

Aliases:

- J = projection
- N = extend
- R = region
- S = circ_subregion
- V = verbose
- Z = z_subregion
- f = coltypes

Parameters

- **grid** (`str` | `PathLike` | `DataArray`) – Name of the input grid file or the grid loaded as a `xarray.DataArray` object.

  For reading a specific grid file format or applying basic data operations, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **kind** (`Literal['grid', 'image']`, default: `'grid'`) – The raster data kind. Valid values are `"grid"` and `"image"`. When the input `grid` is a file name, it's difficult to determine if the file is a grid or an image, so we need to specify the raster kind explicitly. The default is `"grid"`.

- **outgrid** (`str` | `PathLike` | `None`, default: `None`) – Name of the output netCDF grid file. If not specified, will return an `xarray.DataArray` object. For writing a specific grid file format or applying basic data operations to the output grid, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **projection** (*str*) – *projcode*[*projparams*/]*width*|*scale*. Select map *projection*.

- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.

- **extend** (*bool 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 [*Full usage*].
- **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/6.5/gmt.html#f-full.

Return type

DataArray | None

Returns

ret – 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 the file set by `outgrid`)

Example

```
>>> import pygmt
>>> # Load a grid of @earth_relief_30m data, with a longitude range of
>>> # 10° E to 30° E, and a latitude range of 15° N to 25° N
>>> grid = pygmt.datasets.load_earth_relief(
...     resolution="30m", region=[10, 30, 15, 25]
... )
>>> # Create a new grid from an input grid, with a longitude range of
>>> # 12° E to 15° E and a latitude range of 21° N to 24° N
>>> new_grid = pygmt.grdcut(grid=grid, region=[12, 15, 21, 24])
```

pygmt.grdfill

pygmt.**grdfill**(*grid*, *outgrid=None*, *constantfill=None*, *gridfill=None*, *neighborfill=None*, *splinefill=None*, *inquire=False*, *mode=None*, ***kwargs*)

Interpolate across holes in a grid.

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 via the `hole` parameter. There are several different algorithms that can be used to replace the hole values. If no holes are found the original unchanged grid is returned.

Full option list at https://docs.generic-mapping-tools.org/6.5/grdfill.html.

Aliases:

- N = hole
- R = region
- V = verbose
- f = coltypes

Parameters

- **grid** (str | PathLike | DataArray) – Name of the input grid file or the grid loaded as a `xarray.DataArray` object.

For reading a specific grid file format or applying basic data operations, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **outgrid** (str | PathLike | None, default: None) – Name of the output netCDF grid file. If not specified, will return an xarray.DataArray object. For writing a specific grid file format or applying basic data operations to the output grid, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.
- **constantfill** (float | None, default: None) – Fill the holes with a constant value. Specify the constant value to use.
- **gridfill** (str | PathLike | DataArray | None, default: None) – Fill the holes with values sampled from another (possibly coarser) grid. Specify the grid (a file name or an xarray.DataArray) to use for the fill.
- **neighborfill** (float | bool | None, default: None) – Fill the holes with the nearest neighbor. Specify the search radius in pixels. If set to True, the default search radius will be used ($r^2 = \sqrt{n^2 + m^2}$, where (*n,m*) are the node dimensions of the grid).
- **splinefill** (float | bool | None, default: None) – Fill the holes with a bicubic spline. Specify the tension value to use. If set to True, no tension will be used.
- **hole** (*float*) – Set the node value used to identify a point as a member of a hole [Default is NaN].
- **inquire** (bool, default: False) – Output the bounds of each hole. The bounds are returned as a 2-D numpy array in the form of (west, east, south, north). No grid fill takes place and outgrid is ignored.
- **mode** (str | None, default: None) – 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]).

  Deprecated since version 0.15.0: Use constantfill, gridfill, neighborfill, or splinefill instead. The parameter will be removed in v0.19.0.
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **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/6.5/gmt.html#f-full.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].

Return type

DataArray | ndarray | None

Returns

ret – If inquire is True, return the bounds of each hole as a 2-D numpy array. Otherwise, the 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 the file set by outgrid)

Example

Fill holes in a bathymetric grid with a constant value of 20.

```
>>> import pygmt
>>> # Load a bathymetric grid with missing data
>>> earth_relief_holes = pygmt.datasets.load_sample_data(name="earth_relief_holes
↪")
>>> # Fill the holes with a constant value of 20
>>> filled_grid = pygmt.grdfill(grid=earth_relief_holes, constantfill=20)
```

Inquire the bounds of each hole.

```
>>> pygmt.grdfill(grid=earth_relief_holes, inquire=True)
array([[1.83333333, 6.16666667, 3.83333333, 8.16666667],
       [6.16666667, 7.83333333, 0.5       , 2.5       ]])
```

pygmt.grdfilter

`pygmt.`**`grdfilter`**(*grid*, *outgrid=None*, ***kwargs*)

Filter a grid in the space (or time) domain.

Filter a grid file in the space (or 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/6.5/grdfilter.html

Aliases:

- D = distance
- F = filter
- I = spacing
- N = nans
- R = region
- T = toggle
- V = verbose
- f = coltypes
- r = registration
- x = cores

Parameters

- **`grid`** (`str` | `PathLike` | `DataArray`) – Name of the input grid file or the grid loaded as a `xarray.DataArray` object.

  For reading a specific grid file format or applying basic data operations, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **`outgrid`** (`str` | `PathLike` | `None`, default: `None`) – Name of the output netCDF grid file. If not specified, will return an `xarray.DataArray` object. For writing a specific grid file format or applying basic data operations to the output grid, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **`filter`** (*`str`*) – **b**|**c**|**g**|**o**|**m**|**p**|**h***width*[*/width2*][*modifiers*]. Name of the filter type you wish 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*) – State how the grid (x,y) relates to the filter *width*:
  - "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 cos(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** (*float, str, or list*) – *x_inc*[**+e**|**n**][/*y_inc*[**+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 affect the filtered output. Use **i** to ignore all NaNs in the calculation of the filtered value [Default]. **r** is same as **i** except if the input node was NaN then the output node will be set to NaN (only applies if both grids are co-registered). **p** will force the filtered value to be NaN if any grid nodes with NaN-values are found inside the filter circle.
- **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 to get the opposite of the input grid [Default gives the same registration as the input grid].
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].

- **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/6.5/gmt.html#f-full.
- **registration** (*str*) – **g**|**p**. Force gridline (**g**) or pixel (**p**) node registration [Default is **g**(ridline)].
- **cores** (*bool or int*) – Specify the number of active cores to be used in any OpenMP-enabled multi-threaded algorithms. By default, all available cores are used. Set a positive number *n* to use *n* cores (if too large it will be truncated to the maximum cores available); or set a negative number *-n* to select (all - *n*) cores (or at least 1 if *n* equals or exceeds all).

Return type

`DataArray` | `None`

Returns

ret – 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 the file set by `outgrid`)

Examples

```
>>> from pathlib import Path
>>> import pygmt
>>> # Apply a filter of 600 km (full width) to the @earth_relief_30m_g file
>>> # and return a filtered field (saved as netCDF)
>>> pygmt.grdfilter(
...     grid="@earth_relief_30m_g",
...     filter="m600",
...     distance="4",
...     region=[150, 250, 10, 40],
...     spacing=0.5,
...     outgrid="filtered_pacific.nc",
... )
>>> Path("filtered_pacific.nc").unlink()  # Cleanup file
>>> # Apply a Gaussian smoothing filter of 600 km to the input DataArray
>>> # and return a filtered DataArray 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*, *outgrid=None*, ***kwargs*)

Compute directional gradients from a grid.

Can accept `azimuth`, `direction`, and `radiance` input to create the resulting gradient.

Full option list at https://docs.generic-mapping-tools.org/6.5/grdgradient.html

Aliases:

- A = azimuth
- D = direction
- E = radiance
- N = normalize
- Q = tiles
- R = region
- S = slope_file
- V = verbose
- f = coltypes
- n = interpolation

Parameters

- **grid** (str | PathLike | DataArray) – Name of the input grid file or the grid loaded as a xarray.DataArray object.

  For reading a specific grid file format or applying basic data operations, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **outgrid** (str | PathLike | None, default: None) – Name of the output netCDF grid file. If not specified, will return an xarray.DataArray object. For writing a specific grid file format or applying basic data operations to the output grid, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **azimuth** (*float, 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**. Control 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 [*Full usage*].
- **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/6.5/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

Return type

`DataArray` | `None`

Returns

ret – 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 the file set by `outgrid`)

Example

```
>>> import pygmt
>>> # Load a grid of @earth_relief_30m data, with a longitude range of
>>> # 10° E to 30° E, and a latitude range of 15° N to 25° N
>>> grid = pygmt.datasets.load_earth_relief(
...     resolution="30m", region=[10, 30, 15, 25]
... )
>>> # Create a new grid from an input grid, set the azimuth to 10 degrees,
>>> new_grid = pygmt.grdgradient(grid=grid, azimuth=10)
```

Examples using `pygmt.grdgradient`

pygmt.grdhisteq

class pygmt.**grdhisteq**

Perform histogram equalization for a grid.

Two common use cases of *pygmt.grdhisteq* are to find data values that divide a grid into patches of equal area (*pygmt.grdhisteq.compute_bins*) or to write a grid with statistics based on some kind of cumulative distribution function (*pygmt.grdhisteq.equalize_grid*).

Histogram equalization provides a way to highlight data that has most values clustered in a small portion of the dynamic range, such as a grid of flat topography with a mountain in the middle. Ordinary gray shading of this grid (using *pygmt.Figure.grdimage* or *pygmt.Figure.grdview*) with a linear mapping from topography to graytone will result in most of the image being very dark gray, with the mountain being almost white. *pygmt.grdhisteq.compute_bins* can provide a list of data values that divide the data range into divisions which have an equal area in the image [Default is 16 if `divisions` is not set]. The `pandas.DataFrame` or ASCII file output can be used to make a colormap with *pygmt.makecpt* and an image with *pygmt.Figure.grdimage* that has all levels of gray occurring equally.

pygmt.grdhisteq.equalize_grid provides a way to write a grid with statistics based on a cumulative distribution function. In this application, the `outgrid` has relative highs and lows in the same (x,y) locations as the `grid`, but the values are changed to reflect their place in the cumulative distribution.

Methods Summary

grdhisteq.compute_bins(grid[, output_type, ...])	Perform histogram equalization for a grid.
grdhisteq.equalize_grid(grid[, outgrid])	Perform histogram equalization for a grid.

pygmt.grdhisteq.compute_bins

static grdhisteq.**compute_bins**(*grid*, *output_type='pandas'*, *outfile=None*, ***kwargs*)

Perform histogram equalization for a grid.

Histogram equalization provides a way to highlight data that has most values clustered in a small portion of the dynamic range, such as a grid of flat topography with a mountain in the middle. Ordinary gray shading of this grid (using *pygmt.Figure.grdimage* or *pygmt.Figure.grdview*) with a linear mapping from topography to graytone will result in most of the image being very dark gray, with the mountain being almost white. *pygmt.grdhisteq.compute_bins* can provide a list of data values that divide the data range into divisions which have an equal area in the image [Default is 16 if `divisions` is not set]. The `pandas.DataFrame` or ASCII file output can be used to make a colormap with *pygmt.makecpt* and an image with *pygmt.Figure.grdimage* that has all levels of gray occurring equally.

Full option list at https://docs.generic-mapping-tools.org/6.5/grdhisteq.html

Aliases:

- C = divisions
- N = gaussian
- Q = quadratic
- R = region
- V = verbose
- h = header

Parameters

- **grid** (`str` | `PathLike` | `DataArray`) – Name of the input grid file or the grid loaded as a `xarray.DataArray` object.

  For reading a specific grid file format or applying basic data operations, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **output_type** (Literal['pandas', 'numpy', 'file'], default: 'pandas') – Desired output type of the result data.
  - pandas will return a pandas.DataFrame object.
  - numpy will return a numpy.ndarray object.
  - file will save the result to the file specified by the outfile parameter.
- **outfile** (str | PathLike | None, default: None) – File name for saving the result data. Required if output_type="file". If specified, output_type will be forced to be "file".
- **divisions** (*int*) – Set the number of divisions of the data range.
- **quadratic** (*bool*) – Perform quadratic equalization [Default is linear].
- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **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.

Return type
DataFrame | ndarray | None

Returns

ret – 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)

Example

```
>>> import pygmt
>>> # Load a grid of @earth_relief_30m data, with a longitude range of
>>> # 10° E to 30° E, and a latitude range of 15° N to 25° N
>>> grid = pygmt.datasets.load_earth_relief(
...     resolution="30m", region=[10, 30, 15, 25]
... )
>>> # Find elevation intervals that split the data range into 5
>>> # divisions, each of which have an equal area in the original grid.
>>> bins = pygmt.grdhisteq.compute_bins(grid=grid, divisions=5)
>>> print(bins)
```

(continues on next page)

(continued from previous page)

```
        start    stop
bin_id
0       183.5   395.0
1       395.0   472.0
2       472.0   575.0
3       575.0   709.5
4       709.5  1807.0
```

See also

pygmt.grd2cpt

Note

This method does a weighted histogram equalization for geographic grids to account for node area varying with latitude.

Examples using `pygmt.grdhisteq.compute_bins`

pygmt.grdhisteq.equalize_grid

static grdhisteq.**equalize_grid**(*grid*, *outgrid=None*, ***kwargs*)

Perform histogram equalization for a grid.

pygmt.grdhisteq.equalize_grid provides a way to write a grid with statistics based on a cumulative distribution function. The `outgrid` has relative highs and lows in the same (x,y) locations as the `grid`, but the values are changed to reflect their place in the cumulative distribution.

Full option list at https://docs.generic-mapping-tools.org/6.5/grdhisteq.html

Aliases:

- C = divisions
- N = gaussian
- Q = quadratic
- R = region
- V = verbose
- h = header

Parameters

- **grid** (`str` | `PathLike` | `DataArray`) – Name of the input grid file or the grid loaded as a `xarray.DataArray` object.

  For reading a specific grid file format or applying basic data operations, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **outgrid** (`str` | `PathLike` | `None`, default: `None`) – Name of the output netCDF grid file. If not specified, will return an `xarray.DataArray` object. For writing a specific grid file format or applying basic data operations to the output grid, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **divisions** (*int*) – Set the number of divisions of the data range.

- **gaussian** (*bool or int or float*) – *norm*. Produce an output grid with standard normal scores using `gaussian=True` or force the scores to fall in the ±*norm* range.

- **quadratic** (*bool*) – Perform quadratic equalization [Default is linear].

- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].

Return type
DataArray | None

Returns

ret – Return type depends on the `outgrid` parameter:

- `xarray.DataArray` if `outgrid` is `None`
- `None` if `outgrid` is a str (grid output is stored in `outgrid`)

Example

```
>>> import pygmt
>>> # Load a grid of @earth_relief_30m data, with a longitude range
>>> # of 10°E to 30°E, and a latitude range of 15°N to 25°N
>>> grid = pygmt.datasets.load_earth_relief(
...     resolution="30m", region=[10, 30, 15, 25]
... )
>>> # Create a new grid with a Gaussian data distribution
>>> grid = pygmt.grdhisteq.equalize_grid(grid=grid, gaussian=True)
```

See also

pygmt.grd2cpt

Note

This method does a weighted histogram equalization for geographic grids to account for node area varying with latitude.

Examples using `pygmt.grdhisteq.equalize_grid`

pygmt.grdlandmask

pygmt.**grdlandmask**(*outgrid=None*, ***kwargs*)

Create a "wet-dry" mask grid from shoreline database.

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/6.5/grdlandmask.html

Aliases:

- A = area_thresh
- D = resolution
- E = bordervalues
- I = spacing
- N = maskvalues
- R = region
- V = verbose
- r = registration
- x = cores

Parameters

- **outgrid** (str | PathLike | None, default: None) – Name of the output netCDF grid file. If not specified, will return an xarray.DataArray object. For writing a specific grid file format or applying basic data operations to the output grid, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.
- **spacing** (*float, str, or list*) – *x_inc*[**+e**|**n**][/*y_inc*[**+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** (*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**]. Select 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, str, 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*]. Set 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 [*Full usage*].

- **registration** (*str*) – **g**|**p**. Force gridline (**g**) or pixel (**p**) node registration [Default is **g**(ridline)].
- **cores** (*bool or int*) – Specify the number of active cores to be used in any OpenMP-enabled multi-threaded algorithms. By default, all available cores are used. Set a positive number *n* to use *n* cores (if too large it will be truncated to the maximum cores available); or set a negative number *-n* to select (all - *n*) cores (or at least 1 if *n* equals or exceeds all).

Return type
DataArray | None

Returns

ret – 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 the file set by `outgrid`)

Example

```
>>> import pygmt
>>> # Create a landmask grid with a longitude range of 125° E to 130° E, a
>>> # latitude range of 30° N to 35° N, and a grid spacing of 1 arc-degree
>>> landmask = pygmt.grdlandmask(spacing=1, region=[125, 130, 30, 35])
```

Examples using `pygmt.grdlandmask`

pygmt.grdproject

pygmt.**grdproject**(*grid*, *outgrid=None*, ***kwargs*)

Forward and inverse map transformation of grids.

This method 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` parameter. 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 large or smaller than that implied by the extent of the grid file.

Full option list at https://docs.generic-mapping-tools.org/6.5/grdproject.html

Aliases:

- C = center
- D = spacing
- E = dpi
- F = scaling
- I = inverse
- J = projection
- M = unit
- R = region
- V = verbose
- n = interpolation
- r = registration

Parameters

- **grid** (str | PathLike | DataArray) – Name of the input grid file or the grid loaded as a `xarray.DataArray` object.

  For reading a specific grid file format or applying basic data operations, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **outgrid** (`str` | `PathLike` | `None`, default: `None`) – Name of the output netCDF grid file. If not specified, will return an `xarray.DataArray` object. For writing a specific grid file format or applying basic data operations to the output grid, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.
- **inverse** (*bool*) – When set to `True` transforms grid from rectangular to geographical [Default is `False`].
- **projection** (*str*) – *projcode*[*projparams*/]*width*|*scale*. 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 [Default is `[0, 0]`].
- **spacing** (*float, str, or list*) – *x_inc*[**+e**|**n**][/*y_inc*[**+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 input data are in actual projected meters [**e**]. To specify other units, append **f** (feet), **k** (kilometers), **M** (statute miles), **n** (nautical miles), **u** (US survey feet), **i** (inches), **c** (centimeters), or **p** (points).
- **unit** (*str*) – Append **c**, **i**, or **p** to indicate that centimeters, inches, or points should be the projected measure unit. Cannot be used with `scaling`.
- **verbose** (*bool or str*) – Select verbosity level [*Full usage*].
- **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)].

Return type
DataArray | None

Returns

ret – 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 the file set by `outgrid`)

Example

```
>>> import pygmt
>>> # Load a grid of @earth_relief_30m data, with a longitude range of
>>> # 10° E to 30° E, and a latitude range of 15° N to 25° N
>>> region = [10, 30, 15, 25]
>>> grid = pygmt.datasets.load_earth_relief(resolution="30m", region=region)
>>> # Project the geographic gridded data onto a rectangular grid
>>> new_grid = pygmt.grdproject(grid=grid, projection="M10c", region=region)
```

pygmt.grdsample

pygmt.**grdsample**(*grid*, *outgrid=None*, ***kwargs*)

Resample a grid onto a new lattice.

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.

Full option list at https://docs.generic-mapping-tools.org/6.5/grdsample.html

Aliases:

- I = spacing
- R = region
- T = translate
- V = verbose
- f = coltypes
- n = interpolation
- r = registration
- x = cores

Parameters

- **grid** (str | PathLike | DataArray) – Name of the input grid file or the grid loaded as a `xarray.DataArray` object.

  For reading a specific grid file format or applying basic data operations, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **outgrid** (str | PathLike | None, default: None) – Name of the output netCDF grid file. If not specified, will return an `xarray.DataArray` object. For writing a specific grid file format or applying basic data operations to the output grid, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **spacing** (*float, str, or list*) – *x_inc*[**+e**|**n**][/*y_inc*[**+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 [*Full usage*].
- **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/6.5/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*) – Specify the number of active cores to be used in any OpenMP-enabled multi-threaded algorithms. By default, all available cores are used. Set a positive number *n* to use *n* cores (if too large it will be truncated to the maximum cores available); or set a negative number *-n* to select (all - *n*) cores (or at least 1 if *n* equals or exceeds all).

Return type

`DataArray` | `None`

Returns

ret – 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 the file set by `outgrid`)

Example

```
>>> import pygmt
>>> # Load a grid of @earth_relief_30m data, with a longitude range of
>>> # 10° E to 30° E, and a latitude range of 15° N to 25° N
>>> grid = pygmt.datasets.load_earth_relief(
...     resolution="30m", region=[10, 30, 15, 25]
... )
>>> # Create a new grid from an input grid, change the registration,
>>> # and set both x- and y-spacing to 0.5 arc-degrees
>>> new_grid = pygmt.grdsample(grid=grid, translate=True, spacing=[0.5, 0.5])
```

pygmt.grdtrack

pygmt.**grdtrack**(*grid*, *points=None*, *output_type='pandas'*, *outfile=None*, *newcolname=None*, ***kwargs*)

Sample one or more grids at specified 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/6.5/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

- **grid** (`str` | `PathLike` | `DataArray`) – Name of the input grid file or the grid loaded as a `xarray.DataArray` object.

  For reading a specific grid file format or applying basic data operations, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **points** (`str` | `PathLike` | `dict` | `ndarray` | `DataFrame` | `Dataset` | `GeoDataFrame` | `None`, default: `None`) – Pass in either a file name to an ASCII data table, a 2-D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1-D `xarray.DataArray` data variables, or a `geopandas.GeoDataFrame` containing the tabular data.

- **output_type** (`Literal['pandas', 'numpy', 'file']`, default: `'pandas'`) – Desired output type of the result data.

  – `pandas` will return a `pandas.DataFrame` object.

  – `numpy` will return a `numpy.ndarray` object.

  - `file` will save the result to the file specified by the `outfile` parameter.

- **outfile** (`str` | `PathLike` | `None`, default: `None`) – File name for saving the result data. Required if `output_type="file"`. If specified, `output_type` will be forced to be `"file"`.
- **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.
- **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 meters 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 the 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 file name is grdtrack_stacked_profile.txt].
  - **+c***fact* : Compute envelope on stacked profile as ±*fact* **deviation* [Default fact value is 2].

  Here are some 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, 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 [*Full usage*].
- **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/6.5/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/6.5/gmt.html#bi-full.
- **nodata** (*str*) – **i**|**o***nodata*. Substitute specific values with NaN (for tabular data). For example, `nodata="-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/6.5/gmt.html#f-full.
- **gap** (*str or list*) – **x**|**y**|**z**|**d**|**X**|**Y**|**D***gap*[**u**][**+a**][**+c***col*][**+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.
  - **x**|**X**: define a gap when there is a large enough change in the x coordinates (uppercase to use projected coordinates).
  - **y**|**Y**: define a gap when there is a large enough change in the y coordinates (uppercase to use projected coordinates).
  - **d**|**D**: define a gap when there is a large enough distance between coordinates (uppercase to use projected coordinates).
  - **z**: define a gap when there is a large enough change in the z data. Use **+c***col* to change the z data column [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**(egrees), **m**(inutes), and **s**(econds) , or (m)**e**(ters), **f**(eet), **k**(ilometers), **M**(iles), or **n**(autical miles) [Default is (m)**e**(ters)].
  - For projected data (**X**|**Y**|**D**), the unit may be **i**(nches), **c**(entimeters), or **p**(oints).

  Append modifier **+a** to specify that *all* the criteria must be met [default imposes breaks if any one criterion is met].

  One of the following modifiers can be appended:
  - **+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 1-D 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 *1-D 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*) – Determine how spherical distances are calculated [*Full usage*].
- **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 1-D 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 *1-D 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/6.5/gmt.html#w-full.

Return type
`DataFrame` | `ndarray` | `None`

Returns

ret – 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`)

Example

```
>>> import pygmt
>>> # Load a grid of @earth_relief_30m data, with a longitude range of
>>> # -118° E to -107° E, and a latitude range of -49° N to -42° N
>>> grid = pygmt.datasets.load_earth_relief(
...     resolution="30m", region=[-118, -107, -49, -42]
... )
>>> # Load a pandas dataframe with ocean ridge points
>>> points = pygmt.datasets.load_sample_data(name="ocean_ridge_points")
>>> # Create a pandas dataframe from an input grid and set of points
>>> # The output dataframe adds a column named "bathymetry"
>>> output_dataframe = pygmt.grdtrack(
...     points=points, grid=grid, newcolname="bathymetry"
... )
```

Examples using `pygmt.grdtrack`

pygmt.grdvolume

`pygmt.`**`grdvolume`**(*grid*, *output_type='pandas'*, *outfile=None*, ***kwargs*)

Calculate grid volume and area constrained by a contour.

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/6.5/grdvolume.html

Aliases:

- C = contour
- R = region
- S = unit
- V = verbose

Parameters

- **grid** (str | PathLike | DataArray) – Name of the input grid file or the grid loaded as a xarray.DataArray object.

  For reading a specific grid file format or applying basic data operations, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **output_type** (Literal['pandas', 'numpy', 'file'], default: 'pandas') – Desired output type of the result data.

  – pandas will return a pandas.DataFrame object.

  – numpy will return a numpy.ndarray object.

  – file will save the result to the file specified by the outfile parameter.

- **outfile** (str | PathLike | None, default: None) – File name for saving the result data. Required if output_type="file". If specified, output_type will be forced to be "file".

- **contour** (*str, 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 [*Full usage*].

Return type

DataFrame | ndarray | None

Returns

ret – 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)

Example

```
>>> import pygmt
>>> # Load a grid of @earth_relief_30m data, with a longitude range of
>>> # 10° E to 30° E, and a latitude range of 15° N to 25° N
>>> grid = pygmt.datasets.load_earth_relief(
...     resolution="30m", region=[10, 30, 15, 25]
... )
>>> # Create a pandas dataframe that contains the contour, area, volume,
>>> # and maximum mean height above the plane specified by the given
>>> # contour and below the surface; set the minimum contour z-value to
>>> # 200, the maximum to 400, and the interval to 50.
>>> output_dataframe = pygmt.grdvolume(
```

(continues on next page)

(continued from previous page)

```
...     grid=grid, contour=[200, 400, 50], output_type="pandas"
... )
>>> print(output_dataframe)
       0             1             2           3
0  200.0  2.323600e+12  8.523815e+14  366.836554
1  250.0  2.275864e+12  7.371655e+14  323.905736
2  300.0  2.166707e+12  6.258570e+14  288.851699
3  350.0  2.019284e+12  5.207732e+14  257.899955
4  400.0  1.870441e+12  4.236191e+14  226.480847
```

8.2.3 Crossover analysis with x2sys

x2sys_init(tag, *[, fmtfile, suffix, force, ...])	Initialize a new x2sys track database.
x2sys_cross([tracks, outfile])	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 environment variable *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/6.5/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*) – Specify 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**. Select geographical coordinates. Append **d** for discontinuity at the Dateline (makes longitude go from -180° E to +180° E) or **g** for discontinuity at Greenwich (makes longitude go from 0° E to 360° E [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*. Set 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 ft/s
  - **k**: kilometers or km/hr
  - **m**: miles or mi/hr
  - **n**: nautical miles or knots
  - **u**: survey feet or sft/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 [*Full usage*].
- **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*) – Determine how spherical distances are calculated [*Full usage*].

pygmt.x2sys_cross

`pygmt.`**`x2sys_cross`**(*tracks=None*, *outfile=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/6.5/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, 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 file names 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 environment variable *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 | PathLike | None, default: None) – 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 file name (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 [*Full usage*].
- **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].

Return type
`DataFrame` | `None`

Returns
crossover_errors – Table containing crossover error information. A `pandas.DataFrame` object is returned if `outfile` is not set, otherwise `None` is returned and output will be stored in file set by `outfile`.

8.3 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.

8.3.1 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.

Deprecated since version v0.16.0: The 'pygmt.io.load_dataarray' function will be removed in v0.20.0. Please use *xarray.load_dataarray(..., engine='gmt', raster_kind='grid')* instead if you were reading grids using the engine='netcdf'; otherwise use *raster_kind='image'* if you were reading multi-band images using engine='rasterio'.

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`

8.4 GMT Defaults

Operations on GMT defaults:

config(*[, COLOR_BACKGROUND, ...])	Change GMT default settings globally or locally.

8.4.1 pygmt.config

class pygmt.**config**(*, *COLOR_BACKGROUND=None*, *COLOR_FOREGROUND=None*, *COLOR_CPT=None*, *COLOR_NAN=None*, *COLOR_MODEL=None*, *COLOR_HSV_MIN_S=None*, *COLOR_HSV_MAX_S=None*, *COLOR_HSV_MIN_V=None*, *COLOR_HSV_MAX_V=None*, *COLOR_SET=None*, *DIR_CACHE=None*, *DIR_DATA=None*, *DIR_DCW=None*, *DIR_GSHHG=None*, *FONT_ANNOT_PRIMARY=None*, *FONT_ANNOT_SECONDARY=None*, *FONT_HEADING=None*, *FONT_LABEL=None*, *FONT_LOGO=None*, *FONT_SUBTITLE=None*, *FONT_TAG=None*, *FONT_TITLE=None*, *FORMAT_CLOCK_IN=None*, *FORMAT_CLOCK_OUT=None*, *FORMAT_CLOCK_MAP=None*, *FORMAT_DATE_IN=None*, *FORMAT_DATE_OUT=None*, *FORMAT_DATE_MAP=None*, *FORMAT_GEO_OUT=None*, *FORMAT_GEO_MAP=None*, *FORMAT_FLOAT_OUT=None*, *FORMAT_FLOAT_MAP=None*, *FORMAT_TIME_PRIMARY_MAP=None*, *FORMAT_TIME_SECONDARY_MAP=None*, *FORMAT_TIME_STAMP=None*, *GMT_DATA_SERVER=None*, *GMT_DATA_SERVER_LIMIT=None*, *GMT_DATA_UPDATE_INTERVAL=None*, *GMT_COMPATIBILITY=None*, *GMT_CUSTOM_LIBS=None*, *GMT_EXPORT_TYPE=None*, *GMT_EXTRAPOLATE_VAL=None*, *GMT_FFT=None*, *GMT_GRAPHICS_DPU=None*, *GMT_GRAPHICS_FORMAT=None*, *GMT_HISTORY=None*, *GMT_INTERPOLANT=None*, *GMT_LANGUAGE=None*, *GMT_MAX_CORES=None*, *GMT_THEME=None*, *GMT_TRIANGULATE=None*, *GMT_VERBOSE=None*, *IO_COL_SEPARATOR=None*, *IO_FIRST_HEADER=None*, *IO_GRIDFILE_FORMAT=None*, *IO_GRIDFILE_SHORTHAND=None*, *IO_HEADER=None*, *IO_HEADER_MARKER=None*, *IO_N_HEADER_RECS=None*, *IO_NAN_RECORDS=None*, *IO_NC4_CHUNK_SIZE=None*, *IO_NC4_DEFLATION_LEVEL=None*, *IO_LONLAT_TOGGLE=None*, *IO_SEGMENT_BINARY=None*, *IO_SEGMENT_MARKER=None*, *MAP_ANNOT_MIN_ANGLE=None*, *MAP_ANNOT_MIN_SPACING=None*, *MAP_ANNOT_OBLIQUE=None*, *MAP_ANNOT_OFFSET_PRIMARY=None*, *MAP_ANNOT_OFFSET_SECONDARY=None*, *MAP_ANNOT_ORTHO=None*, *MAP_DEFAULT_PEN=None*, *MAP_DEGREE_SYMBOL=None*, *MAP_EMBELLISHMENT_MODE=None*, *MAP_FRAME_AXES=None*, *MAP_FRAME_PEN=None*, *MAP_FRAME_PERCENT=None*, *MAP_FRAME_TYPE=None*, *MAP_FRAME_WIDTH=None*, *MAP_GRID_CROSS_SIZE_PRIMARY=None*, *MAP_GRID_CROSS_SIZE_SECONDARY=None*, *MAP_GRID_PEN_PRIMARY=None*, *MAP_GRID_PEN_SECONDARY=None*, *MAP_HEADING_OFFSET=None*, *MAP_LABEL_MODE=None*, *MAP_LABEL_OFFSET=None*, *MAP_LINE_STEP=None*, *MAP_LOGO=None*, *MAP_LOGO_POS=None*, *MAP_ORIGIN_X=None*, *MAP_ORIGIN_Y=None*, *MAP_POLAR_CAP=None*, *MAP_SCALE_HEIGHT=None*, *MAP_TICK_LENGTH_PRIMARY=None*, *MAP_TICK_LENGTH_SECONDARY=None*, *MAP_TICK_PEN_PRIMARY=None*, *MAP_TICK_PEN_SECONDARY=None*, *MAP_TITLE_OFFSET=None*, *MAP_VECTOR_SHAPE=None*, *PROJ_AUX_LATITUDE=None*, *PROJ_DATUM=None*, *PROJ_ELLIPSOID=None*, *PROJ_GEODESIC=None*, *PROJ_LENGTH_UNIT=None*, *PROJ_MEAN_RADIUS=None*, *PROJ_SCALE_FACTOR=None*, *PS_CHAR_ENCODING=None*, *PS_COLOR_MODEL=None*, *PS_COMMENTS=None*, *PS_CONVERT=None*, *PS_IMAGE_COMPRESS=None*, *PS_LINE_CAP=None*, *PS_LINE_JOIN=None*, *PS_MITER_LIMIT=None*, *PS_MEDIA=None*, *PS_PAGE_COLOR=None*, *PS_PAGE_ORIENTATION=None*, *PS_SCALE_X=None*, *PS_SCALE_Y=None*, *PS_TRANSPARENCY=None*, *TIME_EPOCH=None*, *TIME_IS_INTERVAL=None*, *TIME_INTERVAL_FRACTION=None*, *TIME_LEAP_SECONDS=None*, *TIME_REPORT=None*, *TIME_UNIT=None*, *TIME_WEEK_START=None*, *TIME_Y2K_OFFSET_YEAR=None*, *FONT=None*, *FONT_ANNOT=None*, *FORMAT_TIME_MAP=None*, *MAP_ANNOT_OFFSET=None*, *MAP_GRID_CROSS_SIZE=None*, *MAP_GRID_PEN=None*, *MAP_TICK_LENGTH=None*, *MAP_TICK_PEN=None*)

Change GMT default settings globally or locally.

Change GMT default settings globally:

```
pygmt.config(PARAMETER=value)
```

Change GMT default settings locally by using it as a context manager:

```
with pygmt.config(PARAMETER=value):
    ...
```

Full GMT defaults list at https://docs.generic-mapping-tools.org/6.5/gmt.conf.html.

Examples using `pygmt.config`

8.5 Metadata

Getting metadata from tabular or grid data:

info(data, **kwargs)	Get information about data tables.
grdinfo(grid, **kwargs)	Extract information from 2-D grids or 3-D cubes.

8.5.1 pygmt.info

pygmt.**info**(*data*, ***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/6.5/gmtinfo.html

Aliases:

- C = per_column
- I = spacing
- T = nearest_multiple
- V = verbose
- a = aspatial
- f = coltypes
- i = incols
- r = registration

Parameters

- **data** (`str` | `PathLike` | `dict` | `ndarray` | `DataFrame` | `Dataset` | `GeoDataFrame`) – Pass in either a file name to an ASCII data table, a 1-D/2-D `numpy.ndarray`, a `pandas.DataFrame`, an `xarray.Dataset` made up of 1-D `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 [*Full usage*].
- **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/6.5/gmt.html#aspatial-full.
- **incols** (*str or 1-D 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 *1-D 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/6.5/gmt.html#f-full.
- **registration** (*str*) – **g**|**p**. Force gridline (**g**) or pixel (**p**) node registration [Default is **g**(ridline)].

Return type
`ndarray` | `str`

Returns

output (`numpy.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`

8.5.2 pygmt.grdinfo

`pygmt.`**`grdinfo`**(*grid*, ***kwargs*)

Extract information from 2-D grids or 3-D cubes.

Can read the grid from a file or given as an `xarray.DataArray` grid.

Full option list at https://docs.generic-mapping-tools.org/6.5/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 | PathLike | DataArray*) – Name of the input grid file or the grid loaded as a `xarray.DataArray` object.

  For reading a specific grid file format or applying basic data operations, see https://docs.generic-mapping-tools.org/6.5/gmt.html#grd-inout-full for the available modifiers.

- **region** (*str or list*) – *xmin/xmax/ymin/ymax*[**+r**][**+u***unit*]. Specify the *region* of interest.

- **per_column** (*str or bool*) – **n|t**. Format 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.

- **minmax_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 minimum and maximum z-values. 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 minimum and maximum 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 minimum and maximum. To force a symmetrical range about zero, using minus/plus the maximum 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 [*Full usage*].
- **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/6.5/gmt.html#f-full.

Return type
str

Returns
info (*str*) – A string with information about the grid.

8.6 Xarray Integration

`GMTBackendEntrypoint`()	Xarray backend to read raster grid/image files using 'gmt' engine.
`GMTDataArrayAccessor`(xarray_obj)	GMT accessor for `xarray.DataArray`.

8.6.1 pygmt.GMTBackendEntrypoint

class pygmt.**GMTBackendEntrypoint**

Xarray backend to read raster grid/image files using 'gmt' engine.

Internally, GMT uses the netCDF C library to read netCDF files, and GDAL for GeoTIFF and other raster formats. See https://docs.generic-mapping-tools.org/6.5/reference/features.html#grid-file-format for more details about supported formats. This GMT engine can also read GMT remote datasets (file names starting with an @) directly, and pre-loads `pygmt.GMTDataArrayAccessor` properties (in the '.gmt' accessor) for easy access to GMT-specific metadata and features.

When using `xarray.open_dataarray` or `xarray.load_dataarray` with `engine="gmt"`, the `raster_kind` parameter is required and can be either:

- `"grid"`: for reading single-band raster grids
- `"image"`: for reading multi-band raster images

Examples

Read a single-band netCDF file using `raster_kind="grid"`

```
>>> import pygmt
>>> import xarray as xr
>>>
>>> da_grid = xr.open_dataarray(
...     "@static_earth_relief.nc", engine="gmt", raster_kind="grid"
... )
>>> da_grid
<xarray.DataArray 'z' (lat: 14, lon: 8)>...
[112 values with dtype=float32]
Coordinates:
```

(continues on next page)

(continued from previous page)

```
  * lat      (lat) float64... -23.5 -22.5 -21.5 -20.5 ... -12.5 -11.5 -10.5
  * lon      (lon) float64... -54.5 -53.5 -52.5 -51.5 -50.5 -49.5 -48.5 -47.5
Attributes:...
    Conventions:   CF-1.7
    title:         Produced by grdcut
    history:       grdcut @earth_relief_01d_p -R-55/-47/-24/-10 -Gstatic_eart...
    description:   Reduced by Gaussian Cartesian filtering (111.2 km fullwidt...
    actual_range:  [190. 981.]
    long_name:     elevation (m)
```

Read a multi-band GeoTIFF file using `raster_kind="image"`

```
>>> da_image = xr.open_dataarray(
...     "@earth_night_01d", engine="gmt", raster_kind="image"
... )
>>> da_image
<xarray.DataArray 'z' (band: 3, y: 180, x: 360)>...
[194400 values with dtype=uint8]
Coordinates:
  * y        (y) float64... 89.5 88.5 87.5 86.5 ... -86.5 -87.5 -88.5 -89.5
  * x        (x) float64... -179.5 -178.5 -177.5 -176.5 ... 177.5 178.5 179.5
  * band     (band) uint8... 1 2 3
Attributes:...
    long_name:  z
```

Attributes

Methods Summary

GMTBackendEntrypoint.guess_can_open(...)	Backend open_dataset method used by Xarray in open_dataset.
GMTBackendEntrypoint.open_dataset(...[, ...])	Backend open_dataset method used by Xarray in open_dataset.
GMTBackendEntrypoint.open_datatree(...[, ...])	Backend open_datatree method used by Xarray in open_datatree.
GMTBackendEntrypoint.open_groups_as_dict(...)	Opens a dictionary mapping from group names to Datasets.

pygmt.GMTBackendEntrypoint.guess_can_open

GMTBackendEntrypoint.**guess_can_open**(*filename_or_obj*)

Backend open_dataset method used by Xarray in open_dataset.

Return type

bool

pygmt.GMTBackendEntrypoint.open_dataset

GMTBackendEntrypoint.**open_dataset**(*filename_or_obj*, *, *drop_variables=None*, *raster_kind*)

Backend open_dataset method used by Xarray in open_dataset.

Parameters

- **filename_or_obj** (str | PathLike) – File path to a netCDF (.nc), GeoTIFF (.tif) or other grid/image file format that can be read by GMT via the netCDF or GDAL C libraries. See also https://docs.generic-mapping-tools.org/6.5/reference/features.html#grid-file-format.
- **raster_kind** (Literal['grid', 'image']) – Whether to read the file as a "grid" (single-band) or "image" (multi-band).

Return type
Dataset

pygmt.GMTBackendEntrypoint.open_datatree

GMTBackendEntrypoint.**open_datatree**(*filename_or_obj*, *, *drop_variables=None*)

Backend open_datatree method used by Xarray in open_datatree.

Return type
DataTree

pygmt.GMTBackendEntrypoint.open_groups_as_dict

GMTBackendEntrypoint.**open_groups_as_dict**(*filename_or_obj*, *, *drop_variables=None*)

Opens a dictionary mapping from group names to Datasets.

Called by open_groups. This function exists to provide a universal way to open all groups in a file, before applying any additional consistency checks or requirements necessary to create a *DataTree* object (typically done using from_dict).

Return type
dict[str, Dataset]

8.6.2 pygmt.GMTDataArrayAccessor

class pygmt.**GMTDataArrayAccessor**(*xarray_obj*)

GMT accessor for xarray.DataArray.

The *gmt* accessor extends xarray.DataArray to store GMT-specific properties for grids, which are important for PyGMT to correctly process and plot the grids. The *gmt* accessor contains the following properties:

- registration: Grid registration type *pygmt.enums.GridRegistration*.
- gtype: Grid coordinate system type *pygmt.enums.GridType*.

Examples

For GMT's built-in remote datasets, these GMT-specific properties are automatically determined and you can access them 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", registration="pixel")
>>> # See if grid uses Gridline or Pixel registration
>>> grid.gmt.registration
<GridRegistration.PIXEL: 1>
>>> # See if grid is in Cartesian or Geographic coordinate system
>>> grid.gmt.gtype
<GridType.GEOGRAPHIC: 1>
```

For `xarray.DataArray` grids created by yourself, `registration` and `gtype` default to `GridRegistration.GRIDLINE` and `GridType.CARTESIAN` (i.e., a gridline-registered, Cartesian grid). You need to set the correct properties before passing it to PyGMT functions:

```
>>> import numpy as np
>>> import xarray as xr
>>> import pygmt
>>> from pygmt.enums import GridRegistration, GridType
>>> # Create a DataArray in gridline coordinates of sin(lon) * cos(lat)
>>> interval = 2.5
>>> lat = np.arange(90, -90 - interval, -interval)
>>> lon = np.arange(0, 360 + interval, interval)
>>> longrid, latgrid = np.meshgrid(lon, lat)
>>> data = np.sin(np.deg2rad(longrid)) * np.cos(np.deg2rad(latgrid))
>>> grid = xr.DataArray(data, coords=[("latitude", lat), ("longitude", lon)])
>>> # Default to a gridline-registered Cartesian grid
>>> grid.gmt.registration
<GridRegistration.GRIDLINE: 0>
>>> grid.gmt.gtype
<GridType.CARTESIAN: 0>
>>> # Manually set it to a gridline-registered geographic grid
>>> grid.gmt.registration = GridRegistration.GRIDLINE
>>> grid.gmt.gtype = GridType.GEOGRAPHIC
>>> grid.gmt.registration
<GridRegistration.GRIDLINE: 0>
>>> grid.gmt.gtype
<GridType.GEOGRAPHIC: 1>
```

Notes

Due to the limitations of xarray accessors, the GMT accessors are created once per `xarray.DataArray` instance. You may lose these GMT-specific properties when manipulating grids (e.g., arithmetic and slice operations) or when accessing a `xarray.DataArray` from a `xarray.Dataset`. In these cases, you need to manually set these properties before passing the grid to PyGMT.

Inplace assignment operators like `*=` don't create new instances, so the properties are still kept:

```
>>> grid *= 2.0
>>> grid.gmt.registration
<GridRegistration.GRIDLINE: 0>
>>> grid.gmt.gtype
<GridType.GEOGRAPHIC: 1>
```

Other grid operations (e.g., arithmetic or slice operations) create new instances, so the properties will be lost:

```
>>> # grid2 is a slice of the original grid
>>> grid2 = grid[0:30, 50:80]
>>> # Properties are reset to the default values for new instance
>>> grid2.gmt.registration
<GridRegistration.GRIDLINE: 0>
>>> grid2.gmt.gtype
<GridType.CARTESIAN: 0>
>>> # Need to set these properties before passing the grid to PyGMT
>>> grid2.gmt.registration = grid.gmt.registration
```

(continues on next page)

(continued from previous page)

```
>>> grid2.gmt.gtype = grid.gmt.gtype
>>> grid2.gmt.registration
<GridRegistration.GRIDLINE: 0>
>>> grid2.gmt.gtype
<GridType.GEOGRAPHIC: 1>
```

Accessing a `xarray.DataArray` from a `xarray.Dataset` always creates new instances, so these properties are always lost. The workaround is to assign the `xarray.DataArray` into a variable:

```
>>> ds = xr.Dataset({"zval": grid})
>>> ds.zval.gmt.registration
<GridRegistration.GRIDLINE: 0>
>>> ds.zval.gmt.gtype
<GridType.CARTESIAN: 0>
>>> # Manually set these properties won't work as expected
>>> ds.zval.gmt.registration = GridRegistration.GRIDLINE
>>> ds.zval.gmt.gtype = GridType.GEOGRAPHIC
>>> ds.zval.gmt.registration, ds.zval.gmt.gtype
(<GridRegistration.GRIDLINE: 0>, <GridType.CARTESIAN: 0>)
>>> # workaround: assign the DataArray into a variable
>>> zval = ds.zval
>>> zval.gmt.registration, zval.gmt.gtype
(<GridRegistration.GRIDLINE: 0>, <GridType.CARTESIAN: 0>)
>>> zval.gmt.registration = GridRegistration.GRIDLINE
>>> zval.gmt.gtype = GridType.GEOGRAPHIC
>>> zval.gmt.registration, zval.gmt.gtype
(<GridRegistration.GRIDLINE: 0>, <GridType.GEOGRAPHIC: 1>)
```

Attributes

property GMTDataArrayAccessor.**gtype**: *GridType*

Grid coordinate system type *pygmt.enums.GridType*.

property GMTDataArrayAccessor.**registration**: *GridRegistration*

Grid registration type *pygmt.enums.GridRegistration*.

8.7 Enums

GridRegistration	Enum for the grid registration.
GridType	Enum for the grid type.

8.7.1 pygmt.enums.GridRegistration

class pygmt.enums.**GridRegistration**(**values*)

Enum for the grid registration.

GRIDLINE = 0

Gridline registration

PIXEL = 1

Pixel registration

8.7.2 pygmt.enums.GridType

class pygmt.enums.**GridType**(**values*)

Enum for the grid type.

CARTESIAN = 0

Cartesian grid

GEOGRAPHIC = 1

Geographic grid

8.8 Miscellaneous

which(fname, **kwargs)	Find full path to specified files.
show_versions([file])	Print various dependency versions which are useful when submitting bug reports.

8.8.1 pygmt.which

pygmt.**which**(*fname*, ***kwargs*)

Find 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 complete URL, an @file for downloading from the GMT data server, or any of the remote datasets at https://www.pygmt.org/latest/api/index.html#datasets). In these cases, use the `download` parameter 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/6.5/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 any of the remote datasets at https://www.pygmt.org/latest/api/index.html#datasets) we will try to download the file if it is not found in your local data or cache directories. If set to `True` or **l** is passed the file is downloaded 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 [*Full usage*].

Return type

`str` | `list`[`str`]

Returns

path – The path(s) to the file(s), depending on the parameters used.

Raises

FileNotFoundError – If the file is not found.

8.8.2 pygmt.show_versions

pygmt.**show_versions**(*file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Print various dependency versions which are 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

It also warns users if the installed Ghostscript version has serious bugs or is incompatible with the installed GMT version.

Return type

None

8.9 Datasets

PyGMT provides access to GMT's datasets through the `pygmt.datasets` module. These functions will download the datasets automatically the first time they are used and store them in GMT's user data directory.

datasets.list_sample_data()	Report datasets available for tests and documentation examples.
datasets.load_black_marble([resolution, region])	Load NASA Black Marble images in various resolutions.
datasets.load_blue_marble([resolution, region])	Load NASA Blue Marble images in various resolutions.
datasets.load_earth_age([resolution, ...])	Load the Earth seafloor crustal age dataset in various resolutions.
datasets.load_earth_deflection([resolution, ...])	Load the IGPP Earth east-west and north-south deflection datasets in various resolutions.
datasets.load_earth_dist([resolution, ...])	Load the GSHHG Earth distance to shoreline dataset in various resolutions.
datasets.load_earth_free_air_anomaly([...])	Load the IGPP Earth free-air anomaly and uncertainty datasets in various resolutions.
datasets.load_earth_geoid([resolution, ...])	Load the EGM2008 Earth geoid dataset in various resolutions.
datasets.load_earth_magnetic_anomaly([...])	Load the Earth magnetic anomaly datasets in various resolutions.
datasets.load_earth_mask([resolution, ...])	Load the GSHHG Earth mask dataset in various resolutions.
datasets.load_earth_mean_dynamic_topography	Load the CNES Earth mean dynamic topography dataset in various resolutions.
datasets.load_earth_mean_sea_surface([...])	Load the CNES Earth mean sea surface dataset in various resolutions.
datasets.load_earth_relief([resolution, ...])	Load the Earth relief datasets (topography and bathymetry) in various resolutions.
datasets.load_earth_vertical_gravity_gradi	Load the IGPP Earth vertical gravity gradient dataset in various resolutions.

continues on next page

Table 23 – continued from previous page

datasets.load_mars_relief([resolution, ...])	Load the Mars relief dataset in various resolutions.
datasets.load_mercury_relief([resolution, ...])	Load the Mercury relief dataset in various resolutions.
datasets.load_moon_relief([resolution, ...])	Load the Moon relief dataset in various resolutions.
datasets.load_pluto_relief([resolution, ...])	Load the Pluto relief dataset in various resolutions.
datasets.load_venus_relief([resolution, ...])	Load the Venus relief dataset in various resolutions.
datasets.load_sample_data(name)	Load an example dataset from the GMT server.

8.9.1 pygmt.datasets.list_sample_data

pygmt.datasets.**list_sample_data**()

Report datasets available for tests and documentation examples.

Return type
dict[str, str]

Returns
dict – Names and short descriptions of available sample datasets.

See also

load_sample_data
Load an example dataset from the GMT server.

8.9.2 pygmt.datasets.load_black_marble

pygmt.datasets.**load_black_marble**(*resolution='01d'*, *region=None*)

Load NASA Black Marble images in various resolutions.

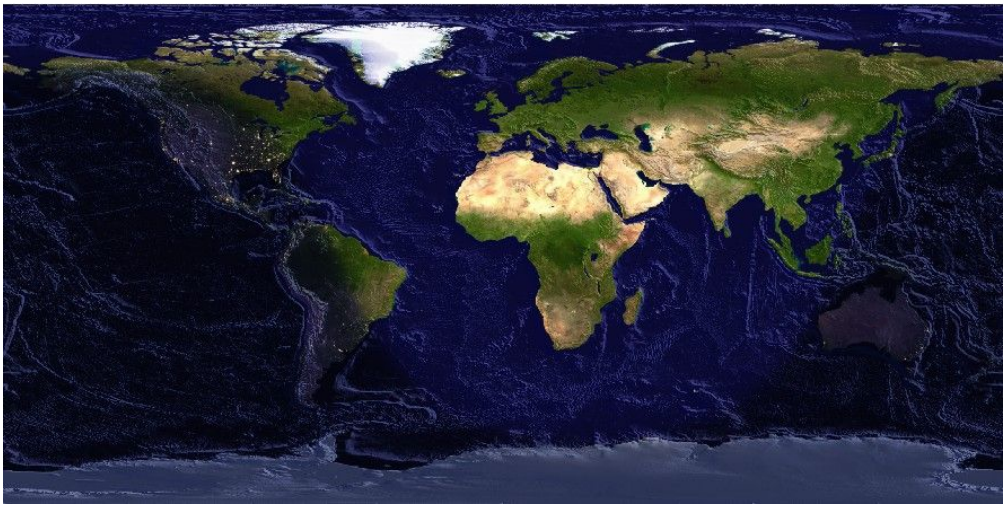


Fig. 2: Earth day/night dataset.

This function downloads the dataset from the GMT data server, caches it in a user data directory (usually `~/.gmt/server/earth/earth_night/`), and load the dataset as an `xarray.DataArray`. An internet connection is required the first time around, but subsequent calls will load the dataset from the local data directory.

The dataset can also be accessed by specifying a file name in any grid processing function or plotting method, using the following file name format: **@earth_night**_*res*. *res* is the grid resolution. If *res* is omitted (i.e., `@earth_night`), GMT automatically selects a suitable resolution based on the current region and projection settings.

Refer to https://www.generic-mapping-tools.org/remote-datasets/earth-daynight.html for more details about available datasets, including version information and references.

Parameters

- **resolution** (`Literal['01d', '30m', '20m', '15m', '10m', '06m', '05m', '04m', '03m', '02m', '01m', '30s']`, default: `'01d'`) – The image resolution. The suffix `d`, `m`, and `s` stand for arc-degrees, arc-minutes, and arc-seconds.
- **region** (`Sequence[float] | str | None`, default: `None`) – The subregion of the image to load, in the form of a sequence [*xmin*, *xmax*, *ymin*, *ymax*].

Return type

`DataArray`

Returns

image – The NASA Black Marble image. Coordinates are latitude and longitude in degrees.

Note

The registration and coordinate system type of the returned `xarray.DataArray` image can be accessed via the GMT accessors (i.e., `image.gmt.registration` and `image.gmt.gtype` respectively). However, these properties may be lost after specific image operations (such as slicing) and will need to be manually set before passing the image to any PyGMT data processing or plotting functions. Refer to *pygmt.GMTDataArrayAccessor* for detailed explanations and workarounds.

Examples

```
>>> from pygmt.datasets import load_black_marble
>>> # Load the default image (pixel-registered 1 arc-degree image)
>>> image = load_black_marble()
```

8.9.3 pygmt.datasets.load_blue_marble

pygmt.datasets.**load_blue_marble**(*resolution='01d'*, *region=None*)

Load NASA Blue Marble images in various resolutions.

This function downloads the dataset from the GMT data server, caches it in a user data directory (usually `~/.gmt/server/earth/earth_day/`), and load the dataset as an `xarray.DataArray`. An internet connection is required the first time around, but subsequent calls will load the dataset from the local data directory.

The dataset can also be accessed by specifying a file name in any image processing function or plotting method, using the following file name format: **@earth_day**_*res*. *res* is the image resolution. If *res* is omitted (i.e., `@earth_day`), GMT automatically selects a suitable resolution based on the current region and projection settings.

Refer to https://www.generic-mapping-tools.org/remote-datasets/earth-daynight.html for more details about available datasets, including version information and references.

Parameters

- **resolution** (`Literal['01d', '30m', '20m', '15m', '10m', '06m', '05m', '04m', '03m', '02m', '01m', '30s']`, default: `'01d'`) – The image resolution. The suffix `d`, `m`, and `s` stand for arc-degrees, arc-minutes, and arc-seconds.

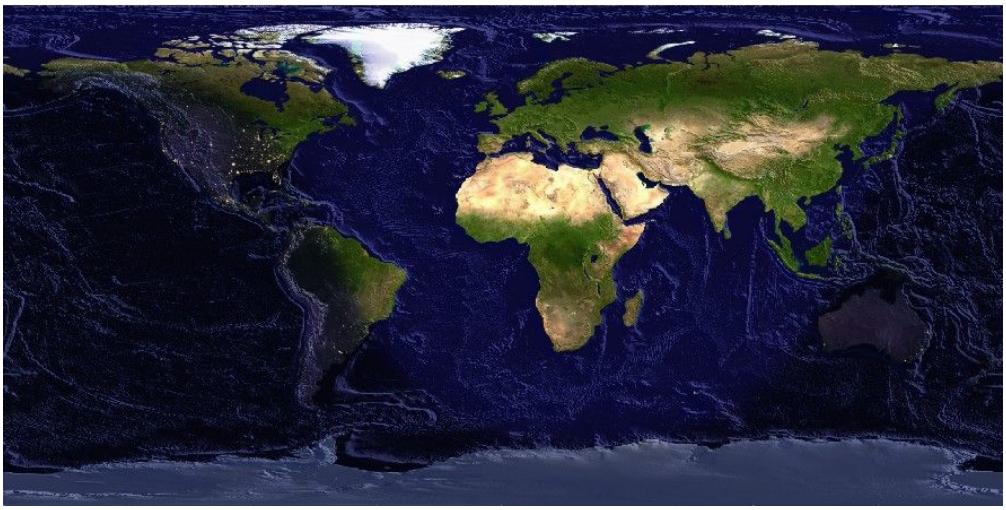


Fig. 3: Earth day/night dataset.

- **region** (Sequence[float] | str | None, default: None) – The subregion of the image to load, in the form of a sequence [*xmin*, *xmax*, *ymin*, *ymax*].

Return type
DataArray

Returns
image – The NASA Blue Marble image. Coordinates are latitude and longitude in degrees.

Note

The registration and coordinate system type of the returned xarray.DataArray image can be accessed via the GMT accessors (i.e., image.gmt.registration and image.gmt.gtype respectively). However, these properties may be lost after specific image operations (such as slicing) and will need to be manually set before passing the image to any PyGMT data processing or plotting functions. Refer to *pygmt.GMTDataArrayAccessor* for detailed explanations and workarounds.

Examples

```
>>> from pygmt.datasets import load_blue_marble
>>> # Load the default image (pixel-registered 1 arc-degree image)
>>> image = load_blue_marble()
```

8.9.4 pygmt.datasets.load_earth_age

pygmt.datasets.**load_earth_age**(*resolution='01d'*, *region=None*, *registration='gridline'*)

Load the Earth seafloor crustal age dataset in various resolutions.

This function downloads the dataset from the GMT data server, caches it in a user data directory (usually ~/.gmt/server/earth/earth_age/), and load the dataset as an xarray.DataArray. An internet connection is required the first time around, but subsequent calls will load the dataset from the local data directory.

The dataset can also be accessed by specifying a file name in any grid processing function or plotting method, using the following file name format: **@earth_age**_*res*_*reg*. *res* is the grid resolution; *reg* is the grid registration type (**p** for pixel registration, **g** for gridline registration). If *reg* is omitted (e.g., @earth_age_01d), the gridline-registered

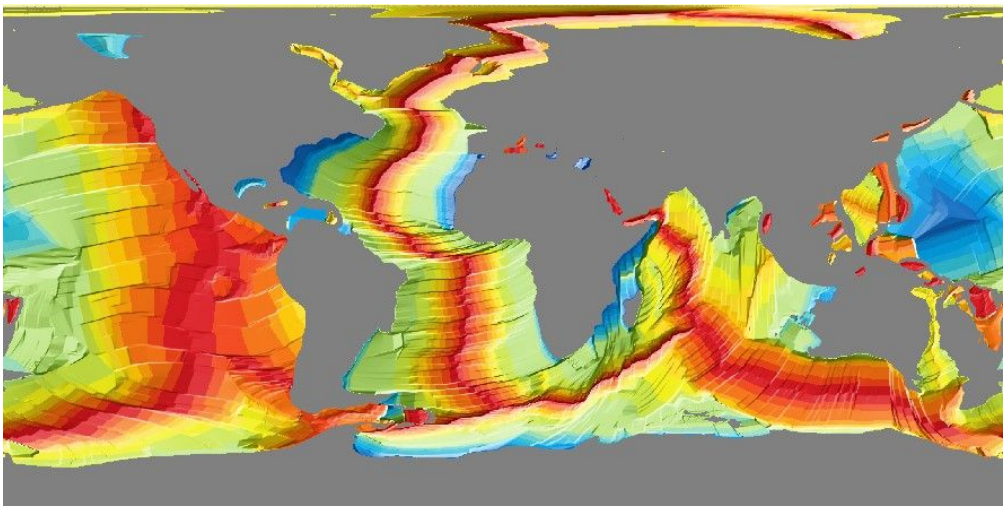


Fig. 4: Earth seafloor crustal age dataset.

grid will be loaded for grid processing functions and the pixel-registered grid will be loaded for plotting functions. If *res* is also omitted (i.e., `@earth_age`), GMT automatically selects a suitable resolution based on the current region and projection settings.

This dataset comes with a color palette table (CPT) file, `@earth_age.cpt`. To use the dataset-specific CPT when plotting the dataset, explicitly set `cmap="@earth_age.cpt"`, otherwise GMT's default CPT (*turbo*) will be used. If the dataset is referenced by the file name in a grid plotting method, the dataset-specific CPT file is used automatically unless another CPT is specified.

Refer to https://www.generic-mapping-tools.org/remote-datasets/earth-age.html for more details about available datasets, including version information and references.

Parameters

- **resolution** (`Literal['01d', '30m', '20m', '15m', '10m', '06m', '05m', '04m', '03m', '02m', '01m']`, default: `'01d'`) – The grid resolution. The suffix `d` and `m` stand for arc-degrees and arc-minutes.
- **region** (`Sequence[float] | str | None`, default: `None`) – The subregion of the grid to load, in the form of a sequence [*xmin*, *xmax*, *ymin*, *ymax*] or an ISO country code. Required for grids with resolutions higher than 5 arc-minutes (i.e., `"05m"`).
- **registration** (`Literal['gridline', 'pixel']`, default: `'gridline'`) – Grid registration type. Either `"pixel"` for pixel registration or `"gridline"` for gridline registration.

Return type

`DataArray`

Returns

grid – The Earth seafloor crustal age grid. Coordinates are latitude and longitude in degrees. Age is in millions of years (Myr).

Note

The registration and coordinate system type of the returned `xarray.DataArray` grid can be accessed via the GMT accessors (i.e., `grid.gmt.registration` and `grid.gmt.gtype` respectively). However, these properties may be lost after specific grid operations (such as slicing) and will need to be manually set before passing

the grid to any PyGMT data processing or plotting functions. Refer to *pygmt.GMTDataArrayAccessor* for detailed explanations and workarounds.

Examples

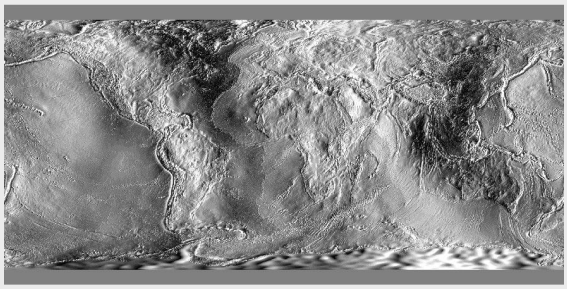
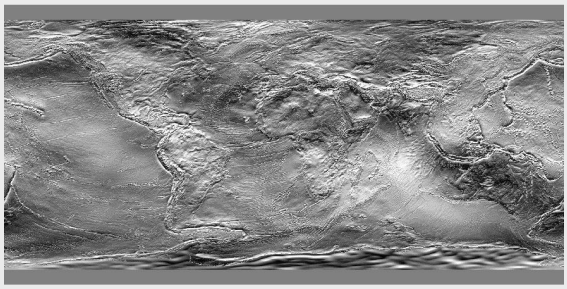
```
>>> from pygmt.datasets import load_earth_age
>>> # Load the default grid (gridline-registered 1 arc-degree grid)
>>> grid = load_earth_age()
>>> # Load the 30 arc-minutes grid with "gridline" registration
>>> grid = load_earth_age(resolution="30m", registration="gridline")
>>> # Load high-resolution (5 arc-minutes) grid for a specific region
>>> grid = load_earth_age(
...     resolution="05m",
...     region=[120, 160, 30, 60],
...     registration="gridline",
... )
```

Examples using `pygmt.datasets.load_earth_age`

8.9.5 pygmt.datasets.load_earth_deflection

pygmt.datasets.**load_earth_deflection**(*resolution='01d'*, *region=None*, *registration=None*, *component='east'*)

Load the IGPP Earth east-west and north-south deflection datasets in various resolutions.

IGPP Earth east-west deflection	IGPP Earth north-south deflection
	

This function downloads the dataset from the GMT data server, caches it in a user data directory (usually `~/.gmt/server/earth/earth_edefl/` and `~/.gmt/server/earth/earth_ndefl/`), and load the dataset as an `xarray.DataArray`. An internet connection is required the first time around, but subsequent calls will load the dataset from the local data directory.

The datasets can also be accessed by specifying a file name in any grid processing function or plotting method, using the following file name format: *@earth_defl_type_res_reg*. *earth_defl_type* is the GMT name for the dataset. The available options are **earth_edefl** and **earth_ndefl**; *res* is the grid resolution; *reg* is the grid registration type (**p** for pixel registration, **g** for gridline registration). If *reg* is omitted (e.g., `@earth_edefl_01d`), the gridline-registered grid will be loaded for grid processing functions and the pixel-registered grid will be loaded for plotting functions. If *res* is also omitted (i.e., `@earth_edefl`), GMT automatically selects a suitable resolution based on the current region and projection settings.

This dataset comes with a color palette table (CPT) file, `@earth_defl.cpt`. To use the dataset-specific CPT when plotting the dataset, explicitly set `cmap="@earth_defl.cpt"`, otherwise GMT's default CPT (*turbo*) will

be used. If the dataset is referenced by the file name in a grid plotting method, the dataset-specific CPT file is used automatically unless another CPT is specified.

Refer to https://www.generic-mapping-tools.org/remote-datasets/earth-edefl.html and https://www.generic-mapping-tools.org/remote-datasets/earth-ndefl.html for more details about available datasets, including version information and references.

Parameters

- **resolution** (Literal['01d', '30m', '20m', '15m', '10m', '06m', '05m', '04m', '03m', '02m', '01m'], default: '01d') – The grid resolution. The suffix d and m stand for arc-degrees and arc-minutes.
- **region** (Sequence[float] | str | None, default: None) – The subregion of the grid to load, in the form of a sequence [*xmin*, *xmax*, *ymin*, *ymax*] or an ISO country code. Required for grids with resolutions higher than 5 arc-minutes (i.e., "05m").
- **registration** (Literal['gridline', 'pixel', None], default: None) – Grid registration type. Either "pixel" for pixel registration or "gridline" for gridline registration. Default is None, which means "gridline" for all resolutions except "01m" which is "pixel" only.
- **component** (Literal['east', 'north'], default: 'east') – By default, the east-west deflection (component="east") is returned, set component="north" to return the north-south deflection.

Return type

DataArray

Returns

grid – The Earth east-west or north-south deflection grid. Coordinates are latitude and longitude in degrees. Deflection values are in micro-radians, where positive (negative) values indicate a deflection to the east or north (west or south).

Note

The registration and coordinate system type of the returned xarray.DataArray grid can be accessed via the GMT accessors (i.e., grid.gmt.registration and grid.gmt.gtype respectively). However, these properties may be lost after specific grid operations (such as slicing) and will need to be manually set before passing the grid to any PyGMT data processing or plotting functions. Refer to *pygmt.GMTDataArrayAccessor* for detailed explanations and workarounds.

Examples

```
>>> from pygmt.datasets import load_earth_deflection
>>> # Load the default grid for east-west deflection (gridline-registered
>>> # 1 arc-degree grid)
>>> grid = load_earth_deflection()
>>> # Load the default grid for north-south deflection
>>> grid = load_earth_deflection(component="north")
>>> # Load the 30 arc-minutes grid with "gridline" registration
>>> grid = load_earth_deflection(resolution="30m", registration="gridline")
>>> # Load high-resolution (5 arc-minutes) grid for a specific region
>>> grid = load_earth_deflection(
...     resolution="05m", region=[120, 160, 30, 60], registration="gridline"
... )
```

8.9.6 pygmt.datasets.load_earth_dist

`pygmt.datasets.load_earth_dist`(*resolution='01d'*, *region=None*, *registration='gridline'*)

Load the GSHHG Earth distance to shoreline dataset in various resolutions.

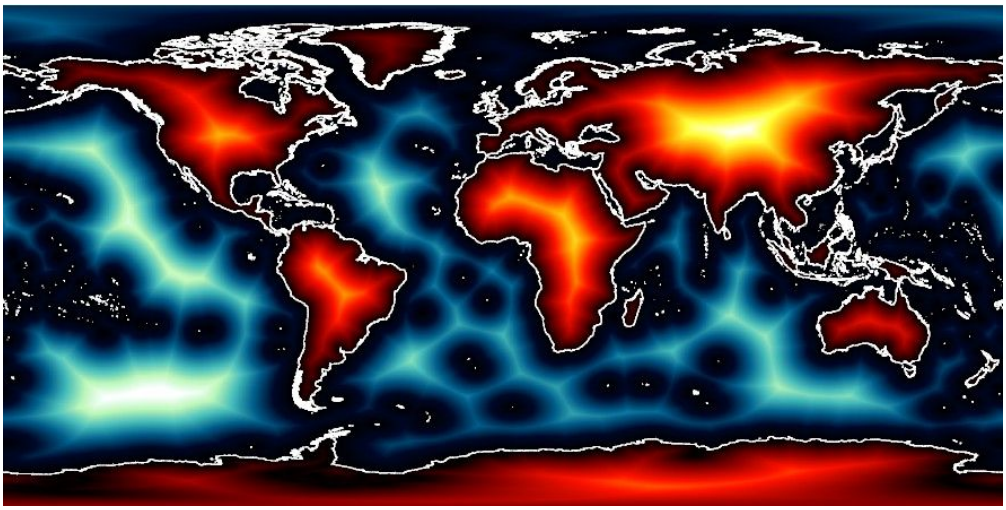


Fig. 5: GSHHG Earth distance to shoreline dataset.

This function downloads the dataset from the GMT data server, caches it in a user data directory (usually `~/.gmt/server/earth/earth_dist/`), and load the dataset as an `xarray.DataArray`. An internet connection is required the first time around, but subsequent calls will load the dataset from the local data directory.

The dataset can also be accessed by specifying a file name in any grid processing function or plotting method, using the following file name format: **@earth_dist**_*res_reg*. *res* is the grid resolution; *reg* is the grid registration type (**p** for pixel registration, **g** for gridline registration). If *reg* is omitted (e.g., `@earth_dist_01d`), the gridline-registered grid will be loaded for grid processing functions and the pixel-registered grid will be loaded for plotting functions. If *res* is also omitted (i.e., `@earth_dist`), GMT automatically selects a suitable resolution based on the current region and projection settings.

This dataset comes with a color palette table (CPT) file, `@earth_dist.cpt`. To use the dataset-specific CPT when plotting the dataset, explicitly set `cmap="@earth_dist.cpt"`, otherwise GMT's default CPT (*turbo*) will be used. If the dataset is referenced by the file name in a grid plotting method, the dataset-specific CPT file is used automatically unless another CPT is specified.

Refer to https://www.generic-mapping-tools.org/remote-datasets/earth-dist.html for more details about available datasets, including version information and references.

Parameters

- **resolution** (`Literal['01d', '30m', '20m', '15m', '10m', '06m', '05m', '04m', '03m', '02m', '01m']`, default: `'01d'`) – The grid resolution. The suffix `d` and `m` stand for arc-degrees and arc-minutes.
- **region** (`Sequence[float] | str | None`, default: `None`) – The subregion of the grid to load, in the form of a sequence [*xmin*, *xmax*, *ymin*, *ymax*] or an ISO country code. Required for grids with resolutions higher than 5 arc-minutes (i.e., `"05m"`).
- **registration** (`Literal['gridline', 'pixel']`, default: `'gridline'`) – Grid registration type. Either `"pixel"` for pixel registration or `"gridline"` for gridline registration.

Return type

`DataArray`

Returns

grid – The GSHHG Earth distance to shoreline grid. Coordinates are latitude and longitude in degrees. Distances are in kilometers, where positive (negative) values mean land to coastline (ocean to coastline).

Note

The registration and coordinate system type of the returned `xarray.DataArray` grid can be accessed via the GMT accessors (i.e., `grid.gmt.registration` and `grid.gmt.gtype` respectively). However, these properties may be lost after specific grid operations (such as slicing) and will need to be manually set before passing the grid to any PyGMT data processing or plotting functions. Refer to *pygmt.GMTDataArrayAccessor* for detailed explanations and workarounds.

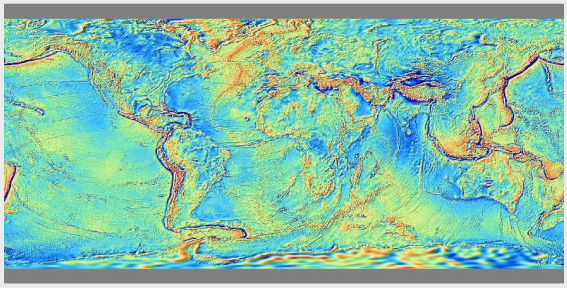
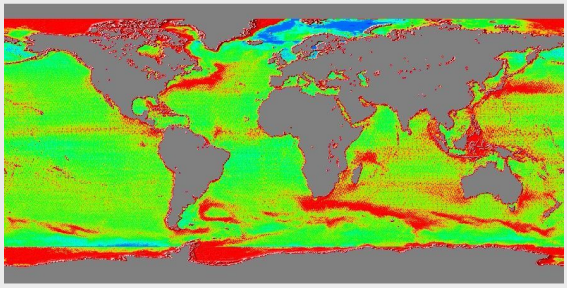
Examples

```
>>> from pygmt.datasets import load_earth_dist
>>> # Load the default grid (gridline-registered 1 arc-degree grid)
>>> grid = load_earth_dist()
>>> # Load the 30 arc-minutes grid with "gridline" registration
>>> grid = load_earth_dist(resolution="30m", registration="gridline")
>>> # Load high-resolution (5 arc-minutes) grid for a specific region
>>> grid = load_earth_dist(
...     resolution="05m",
...     region=[120, 160, 30, 60],
...     registration="gridline",
... )
```

8.9.7 pygmt.datasets.load_earth_free_air_anomaly

`pygmt.datasets.`**`load_earth_free_air_anomaly`**(*resolution='01d'*, *region=None*, *registration=None*, *uncertainty=False*)

Load the IGPP Earth free-air anomaly and uncertainty datasets in various resolutions.

IGPP Earth free-air anomaly	IGPP Earth free-air anomaly uncertainty
	

This function downloads the dataset from the GMT data server, caches it in a user data directory (usually `~/.gmt/server/earth/earth_faa/` or `~/.gmt/server/earth/earth_faaerror/`), and load the dataset as an `xarray.DataArray`. An internet connection is required the first time around, but subsequent calls will load the dataset from the local data directory.

The dataset can also be accessed by specifying a file name in any grid processing function or plotting method, using the following file name format: *@earth_faa_type_res_reg*. *earth_faa_type* is the GMT name for the dataset. The

available options are **earth_faa** and **earth_faaerror**. *res* is the grid resolution; *reg* is the grid registration type (**p** for pixel registration, **g** for gridline registration). If *reg* is omitted (e.g., `@earth_faa_01d`), the gridline-registered grid will be loaded for grid processing functions and the pixel-registered grid will be loaded for plotting functions. If *res* is also omitted (i.e., `@earth_faa`), GMT automatically selects a suitable resolution based on the current region and projection settings.

This dataset comes with two color palette table (CPT) files, `@earth_faa.cpt` and `@earth_faaerror.cpt`. To use the dataset-specific CPT when plotting the dataset, explicitly set `cmap="@earth_faa.cpt"` or `cmap="@earth_faaerror.cpt"`, otherwise GMT's default CPT (*turbo*) will be used. If the dataset is referenced by the file name in a grid plotting method, the dataset-specific CPT file is used automatically unless another CPT is specified.

Refer to https://www.generic-mapping-tools.org/remote-datasets/earth-faa.html and https://www.generic-mapping-tools.org/remote-datasets/earth-faaerror.html for more details about available datasets, including version information and references.

Parameters

- **resolution** (`Literal['01d', '30m', '20m', '15m', '10m', '06m', '05m', '04m', '03m', '02m', '01m']`, default: `'01d'`) – The grid resolution. The suffix `d` and `m` stand for arc-degrees and arc-minutes.
- **region** (`Sequence[float] | str | None`, default: `None`) – The subregion of the grid to load, in the form of a sequence [*xmin*, *xmax*, *ymin*, *ymax*] or an ISO country code. Required for grids with resolutions higher than 5 arc-minutes (i.e., `"05m"`).
- **registration** (`Literal['gridline', 'pixel', None]`, default: `None`) – Grid registration type. Either `"pixel"` for pixel registration or `"gridline"` for gridline registration. Default is `None`, which means `"gridline"` for all resolutions except `"01m"` which is `"pixel"` only.
- **uncertainty** (`bool`, default: `False`) – By default, the Earth free-air anomaly values are returned. Set to `True` to return the related uncertainties instead.

Return type

`DataArray`

Returns

grid – The Earth free-air anomaly (uncertainty) grid. Coordinates are latitude and longitude in degrees. Values and uncertainties are in mGal.

Note

The registration and coordinate system type of the returned `xarray.DataArray` grid can be accessed via the GMT accessors (i.e., `grid.gmt.registration` and `grid.gmt.gtype` respectively). However, these properties may be lost after specific grid operations (such as slicing) and will need to be manually set before passing the grid to any PyGMT data processing or plotting functions. Refer to *`pygmt.GMTDataArrayAccessor`* for detailed explanations and workarounds.

Examples

```
>>> from pygmt.datasets import load_earth_free_air_anomaly
>>> # Load the default grid (gridline-registered 1 arc-degree grid)
>>> grid = load_earth_free_air_anomaly()
>>> # Load the uncertainties related to the default grid
>>> grid = load_earth_free_air_anomaly(uncertainty=True)
```

(continues on next page)

(continued from previous page)

```
>>> # Load the 30 arc-minutes grid with "gridline" registration
>>> grid = load_earth_free_air_anomaly(resolution="30m", registration="gridline")
>>> # Load high-resolution (5 arc-minutes) grid for a specific region
>>> grid = load_earth_free_air_anomaly(
...     resolution="05m", region=[120, 160, 30, 60], registration="gridline"
... )
```

8.9.8 pygmt.datasets.load_earth_geoid

pygmt.datasets.**load_earth_geoid**(*resolution='01d'*, *region=None*, *registration='gridline'*)

Load the EGM2008 Earth geoid dataset in various resolutions.

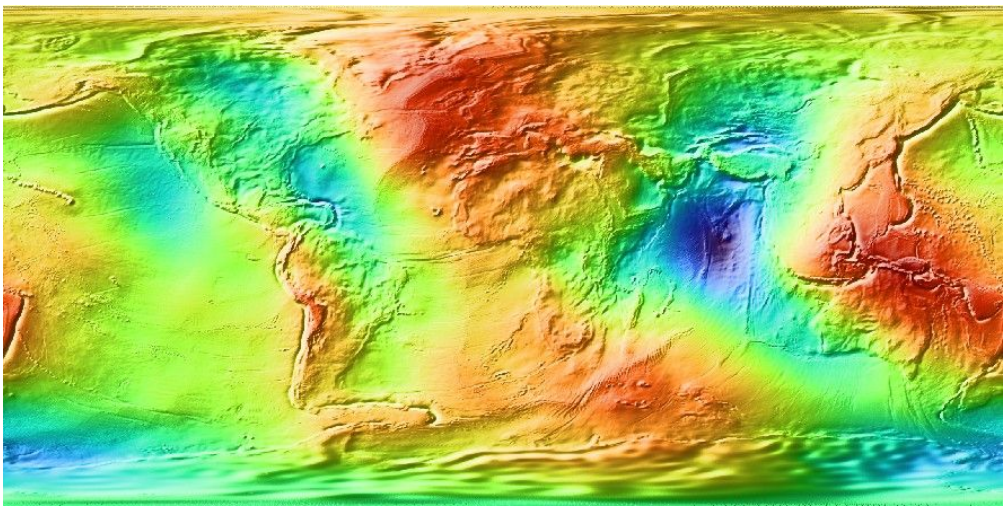


Fig. 6: EGM2008 Earth geoid dataset.

This function downloads the dataset from the GMT data server, caches it in a user data directory (usually `~/.gmt/server/earth/earth_geoid/`), and load the dataset as an `xarray.DataArray`. An internet connection is required the first time around, but subsequent calls will load the dataset from the local data directory.

The dataset can also be accessed by specifying a file name in any grid processing function or plotting method, using the following file name format: **@earth_geoid**_*res*_*reg*. *res* is the grid resolution; *reg* is the grid registration type (**p** for pixel registration, **g** for gridline registration). If *reg* is omitted (e.g., `@earth_geoid_01d`), the gridline-registered grid will be loaded for grid processing functions and the pixel-registered grid will be loaded for plotting functions. If *res* is also omitted (i.e., `@earth_geoid`), GMT automatically selects a suitable resolution based on the current region and projection settings.

Refer to https://www.generic-mapping-tools.org/remote-datasets/earth-geoid.html for more details about available datasets, including version information and references.

Parameters

- **resolution** (`Literal['01d', '30m', '20m', '15m', '10m', '06m', '05m', '04m', '03m', '02m', '01m']`, default: `'01d'`) – The grid resolution. The suffix `d` and `m` stand for arc-degrees and arc-minutes.
- **region** (`Sequence[float] | str | None`, default: `None`) – The subregion of the grid to load, in the form of a sequence [*xmin*, *xmax*, *ymin*, *ymax*] or an ISO country code. Required for grids with resolutions higher than 5 arc-minutes (i.e., `"05m"`).

- **registration** (Literal['gridline', 'pixel'], default: 'gridline') – Grid registration type. Either "pixel" for pixel registration or "gridline" for gridline registration.

Return type
DataArray

Returns
grid – The Earth geoid grid. Coordinates are latitude and longitude in degrees. Units are in meters.

Note

The registration and coordinate system type of the returned xarray.DataArray grid can be accessed via the GMT accessors (i.e., grid.gmt.registration and grid.gmt.gtype respectively). However, these properties may be lost after specific grid operations (such as slicing) and will need to be manually set before passing the grid to any PyGMT data processing or plotting functions. Refer to *pygmt.GMTDataArrayAccessor* for detailed explanations and workarounds.

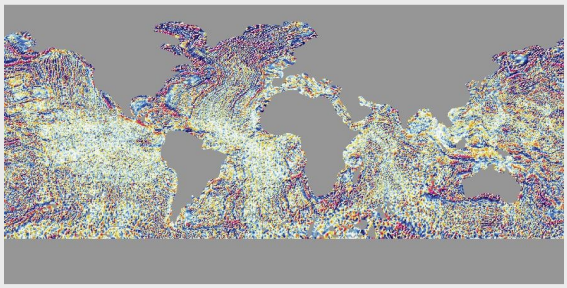
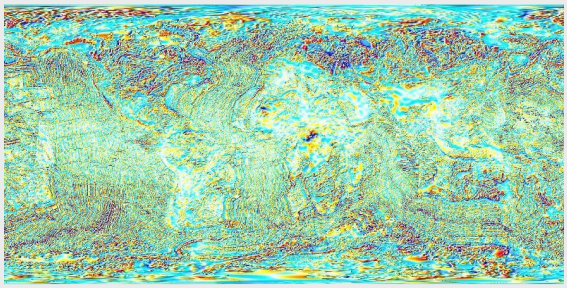
Examples

```
>>> from pygmt.datasets import load_earth_geoid
>>> # Load the default grid (gridline-registered 1 arc-degree grid)
>>> grid = load_earth_geoid()
>>> # Load the 30 arc-minutes grid with "gridline" registration
>>> grid = load_earth_geoid(resolution="30m", registration="gridline")
>>> # Load high-resolution (5 arc-minutes) grid for a specific region
>>> grid = load_earth_geoid(
...     resolution="05m",
...     region=[120, 160, 30, 60],
...     registration="gridline",
... )
```

8.9.9 pygmt.datasets.load_earth_magnetic_anomaly

pygmt.datasets.**load_earth_magnetic_anomaly**(*resolution='01d'*, *region=None*, *registration=None*, *data_source='emag2'*)

Load the Earth magnetic anomaly datasets in various resolutions.

Earth Magnetic Anomaly Model (EMAG2)	World Digital Magnetic Anomaly Map (WDMAM)
	

This function downloads the dataset from the GMT data server, caches it in a user data directory (usually ~/.gmt/server/earth/earth_mag/, ~/.gmt/server/earth/earth_mag4km/, ~/.gmt/server/earth/earth_wdmam/), and load the dataset as an xarray.DataArray. An internet connection is required the first time around, but subsequent calls will load the dataset from the local data directory.

The dataset can also be accessed by specifying a file name in any grid processing function or plotting method, using the following file name format: *@earth_mag_type_res_reg*. *earth_mag_type* is the GMT name for the dataset. The available options are **earth_mag**, **earth_mag4km**, and **earth_wdmam**; *res* is the grid resolution; *reg* is the grid registration type (**p** for pixel registration, **g** for gridline registration). If *reg* is omitted (e.g., `@earth_mag_01d`), the gridline-registered grid will be loaded for grid processing functions and the pixel-registered grid will be loaded for plotting functions. If *res* is also omitted (i.e., `@earth_mag`), GMT automatically selects a suitable resolution based on the current region and projection settings.

The default color palette tables (CPTs) for this dataset are `@earth_mag.cpt` for `data_source="emag2"` and `data_source="emag2_4km"`, and `@earth_wdmam.cpt` for `data_source="wdmam"`. To use the dataset-specific CPT when plotting the dataset, explicitly set `cmap="@earth_mag.cpt"` or `cmap="@earth_wdmam.cpt"`, otherwise GMT's default CPT (*turbo*) will be used. If the dataset is referenced by the file name in a grid plotting method, the dataset-specific CPT file is used automatically unless another CPT is specified.

Refer to https://www.generic-mapping-tools.org/remote-datasets/earth-mag.html and https://www.generic-mapping-tools.org/remote-datasets/earth-wdmam.html for more details about available datasets, including version information and references.

Parameters

- **resolution** (`Literal['01d', '30m', '20m', '15m', '10m', '06m', '05m', '04m', '03m', '02m']`, default: `'01d'`) – The grid resolution. The suffix `d` and `m` stand for arc-degrees and arc-minutes. The resolution `"02m"` is not available for `data_source="wdmam"`.
- **region** (`Sequence[float] | str | None`, default: `None`) – The subregion of the grid to load, in the form of a sequence [*xmin*, *xmax*, *ymin*, *ymax*] or an ISO country code. Required for grids with resolutions higher than 5 arc-minutes (i.e., `"05m"`).
- **registration** (`Literal['gridline', 'pixel', None]`, default: `None`) – Grid registration type. Either `"pixel"` for pixel registration or `"gridline"` for gridline registration. Default is `None`, which means `"gridline"` for all resolutions except `"02m"` for `data_source="emag2"` or `data_source="emag2_4km"`, which are `"pixel"` only.
- **data_source** (`Literal['emag2', 'emag2_4km', 'wdmam']`, default: `'emag2'`) – Select the source of the magnetic anomaly data. Available options are:
  - `"emag2"`: EMAG2 Earth Magnetic Anomaly Model. It only includes data observed at sea level over oceanic regions. See https://www.generic-mapping-tools.org/remote-datasets/earth-mag.html.
  - `"emag2_4km"`: Use a version of EMAG2 where all observations are relative to an altitude of 4 km above the geoid and include data over land.
  - `"wdmam"`: World Digital Magnetic Anomaly Map (WDMAM). See https://www.generic-mapping-tools.org/remote-datasets/earth-wdmam.html.

Return type

`DataArray`

Returns

grid – The Earth magnetic anomaly grid. Coordinates are latitude and longitude in degrees. Units are in nano Tesla (nT).

Note

The registration and coordinate system type of the returned `xarray.DataArray` grid can be accessed via the GMT accessors (i.e., `grid.gmt.registration` and `grid.gmt.gtype` respectively). However, these properties may be lost after specific grid operations (such as slicing) and will need to be manually set before passing

the grid to any PyGMT data processing or plotting functions. Refer to *pygmt.GMTDataArrayAccessor* for detailed explanations and workarounds.

Examples

```
>>> from pygmt.datasets import load_earth_magnetic_anomaly
>>> # Load the default grid (gridline-registered 1 arc-degree grid)
>>> grid = load_earth_magnetic_anomaly()
>>> # Load the 30 arc-minutes grid with "gridline" registration
>>> grid = load_earth_magnetic_anomaly(resolution="30m", registration="gridline")
>>> # Load high-resolution (5 arc-minutes) grid for a specific region
>>> grid = load_earth_magnetic_anomaly(
...     resolution="05m",
...     region=[120, 160, 30, 60],
...     registration="gridline",
... )
>>> # Load the 20 arc-minutes grid of the emag2_4km dataset
>>> grid = load_earth_magnetic_anomaly(
...     resolution="20m", registration="gridline", data_source="emag2_4km"
... )
>>> # Load the 20 arc-minutes grid of the WDMAM dataset
>>> grid = load_earth_magnetic_anomaly(
...     resolution="20m", registration="gridline", data_source="wdmam"
... )
```

8.9.10 pygmt.datasets.load_earth_mask

pygmt.datasets.**load_earth_mask**(*resolution='01d'*, *region=None*, *registration='gridline'*)

Load the GSHHG Earth mask dataset in various resolutions.

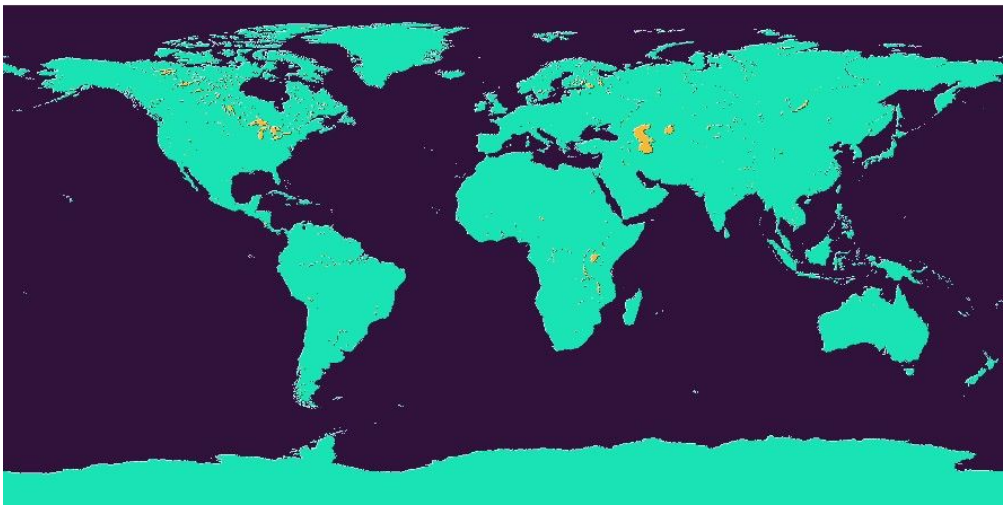


Fig. 7: GSHHG Earth mask dataset.

This function downloads the dataset from the GMT data server, caches it in a user data directory (usually `~/.gmt/server/earth/earth_mask/`), and load the dataset as an `xarray.DataArray`. An internet connection is required the first time around, but subsequent calls will load the dataset from the local data directory.

The dataset can also be accessed by specifying a file name in any grid processing function or plotting method, using the following file name format: **@earth_mask**_*res_reg*. *res* is the grid resolution; *reg* is the grid registration type (**p** for pixel registration, **g** for gridline registration). If *reg* is omitted (e.g., `@earth_mask_01d`), the gridline-registered grid will be loaded for grid processing functions and the pixel-registered grid will be loaded for plotting functions. If *res* is also omitted (i.e., `@earth_mask`), GMT automatically selects a suitable resolution based on the current region and projection settings.

Refer to https://www.generic-mapping-tools.org/remote-datasets/earth-mask.html for more details about available datasets, including version information and references.

Parameters

- **resolution** (`Literal['01d', '30m', '20m', '15m', '10m', '06m', '05m', '04m', '03m', '02m', '01m', '30s', '15s']`, default: `'01d'`) – The grid resolution. The suffix `d`, `m`, and `s` stand for arc-degrees, arc-minutes, and arc-seconds.
- **region** (`Sequence[float] | str | None`, default: `None`) – The subregion of the grid to load, in the form of a sequence [*xmin*, *xmax*, *ymin*, *ymax*] or an ISO country code.
- **registration** (`Literal['gridline', 'pixel']`, default: `'gridline'`) – Grid registration type. Either `"pixel"` for pixel registration or `"gridline"` for gridline registration.

Return type

`DataArray`

Returns

grid – The Earth mask grid. Coordinates are latitude and longitude in degrees. The node values in the mask grids are all in the 0-4 range and reflect different surface types:

- 0: Oceanic areas beyond the shoreline
- 1: Land areas inside the shoreline
- 2: Lakes inside the land areas
- 3: Islands in lakes in the land areas
- 4: Smaller lakes in islands that are found within lakes inside the land area

Note

The registration and coordinate system type of the returned `xarray.DataArray` grid can be accessed via the GMT accessors (i.e., `grid.gmt.registration` and `grid.gmt.gtype` respectively). However, these properties may be lost after specific grid operations (such as slicing) and will need to be manually set before passing the grid to any PyGMT data processing or plotting functions. Refer to *pygmt.GMTDataArrayAccessor* for detailed explanations and workarounds.

Examples

```
>>> from pygmt.datasets import load_earth_mask
>>> # Load the default grid (gridline-registered 1 arc-degree grid)
>>> grid = load_earth_mask()
>>> # location (120°E, 50°N) is in land area (1)
>>> grid.sel(lon=120, lat=50).values
array(1, dtype=int8)
>>> # location (170°E, 50°N) is in oceanic area (0)
>>> grid.sel(lon=170, lat=50).values
array(0, dtype=int8)
```

8.9.11 pygmt.datasets.load_earth_mean_dynamic_topography

pygmt.datasets.**load_earth_mean_dynamic_topography**(*resolution='01d'*, *region=None*, *registration='gridline'*)

Load the CNES Earth mean dynamic topography dataset in various resolutions.

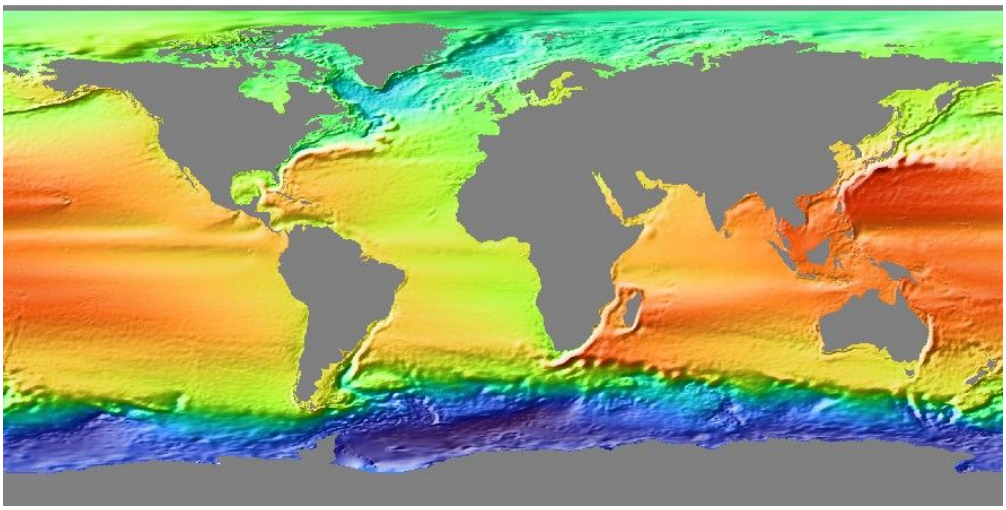


Fig. 8: CNES Earth mean dynamic topography dataset.

This function downloads the dataset from the GMT data server, caches it in a user data directory (usually `~/.gmt/server/earth/earth_mdt/`), and load the dataset as an `xarray.DataArray`. An internet connection is required the first time around, but subsequent calls will load the dataset from the local data directory.

The dataset can also be accessed by specifying a file name in any grid processing function or plotting method, using the following file name format: **@earth_mdt**_*res_reg*. *res* is the grid resolution; *reg* is the grid registration type (**p** for pixel registration, **g** for gridline registration). If *reg* is omitted (e.g., `@earth_mdt_01d`), the gridline-registered grid will be loaded for grid processing functions and the pixel-registered grid will be loaded for plotting functions. If *res* is also omitted (i.e., `@earth_mdt`), GMT automatically selects a suitable resolution based on the current region and projection settings.

This dataset comes with a color palette table (CPT) file, `@earth_mdt.cpt`. To use the dataset-specific CPT when plotting the dataset, explicitly set `cmap="@earth_mdt.cpt"`, otherwise GMT's default CPT (*turbo*) will be used. If the dataset is referenced by the file name in a grid plotting method, the dataset-specific CPT file is used automatically unless another CPT is specified.

Refer to https://www.generic-mapping-tools.org/remote-datasets/earth-mdt.html for more details about available datasets, including version information and references.

Parameters

- **resolution** (`Literal['01d', '30m', '20m', '15m', '10m', '07m']`, default: `'01d'`) – The grid resolution. The suffix `d` and `m` stand for arc-degrees and arc-minutes. Note that `"07m"` refers to a resolution of 7.5 arc-minutes.
- **region** (`Sequence[float] | str | None`, default: `None`) – The subregion of the grid to load, in the form of a sequence [*xmin*, *xmax*, *ymin*, *ymax*] or an ISO country code.
- **registration** (`Literal['gridline', 'pixel']`, default: `'gridline'`) – Grid registration type. Either `"pixel"` for pixel registration or `"gridline"` for gridline registration.

Return type

`DataArray`

Returns

grid – The CNES Earth mean dynamic topography grid. Coordinates are latitude and longitude in degrees. Values are in meters.

Note

The registration and coordinate system type of the returned `xarray.DataArray` grid can be accessed via the GMT accessors (i.e., `grid.gmt.registration` and `grid.gmt.gtype` respectively). However, these properties may be lost after specific grid operations (such as slicing) and will need to be manually set before passing the grid to any PyGMT data processing or plotting functions. Refer to *`pygmt.GMTDataArrayAccessor`* for detailed explanations and workarounds.

Examples

```
>>> from pygmt.datasets import load_earth_mean_dynamic_topography
>>> # Load the default grid (gridline-registered 1 arc-degree grid)
>>> grid = load_earth_mean_dynamic_topography()
>>> # Load the 30 arc-minutes grid with "gridline" registration
>>> grid = load_earth_mean_dynamic_topography(
...     resolution="30m", registration="gridline"
... )
>>> # Load high-resolution (7 arc-minutes) grid for a specific region
>>> grid = load_earth_mean_dynamic_topography(
...     resolution="07m",
...     region=[120, 160, 30, 60],
...     registration="gridline",
... )
```

8.9.12 pygmt.datasets.load_earth_mean_sea_surface

`pygmt.datasets.`**`load_earth_mean_sea_surface`**(*resolution='01d'*, *region=None*, *registration='gridline'*)

Load the CNES Earth mean sea surface dataset in various resolutions.

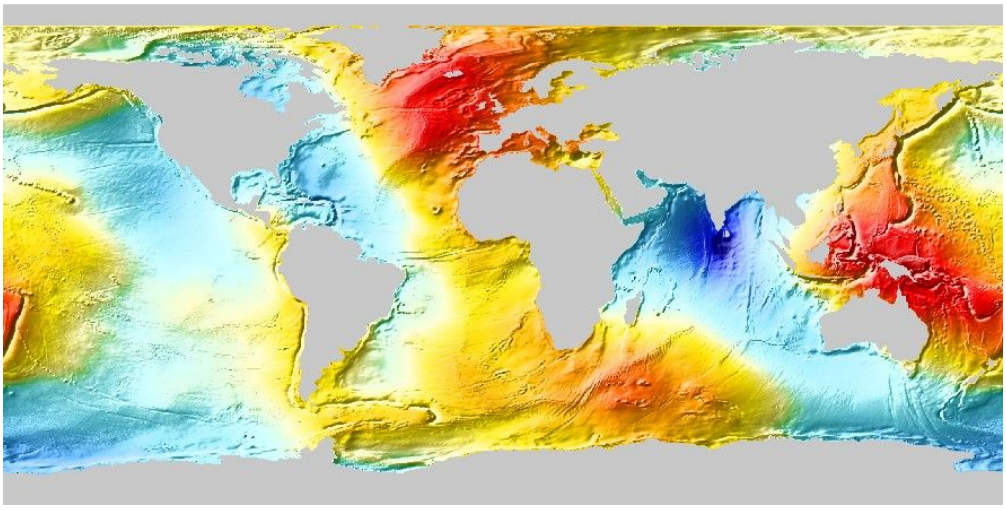


Fig. 9: CNES Earth mean sea surface dataset.

This function downloads the dataset from the GMT data server, caches it in a user data directory (usually `~/.gmt/server/earth/earth_mss/`), and load the dataset as an `xarray.DataArray`. An internet connection is required the first time around, but subsequent calls will load the dataset from the local data directory.

The dataset can also be accessed by specifying a file name in any grid processing function or plotting method, using the following file name format: **@earth_mss**_*res*_*reg*. *res* is the grid resolution; *reg* is the grid registration type (**p** for pixel registration, **g** for gridline registration). If *reg* is omitted (e.g., `@earth_mss_01d`), the gridline-registered grid will be loaded for grid processing functions and the pixel-registered grid will be loaded for plotting functions. If *res* is also omitted (i.e., `@earth_mss`), GMT automatically selects a suitable resolution based on the current region and projection settings.

This dataset comes with a color palette table (CPT) file, `@earth_mss.cpt`. To use the dataset-specific CPT when plotting the dataset, explicitly set `cmap="@earth_mss.cpt"`, otherwise GMT's default CPT (*turbo*) will be used. If the dataset is referenced by the file name in a grid plotting method, the dataset-specific CPT file is used automatically unless another CPT is specified.

Refer to https://www.generic-mapping-tools.org/remote-datasets/earth-mss.html for more details about available datasets, including version information and references.

Parameters

- **resolution** (`Literal['01d', '30m', '20m', '15m', '10m', '06m', '05m', '04m', '03m', '02m', '01m']`, default: `'01d'`) – The grid resolution. The suffix `d` and `m` stand for arc-degrees and arc-minutes.
- **region** (`Sequence[float] | str | None`, default: `None`) – The subregion of the grid to load, in the form of a sequence [*xmin*, *xmax*, *ymin*, *ymax*] or an ISO country code. Required for grids with resolutions higher than 5 arc-minutes (i.e., `"05m"`).
- **registration** (`Literal['gridline', 'pixel']`, default: `'gridline'`) – Grid registration type. Either `"pixel"` for pixel registration or `"gridline"` for gridline registration.

Return type

`DataArray`

Returns

grid – The CNES Earth mean sea surface grid. Coordinates are latitude and longitude in degrees. Values are in meters.

Note

The registration and coordinate system type of the returned `xarray.DataArray` grid can be accessed via the GMT accessors (i.e., `grid.gmt.registration` and `grid.gmt.gtype` respectively). However, these properties may be lost after specific grid operations (such as slicing) and will need to be manually set before passing the grid to any PyGMT data processing or plotting functions. Refer to *pygmt.GMTDataArrayAccessor* for detailed explanations and workarounds.

Examples

```
>>> from pygmt.datasets import load_earth_mean_sea_surface
>>> # Load the default grid (gridline-registered 1 arc-degree grid)
>>> grid = load_earth_mean_sea_surface()
>>> # Load the 30 arc-minutes grid with "gridline" registration
>>> grid = load_earth_mean_sea_surface(resolution="30m", registration="gridline")
>>> # Load high-resolution (5 arc-minutes) grid for a specific region
>>> grid = load_earth_mean_sea_surface(
```

(continues on next page)

(continued from previous page)

```
...     resolution="05m",
...     region=[120, 160, 30, 60],
...     registration="gridline",
... )
```

8.9.13 pygmt.datasets.load_earth_relief

pygmt.datasets.**load_earth_relief**(*resolution='01d'*, *region=None*, *registration=None*, *data_source='igpp'*, *use_srtm=False*)

Load the Earth relief datasets (topography and bathymetry) in various resolutions.

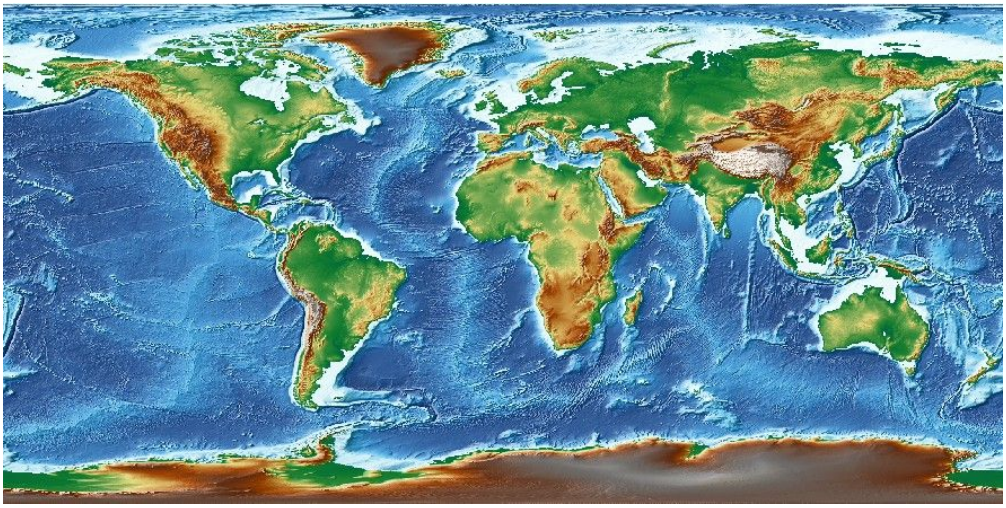


Fig. 10: Earth relief datasets (topography and bathymetry).

This function downloads the dataset from the GMT data server, caches it in a user data directory (usually `~/.gmt/server/earth/earth_relief`, `~/.gmt/server/earth/earth_gebco`, `~/.gmt/server/earth/earth_gebcosi`, `~/.gmt/server/earth/earth_synbath`), and load the dataset as an `xarray.DataArray`. An internet connection is required the first time around, but subsequent calls will load the dataset from the local data directory.

The dataset can also be accessed by specifying a file name in any grid processing function or plotting method, using the following file name format: *@earth_relief_type_res_reg*. *earth_relief_type* is the GMT name for the dataset. The available options are **earth_relief**, **earth_gebco**, **earth_gebcosi**, and **earth_synbath**. *res* is the grid resolution; *reg* is the grid registration type (**p** for pixel registration, **g** for gridline registration). If *reg* is omitted (e.g., `@earth_relief_01d`), the gridline-registered grid will be loaded for grid processing functions and the pixel-registered grid willcbe loaded for plotting functions. If *res* is also omitted (i.e., `@earth_relief`), GMT automatically selects a suitable resolution based on the current region and projection settings.

This dataset comes with a color palette table (CPT) file, `geo`. To use the dataset-specific CPT when plotting the dataset, explicitly set `cmap="geo"`, otherwise GMT's default CPT (*turbo*) will be used. If the dataset is referenced by the file name in a grid plotting method, the dataset-specific CPT file is used automatically unless another CPT is specified.

Refer to https://www.generic-mapping-tools.org/remote-datasets/earth-relief.html for more details about available datasets, including version information and references.

Parameters

- **resolution** (Literal['01d', '30m', '20m', '15m', '10m', '06m', '05m', '04m', '03m', '02m', '01m', '30s', '15s', '03s', '01s'], default: '01d') – The grid resolution. The suffix d, m and s stand for arc-degrees, arc-minutes, and arc-seconds.
- **region** (Sequence[float] | str | None, default: None) – The subregion of the grid to load, in the form of a sequence [*xmin*, *xmax*, *ymin*, *ymax*] or an ISO country code. Required for grids with resolutions higher than 5 arc-minutes (i.e., "05m").
- **registration** (Literal['gridline', 'pixel', None], default: None) – Grid registration type. Either "pixel" for pixel registration or "gridline" for gridline registration. Default is None, which means "gridline" for all resolutions except "15s" which is "pixel" only.
- **data_source** (Literal['igpp', 'gebco', 'gebcosi', 'synbath'], default: 'igpp') – Select the source for the Earth relief data. Available options are:
  - "igpp": IGPP Earth Relief. See https://www.generic-mapping-tools.org/remote-datasets/earth-relief.html.
  - "synbath": IGPP Earth Relief dataset that uses stastical properties of young seafloor to provide a more realistic relief of young areas with small seamounts.
  - "gebco": GEBCO Earth Relief with only observed relief and inferred relief via altimetric gravity. See https://www.generic-mapping-tools.org/remote-datasets/earth-gebco.html.
  - "gebcosi": GEBCO Earth Relief that gives sub-ice (si) elevations.
- **use_srtm** (bool, default: False) – 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 upsampling the SRTM15 tiles which have a resolution of 15 arc-seconds (i.e., "15s"). If True, will only load the original land-only SRTM tiles. Only works when data_source="igpp".

Return type

DataArray

Returns

grid – The Earth relief grid. Coordinates are latitude and longitude in degrees. Relief is in meters.

Note

The registration and coordinate system type of the returned xarray.DataArray grid can be accessed via the GMT accessors (i.e., grid.gmt.registration and grid.gmt.gtype respectively). However, these properties may be lost after specific grid operations (such as slicing) and will need to be manually set before passing the grid to any PyGMT data processing or plotting functions. Refer to *pygmt.GMTDataArrayAccessor* for detailed explanations and workarounds.

Examples

```
>>> from pygmt.datasets import load_earth_relief
>>> # Load the default grid (gridline-registered 1 arc-degree grid)
>>> grid = load_earth_relief()
>>> # Load the 30 arc-minutes grid with "gridline" registration
>>> grid = load_earth_relief(resolution="30m", registration="gridline")
>>> # Load high-resolution (5 arc-minutes) grid for a specific region
>>> grid = load_earth_relief(
...     resolution="05m",
...     region=[120, 160, 30, 60],
```

(continues on next page)

(continued from previous page)

```
...     registration="gridline",
... )
>>> # Load the original 3 arc-seconds land-only SRTM tiles from NASA
>>> grid = load_earth_relief(
...     resolution="03s",
...     region=[135, 136, 35, 36],
...     registration="gridline",
...     use_srtm=True,
... )
```

Examples using `pygmt.datasets.load_earth_relief`

8.9.14 pygmt.datasets.load_earth_vertical_gravity_gradient

`pygmt.datasets.`**`load_earth_vertical_gravity_gradient`**(*resolution='01d'*, *region=None*, *registration=None*)

Load the IGPP Earth vertical gravity gradient dataset in various resolutions.

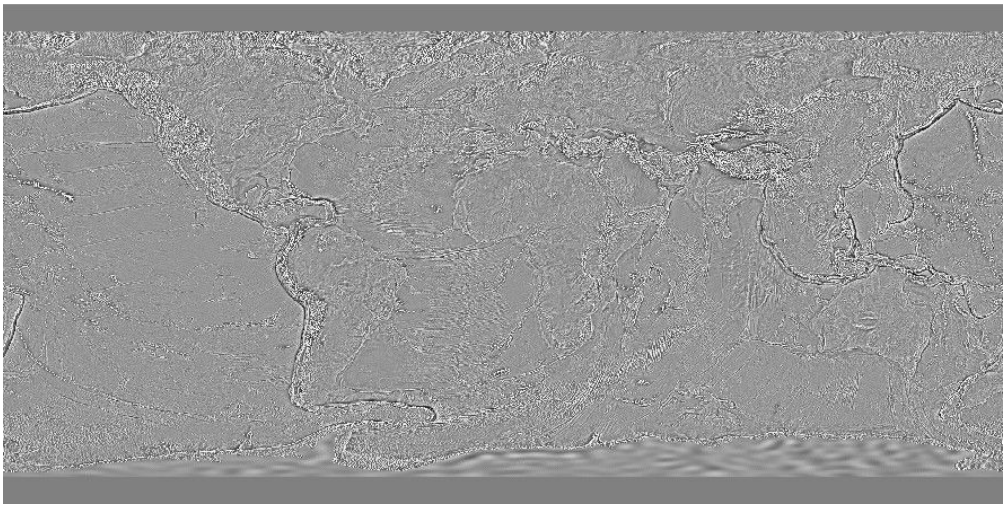


Fig. 11: IGPP Earth vertical gravity gradient dataset.

This function downloads the dataset from the GMT data server, caches it in a user data directory (usually `~/.gmt/server/earth/earth_vgg/`), and load the dataset as an `xarray.DataArray`. An internet connection is required the first time around, but subsequent calls will load the dataset from the local data directory.

The dataset can also be accessed by specifying a file name in any grid processing function or plotting method, using the following file name format: **@earth_vgg**_*res*_*reg*. *res* is the grid resolution; *reg* is the grid registration type (**p** for pixel registration, **g** for gridline registration). If *reg* is omitted (e.g., `@earth_vgg_01d`), the gridline-registered grid will be loaded for grid processing functions and the pixel-registered grid will be loaded for plotting functions. If *res* is also omitted (i.e., `@earth_vgg`), GMT automatically selects a suitable resolution based on the current region and projection settings.

This dataset comes with a color palette table (CPT) file, `@earth_vgg.cpt`. To use the dataset-specific CPT when plotting the dataset, explicitly set `cmap="@earth_vgg.cpt"`, otherwise GMT's default CPT (*turbo*) will be used. If the dataset is referenced by the file name in a grid plotting method, the dataset-specific CPT file is used automatically unless another CPT is specified.

Refer to https://www.generic-mapping-tools.org/remote-datasets/earth-vgg.html for more details about available

datasets, including version information and references.

Parameters

- **resolution** (Literal['01d', '30m', '20m', '15m', '10m', '06m', '05m', '04m', '03m', '02m', '01m'], default: '01d') – The grid resolution. The suffix d and m stand for arc-degrees and arc-minutes.
- **region** (Sequence[float] | str | None, default: None) – The subregion of the grid to load, in the form of a sequence [*xmin*, *xmax*, *ymin*, *ymax*] or an ISO country code. Required for grids with resolutions higher than 5 arc-minutes (i.e., "05m").
- **registration** (Literal['gridline', 'pixel', None], default: None) – Grid registration type. Either "pixel" for pixel registration or "gridline" for gridline registration. Default is None, which means "gridline" for all resolutions except "01m" which is "pixel" only.

Return type

DataArray

Returns

grid – The Earth vertical gravity gradient grid. Coordinates are latitude and longitude in degrees. Units are in Eotvos.

Note

The registration and coordinate system type of the returned xarray.DataArray grid can be accessed via the GMT accessors (i.e., grid.gmt.registration and grid.gmt.gtype respectively). However, these properties may be lost after specific grid operations (such as slicing) and will need to be manually set before passing the grid to any PyGMT data processing or plotting functions. Refer to *pygmt.GMTDataArrayAccessor* for detailed explanations and workarounds.

Examples

```
>>> from pygmt.datasets import load_earth_vertical_gravity_gradient
>>> # Load the default grid (gridline-registered 1 arc-degree grid)
>>> grid = load_earth_vertical_gravity_gradient()
>>> # Load the 30 arc-minutes grid with "gridline" registration
>>> grid = load_earth_vertical_gravity_gradient(
...     resolution="30m", registration="gridline"
... )
>>> # Load high-resolution (5 arc-minutes) grid for a specific region
>>> grid = load_earth_vertical_gravity_gradient(
...     resolution="05m",
...     region=[120, 160, 30, 60],
...     registration="gridline",
... )
```

8.9.15 pygmt.datasets.load_mars_relief

pygmt.datasets.**load_mars_relief**(*resolution='01d'*, *region=None*, *registration=None*)

Load the Mars relief dataset in various resolutions.

This function downloads the dataset from the GMT data server, caches it in a user data directory (usually ~/.gmt/server/mars/mars_relief/), and load the dataset as an xarray.DataArray. An internet connection is required the first time around, but subsequent calls will load the dataset from the local data directory.

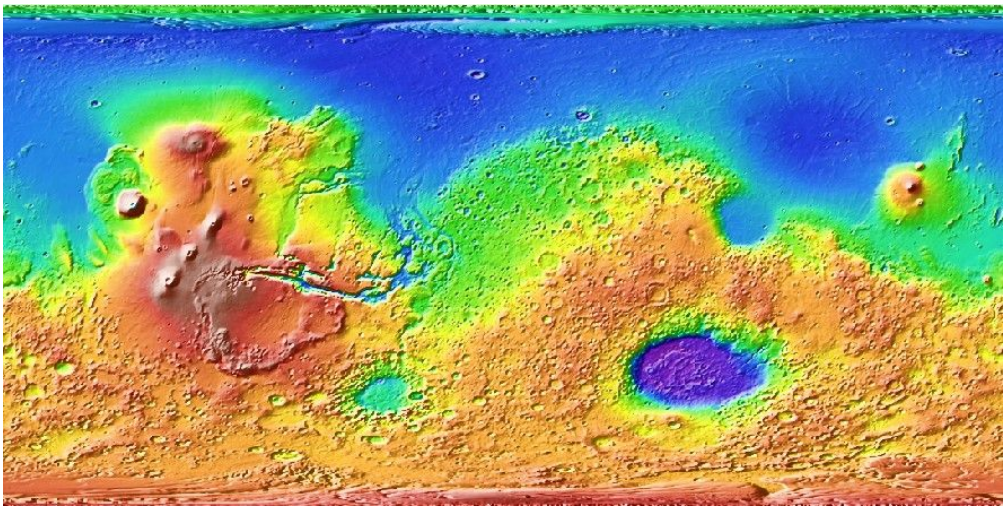


Fig. 12: Mars relief dataset.

The dataset can also be accessed by specifying a file name in any grid processing function or plotting method, using the following file name format: **@mars_relief**_*res_reg*. *res* is the grid resolution; *reg* is the grid registration type (**p** for pixel registration, **g** for gridline registration). If *reg* is omitted (e.g., `@mars_relief_01d`), the gridline-registered grid will be loaded for grid processing functions and the pixel-registered grid will be loaded for plotting functions. If *res* is also omitted (i.e., `@mars_relief`), GMT automatically selects a suitable resolution based on the current region and projection settings.

This dataset comes with a color palette table (CPT) file, `@mars_relief.cpt`. To use the dataset-specific CPT when plotting the dataset, explicitly set `cmap="@mars_relief.cpt"`, otherwise GMT's default CPT (*turbo*) will be used. If the dataset is referenced by the file name in a grid plotting method, the dataset-specific CPT file is used automatically unless another CPT is specified.

Refer to https://www.generic-mapping-tools.org/remote-datasets/mars-relief.html for more details about available datasets, including version information and references.

Parameters

- **resolution** (`Literal['01d', '30m', '20m', '15m', '10m', '06m', '05m', '04m', '03m', '02m', '01m', '30s', '15s', '12s']`, default: `'01d'`) – The grid resolution. The suffix `d`, `m` and `s` stand for arc-degrees, arc-minutes and arc-seconds. Note that `"12s"` refers to a resolution of 12.1468873601 arc-seconds.
- **region** (`Sequence[float] | str | None`, default: `None`) – The subregion of the grid to load, in the form of a sequence [*xmin*, *xmax*, *ymin*, *ymax*] or an ISO country code. Required for grids with resolutions higher than 5 arc-minutes (i.e., `"05m"`).
- **registration** (`Literal['gridline', 'pixel', None]`, default: `None`) – Grid registration type. Either `"pixel"` for pixel registration or `"gridline"` for gridline registration. Default is `None`, which means `"gridline"` for all resolutions except for `"12s"` which is `"pixel"` only.

Return type

`DataArray`

Returns

grid – The Mars relief grid. Coordinates are latitude and longitude in degrees. Relief is in meters.

Note

The registration and coordinate system type of the returned `xarray.DataArray` grid can be accessed via the GMT accessors (i.e., `grid.gmt.registration` and `grid.gmt.gtype` respectively). However, these properties may be lost after specific grid operations (such as slicing) and will need to be manually set before passing the grid to any PyGMT data processing or plotting functions. Refer to *pygmt.GMTDataArrayAccessor* for detailed explanations and workarounds.

Examples

```
>>> from pygmt.datasets import load_mars_relief
>>> # Load the default grid (gridline-registered 1 arc-degree grid)
>>> grid = load_mars_relief()
>>> # Load the 30 arc-minutes grid with "gridline" registration
>>> grid = load_mars_relief(resolution="30m", registration="gridline")
>>> # Load high-resolution (5 arc-minutes) grid for a specific region
>>> grid = load_mars_relief(
...     resolution="05m",
...     region=[120, 160, 30, 60],
...     registration="gridline",
... )
```

8.9.16 pygmt.datasets.load_mercury_relief

`pygmt.datasets.`**`load_mercury_relief`**(*resolution='01d'*, *region=None*, *registration=None*)

Load the Mercury relief dataset in various resolutions.

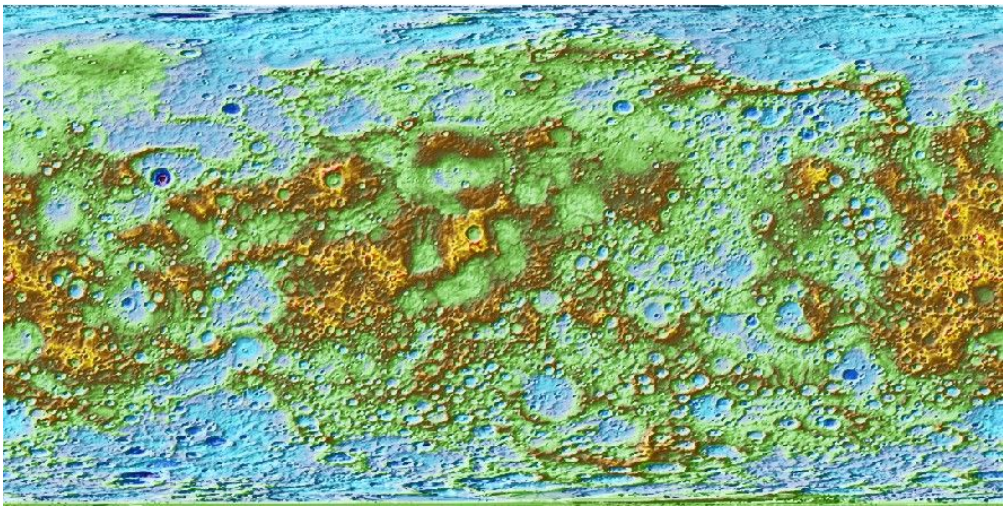


Fig. 13: Mercury relief dataset.

This function downloads the dataset from the GMT data server, caches it in a user data directory (usually `~/.gmt/server/mercury/mercury_relief/`), and load the dataset as an `xarray.DataArray`. An internet connection is required the first time around, but subsequent calls will load the dataset from the local data directory.

The dataset can also be accessed by specifying a file name in any grid processing function or plotting method, using the following file name format: **@mercury_relief**_*res_reg*. *res* is the grid resolution; *reg* is the grid registration type (**p** for pixel registration, **g** for gridline registration). If *reg* is omitted (e.g., `@mercury_relief_01d`), the

gridline-registered grid will be loaded for grid processing functions and the pixel-registered grid will be loaded for plotting functions. If *res* is also omitted (i.e., `@mercury_relief`), GMT automatically selects a suitable resolution based on the current region and projection settings.

This dataset comes with a color palette table (CPT) file, `@mercury_relief.cpt`. To use the dataset-specific CPT when plotting the dataset, explicitly set `cmap="@mercury_relief.cpt"`, otherwise GMT's default CPT (*turbo*) will be used. If the dataset is referenced by the file name in a grid plotting method, the dataset-specific CPT file is used automatically unless another CPT is specified.

Refer to https://www.generic-mapping-tools.org/remote-datasets/mercury-relief.html for more details about available datasets, including version information and references.

Parameters

- **resolution** (`Literal['01d', '30m', '20m', '15m', '10m', '06m', '05m', '04m', '03m', '02m', '01m', '56s']`, default: `'01d'`) – The grid resolution. The suffix `d`, `m` and `s` stand for arc-degrees, arc-minutes and arc-seconds. Note that `"56s"` refers to a resolution of 56.25 arc-seconds.
- **region** (`Sequence[float] | str | None`, default: `None`) – The subregion of the grid to load, in the form of a sequence [*xmin*, *xmax*, *ymin*, *ymax*] or an ISO country code. Required for grids with resolutions higher than 5 arc-minutes (i.e., `"05m"`).
- **registration** (`Literal['gridline', 'pixel', None]`, default: `None`) – Grid registration type. Either `"pixel"` for pixel registration or `"gridline"` for gridline registration. Default is `None`, which means `"gridline"` for all resolutions except for `"56s"` which is `"pixel"` only.

Return type

`DataArray`

Returns

grid – The Mercury relief grid. Coordinates are latitude and longitude in degrees. Relief is in meters.

Note

The registration and coordinate system type of the returned `xarray.DataArray` grid can be accessed via the GMT accessors (i.e., `grid.gmt.registration` and `grid.gmt.gtype` respectively). However, these properties may be lost after specific grid operations (such as slicing) and will need to be manually set before passing the grid to any PyGMT data processing or plotting functions. Refer to *pygmt.GMTDataArrayAccessor* for detailed explanations and workarounds.

Examples

```
>>> from pygmt.datasets import load_mercury_relief
>>> # Load the default grid (gridline-registered 1 arc-degree grid)
>>> grid = load_mercury_relief()
>>> # Load the 30 arc-minutes grid with "gridline" registration
>>> grid = load_mercury_relief(resolution="30m", registration="gridline")
>>> # Load high-resolution (5 arc-minutes) grid for a specific region
>>> grid = load_mercury_relief(
...     resolution="05m",
...     region=[120, 160, 30, 60],
...     registration="gridline",
... )
```

8.9.17 pygmt.datasets.load_moon_relief

pygmt.datasets.**load_moon_relief**(*resolution='01d'*, *region=None*, *registration=None*)

Load the Moon relief dataset in various resolutions.

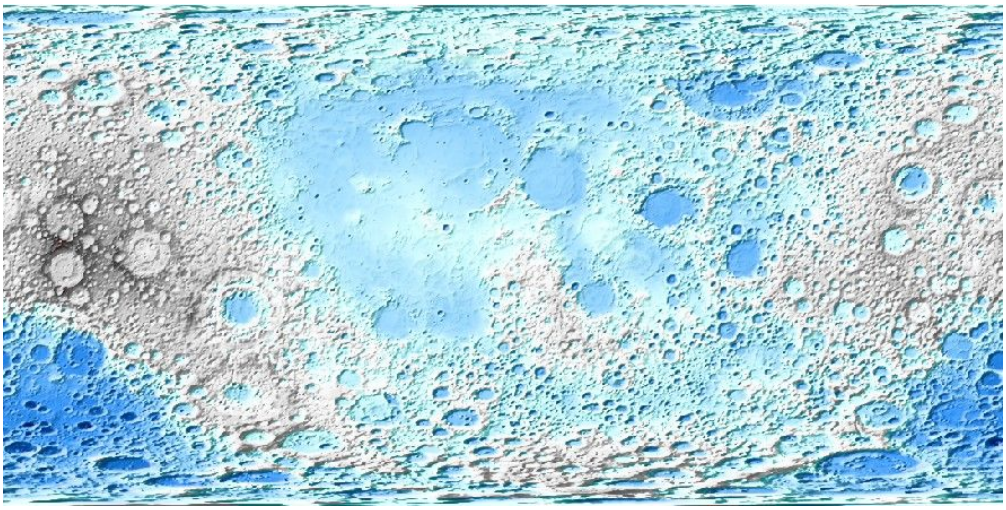


Fig. 14: Moon relief dataset.

This function downloads the dataset from the GMT data server, caches it in a user data directory (usually `~/.gmt/server/moon/moon_relief/`), and load the dataset as an `xarray.DataArray`. An internet connection is required the first time around, but subsequent calls will load the dataset from the local data directory.

The dataset can also be accessed by specifying a file name in any grid processing function or plotting method, using the following file name format: **@moon_relief**_*res_reg*. *res* is the grid resolution; *reg* is the grid registration type (**p** for pixel registration, **g** for gridline registration). If *reg* is omitted (e.g., `@moon_relief_01d`), the gridline-registered grid will be loaded for grid processing functions and the pixel-registered grid will be loaded for plotting functions. If *res* is also omitted (i.e., `@moon_relief`), GMT automatically selects a suitable resolution based on the current region and projection settings.

This dataset comes with a color palette table (CPT) file, `@moon_relief.cpt`. To use the dataset-specific CPT when plotting the dataset, explicitly set `cmap="@moon_relief.cpt"`, otherwise GMT's default CPT (*turbo*) will be used. If the dataset is referenced by the file name in a grid plotting method, the dataset-specific CPT file is used automatically unless another CPT is specified.

Refer to https://www.generic-mapping-tools.org/remote-datasets/moon-relief.html for more details about available datasets, including version information and references.

Parameters

- **resolution** (`Literal['01d', '30m', '20m', '15m', '10m', '06m', '05m', '04m', '03m', '02m', '01m', '30s', '15s', '14s']`, default: `'01d'`) – The grid resolution. The suffix `d`, `m` and `s` stand for arc-degrees, arc-minutes and arc-seconds. Note that `"14s"` refers to a resolution of 14.0625 arc-seconds.
- **region** (`Sequence[float] | str | None`, default: `None`) – The subregion of the grid to load, in the form of a sequence [*xmin*, *xmax*, *ymin*, *ymax*] or an ISO country code. Required for grids with resolutions higher than 5 arc-minutes (i.e., `"05m"`).
- **registration** (`Literal['gridline', 'pixel', None]`, default: `None`) – Grid registration type. Either `"pixel"` for pixel registration or `"gridline"` for gridline registration. Default is `None`, which means `"gridline"` for all resolutions except for `"14s"` which is `"pixel"` only.

Return type
DataArray

Returns
grid – The Moon relief grid. Coordinates are latitude and longitude in degrees. Relief is in meters.

Note

The registration and coordinate system type of the returned `xarray.DataArray` grid can be accessed via the GMT accessors (i.e., `grid.gmt.registration` and `grid.gmt.gtype` respectively). However, these properties may be lost after specific grid operations (such as slicing) and will need to be manually set before passing the grid to any PyGMT data processing or plotting functions. Refer to *pygmt.GMTDataArrayAccessor* for detailed explanations and workarounds.

Examples

```
>>> from pygmt.datasets import load_moon_relief
>>> # Load the default grid (gridline-registered 1 arc-degree grid)
>>> grid = load_moon_relief()
>>> # Load the 30 arc-minutes grid with "gridline" registration
>>> grid = load_moon_relief(resolution="30m", registration="gridline")
>>> # Load high-resolution (5 arc-minutes) grid for a specific region
>>> grid = load_moon_relief(
...     resolution="05m",
...     region=[120, 160, 30, 60],
...     registration="gridline",
... )
```

8.9.18 pygmt.datasets.load_pluto_relief

pygmt.datasets.**load_pluto_relief**(*resolution='01d'*, *region=None*, *registration=None*)

Load the Pluto relief dataset in various resolutions.

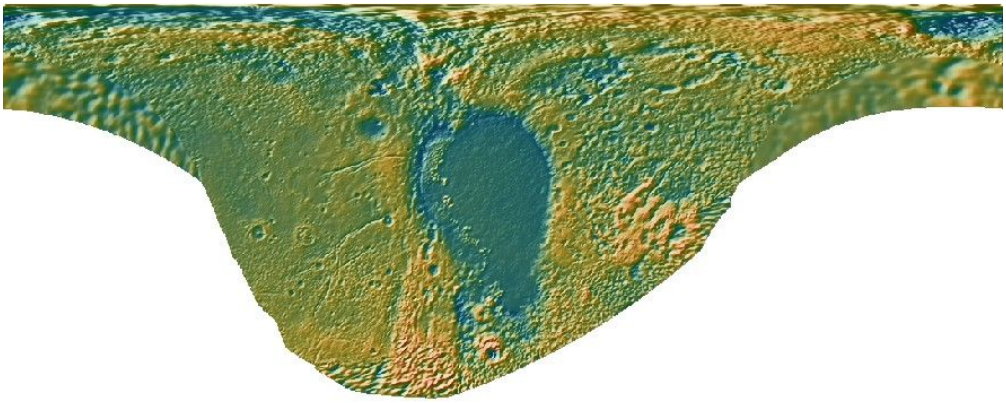


Fig. 15: Pluto relief dataset.

This function downloads the dataset from the GMT data server, caches it in a user data directory (usually `~/.gmt/`

server/pluto/pluto_relief/), and load the dataset as an xarray.DataArray. An internet connection is required the first time around, but subsequent calls will load the dataset from the local data directory.

The dataset can also be accessed by specifying a file name in any grid processing function or plotting method, using the following file name format: **@pluto_relief**_*res*_*reg*. *res* is the grid resolution; *reg* is the grid registration type (**p** for pixel registration, **g** for gridline registration). If *reg* is omitted (e.g., @pluto_relief_01d), the gridline-registered grid will be loaded for grid processing functions and the pixel-registered grid will be loaded for plotting functions. If *res* is also omitted (i.e., @pluto_relief), GMT automatically selects a suitable resolution based on the current region and projection settings.

This dataset comes with a color palette table (CPT) file, @pluto_relief.cpt. To use the dataset-specific CPT when plotting the dataset, explicitly set cmap="@pluto_relief.cpt", otherwise GMT's default CPT (*turbo*) will be used. If the dataset is referenced by the file name in a grid plotting method, the dataset-specific CPT file is used automatically unless another CPT is specified.

Refer to https://www.generic-mapping-tools.org/remote-datasets/pluto-relief.html for more details about available datasets, including version information and references.

Parameters

- **resolution** (Literal['01d', '30m', '20m', '15m', '10m', '06m', '05m', '04m', '03m', '02m', '01m', '52s'], default: '01d') – The grid resolution. The suffix d, m and s stand for arc-degrees, arc-minutes and arc-seconds. Note that "52s" refers to a resolution of 52.0732883317 arc-seconds.
- **region** (Sequence[float] | str | None, default: None) – The subregion of the grid to load, in the form of a sequence [*xmin*, *xmax*, *ymin*, *ymax*] or an ISO country code. Required for grids with resolutions higher than 5 arc-minutes (i.e., "05m").
- **registration** (Literal['gridline', 'pixel', None], default: None) – Grid registration type. Either "pixel" for pixel registration or "gridline" for gridline registration. Default is None, which means "gridline" for all resolutions except for "52s" which is "pixel" only.

Return type

DataArray

Returns

grid – The Pluto relief grid. Coordinates are latitude and longitude in degrees. Relief is in meters.

Note

The registration and coordinate system type of the returned xarray.DataArray grid can be accessed via the GMT accessors (i.e., grid.gmt.registration and grid.gmt.gtype respectively). However, these properties may be lost after specific grid operations (such as slicing) and will need to be manually set before passing the grid to any PyGMT data processing or plotting functions. Refer to *pygmt.GMTDataArrayAccessor* for detailed explanations and workarounds.

Examples

```
>>> from pygmt.datasets import load_pluto_relief
>>> # Load the default grid (gridline-registered 1 arc-degree grid)
>>> grid = load_pluto_relief()
>>> # Load the 30 arc-minutes grid with "gridline" registration
>>> grid = load_pluto_relief(resolution="30m", registration="gridline")
>>> # Load high-resolution (5 arc-minutes) grid for a specific region
```

(continues on next page)

(continued from previous page)

```
>>> grid = load_pluto_relief(
...     resolution="05m",
...     region=[120, 160, 30, 60],
...     registration="gridline",
... )
```

8.9.19 pygmt.datasets.load_venus_relief

pygmt.datasets.**load_venus_relief**(*resolution='01d'*, *region=None*, *registration='gridline'*)

Load the Venus relief dataset in various resolutions.

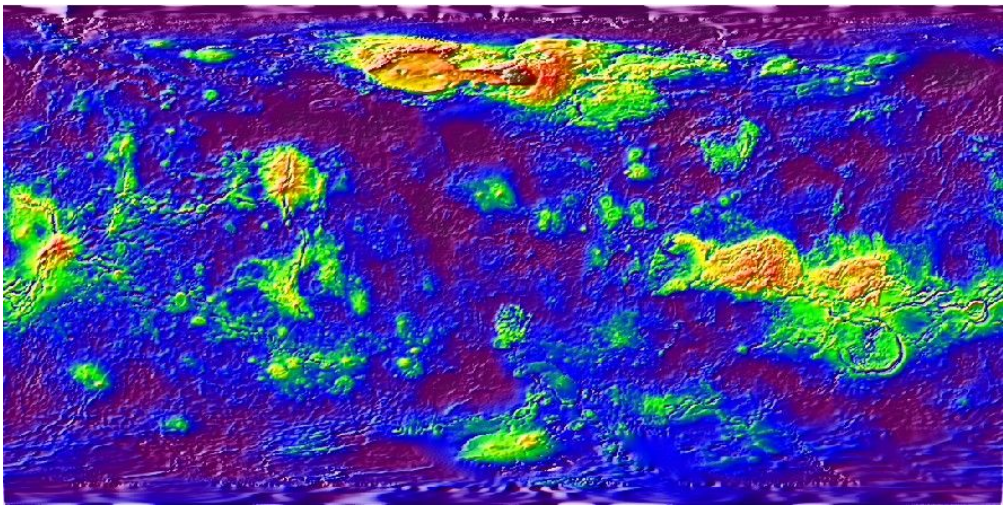


Fig. 16: Venus relief dataset.

This function downloads the dataset from the GMT data server, caches it in a user data directory (usually `~/.gmt/server/venus/venus_relief/`), and load the dataset as an `xarray.DataArray`. An internet connection is required the first time around, but subsequent calls will load the dataset from the local data directory.

The dataset can also be accessed by specifying a file name in any grid processing function or plotting method, using the following file name format: **@venus_relief**_*res_reg*. *res* is the grid resolution; *reg* is the grid registration type (**p** for pixel registration, **g** for gridline registration). If *reg* is omitted (e.g., `@venus_relief_01d`), the gridline-registered grid will be loaded for grid processing functions and the pixel-registered grid will be loaded for plotting functions. If *res* is also omitted (i.e., `@venus_relief`), GMT automatically selects a suitable resolution based on the current region and projection settings.

This dataset comes with a color palette table (CPT) file, `@venus_relief.cpt`. To use the dataset-specific CPT when plotting the dataset, explicitly set `cmap="@venus_relief.cpt"`, otherwise GMT's default CPT (*turbo*) will be used. If the dataset is referenced by the file name in a grid plotting method, the dataset-specific CPT file is used automatically unless another CPT is specified.

Refer to https://www.generic-mapping-tools.org/remote-datasets/venus-relief.html for more details about available datasets, including version information and references.

Parameters

- **resolution** (`Literal['01d', '30m', '20m', '15m', '10m', '06m', '05m', '04m', '03m', '02m', '01m']`, default: `'01d'`) – The grid resolution. The suffix `d` and `m` stand for arc-degrees and arc-minutes.

- **region** (Sequence[float] | str | None, default: None) – The subregion of the grid to load, in the form of a sequence [*xmin*, *xmax*, *ymin*, *ymax*] or an ISO country code. Required for grids with resolutions higher than 5 arc-minutes (i.e., "05m").
- **registration** (Literal['gridline', 'pixel'], default: 'gridline') – Grid registration type. Either "pixel" for pixel registration or "gridline" for gridline registration.

Return type
DataArray

Returns
grid – The Venus relief grid. Coordinates are latitude and longitude in degrees. Relief is in meters.

Note

The registration and coordinate system type of the returned xarray.DataArray grid can be accessed via the GMT accessors (i.e., grid.gmt.registration and grid.gmt.gtype respectively). However, these properties may be lost after specific grid operations (such as slicing) and will need to be manually set before passing the grid to any PyGMT data processing or plotting functions. Refer to *pygmt.GMTDataArrayAccessor* for detailed explanations and workarounds.

Examples

```
>>> from pygmt.datasets import load_venus_relief
>>> # Load the default grid (gridline-registered 1 arc-degree grid)
>>> grid = load_venus_relief()
>>> # Load the 30 arc-minutes grid with "gridline" registration
>>> grid = load_venus_relief(resolution="30m", registration="gridline")
>>> # Load high-resolution (5 arc-minutes) grid for a specific region
>>> grid = load_venus_relief(
...     resolution="05m",
...     region=[120, 160, 30, 60],
...     registration="gridline",
... )
```

8.9.20 pygmt.datasets.load_sample_data

pygmt.datasets.**load_sample_data**(*name*)

Load an example dataset from the GMT server.

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.

Parameters
name (Literal['bathymetry', 'earth_relief_holes', 'fractures', 'hotspots', 'japan_quakes', 'mars_shape', 'maunaloa_co2', 'notre_dame_topography', 'ocean_ridge_points', 'rock_compositions', 'usgs_quakes']) – Name of the dataset to load.

Return type
DataFrame | DataArray

Returns
data – Sample dataset loaded as a pandas.DataFrame for tabular data or xarray.DataArray for raster data.

See also

list_sample_data
Report datasets available for tests and documentation examples.

Examples

```
>>> from pprint import pprint
>>> from pygmt.datasets import list_sample_data, load_sample_data
>>> # use list_sample_data to see the available datasets
>>> pprint(list_sample_data(), width=120)
{'bathymetry': 'Table of ship bathymetric observations off Baja California',
 'earth_relief_holes': 'Regional 20 arc-minutes Earth relief grid with holes',
 'fractures': 'Table of hypothetical fracture lengths and azimuths',
 'hotspots': 'Table of locations, names, and symbol sizes of hotpots from Müller␣
↪et al. (1993)',
 'japan_quakes': 'Table of earthquakes around Japan from the NOAA NGDC database',
 'mars_shape': 'Table of topographic signature of the hemispheric dichotomy of␣
↪Mars from Smith and Zuber (1996)',
 'maunaloa_co2': 'Table of CO2 readings from Mauna Loa',
 'notre_dame_topography': 'Table 5.11 in Davis: Statistics and Data Analysis in␣
↪Geology',
 'ocean_ridge_points': 'Table of ocean ridge points for the entire world',
 'rock_compositions': 'Table of rock sample compositions',
 'usgs_quakes': 'Table of earthquakes from the USGS'}
>>> # Load the sample bathymetry dataset
>>> data = load_sample_data("bathymetry")
```

Examples using `pygmt.datasets.load_sample_data`

In addition, there is also a special function to load XYZ tile maps via contextily to be used as base maps.

datasets.load_tile_map(region[, zoom, ...])	Load a georeferenced raster tile map from XYZ tile providers.

8.9.21 pygmt.datasets.load_tile_map

pygmt.datasets.**load_tile_map**(*region*, *zoom='auto'*, *source=None*, *lonlat=True*, *crs='EPSG:3857'*, *wait=0*, *max_retries=2*, *zoom_adjust=None*)

Load a georeferenced raster tile map from XYZ tile providers.

The tiles that compose the map are merged and georeferenced into an `xarray.DataArray` image with 3 bands (RGB). Note that the returned image is in a Spherical Mercator (EPSG:3857) coordinate reference system (CRS) by default, but can be customized using the `crs` parameter.

Parameters

- **region** (`Sequence[float]`) – The bounding box of the map in the form of a list [*xmin*, *xmax*, *ymin*, *ymax*]. These coordinates should be in longitude/latitude if `lonlat=True` or Spherical Mercator (EPSG:3857) if `lonlat=False`.
- **zoom** (`Union[int, Literal['auto']]`, default: `'auto'`) – Level of detail. Higher levels (e.g. `22`) mean a zoom level closer to the Earth's surface, with more tiles covering a smaller

geographical area and thus more detail. Lower levels (e.g. `0`) mean a zoom level further from the Earth's surface, with less tiles covering a larger geographical area and thus less detail. Default is `"auto"` to automatically determine the zoom level based on the bounding box region extent.

> **Note**
>
> The maximum possible zoom level may be smaller than `22`, and depends on what is supported by the chosen web tile provider source.

- **source** (`TileProvider` | `str` | `None`, default: `None`) – The tile source: web tile provider or path to a local file. Provide either:
  - A web tile provider in the form of a `xyzservices.TileProvider` object. See Contextily providers for a list of tile providers. Default is `xyzservices.providers.OpenStreetMap.HOT`, i.e. OpenStreetMap Humanitarian web tiles.
  - A web tile provider in the form of a URL. The placeholders for the XYZ in the URL need to be {x}, {y}, {z}, respectively. E.g. `https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png`.
  - A local file path. The file is read with rasterio and all bands are loaded into the basemap. See Working with local files.

> **Important**
>
> Tiles are assumed to be in the Spherical Mercator projection (EPSG:3857).

- **lonlat** (`bool`, default: `True`) – If `False`, coordinates in `region` are assumed to be Spherical Mercator as opposed to longitude/latitude.
- **crs** (`str` | `CRS`, default: `'EPSG:3857'`) – Coordinate reference system (CRS) of the returned `xarray.DataArray` image. Default is `"EPSG:3857"` (i.e., Spherical Mercator). The CRS can be in either string or `rasterio.crs.CRS` format.
- **wait** (`int`, default: `0`) – If the tile API is rate-limited, the number of seconds to wait between a failed request and the next try.
- **max_retries** (`int`, default: `2`) – Total number of rejected requests allowed before contextily will stop trying to fetch more tiles from a rate-limited API.
- **zoom_adjust** (`int` | `None`, default: `None`) – The amount to adjust a chosen zoom level if it is chosen automatically. Values outside of -1 to 1 are not recommended as they can lead to slow execution.

Return type

`DataArray`

Returns

raster – Georeferenced 3-D data array of RGB values.

Raises

ImportError – If `contextily` is not installed or can't be imported. Follow the install instructions for contextily, (e.g. via `python -m pip install contextily`) before using this function.

Examples

```
>>> import contextily
>>> from pygmt.datasets import load_tile_map
>>> raster = load_tile_map(
...     region=[-180.0, 180.0, -90.0, 0.0],  # West, East, South, North
...     zoom=1,  # less detailed zoom level
...     source=contextily.providers.OpenTopoMap,
...     lonlat=True,  # bounding box coordinates are longitude/latitude
... )
>>> raster.sizes
Frozen({'band': 3, 'y': 256, 'x': 512})
>>> raster.coords
Coordinates:
  * band         (band) uint8... 1 2 3
  * y            (y) float64... -7.081e-10 -7.858e+04 ... -1.996e+07 -2.004e+07
  * x            (x) float64... -2.004e+07 -1.996e+07 ... 1.996e+07 2.004e+07
    spatial_ref  int... 0
>>> # CRS is set only if rioxarray is available
>>> if hasattr(raster, "rio"):
...     raster.rio.crs.to_string()
'EPSG:3857'
```

8.10 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.

8.10.1 pygmt.exceptions.GMTError

exception pygmt.exceptions.**GMTError**

Base class for all GMT related errors.

8.10.2 pygmt.exceptions.GMTInvalidInput

exception pygmt.exceptions.**GMTInvalidInput**

Raised when the input of a function/method is invalid.

8.10.3 pygmt.exceptions.GMTVersionError

exception pygmt.exceptions.**GMTVersionError**

Raised when an incompatible version of GMT is being used.

8.10.4 pygmt.exceptions.GMTOSError

exception pygmt.exceptions.**GMTOSError**

Unsupported operating system.

8.10.5 pygmt.exceptions.GMTCLibError

exception pygmt.exceptions.**GMTCLibError**

Error encountered when running a function from the GMT shared library.

8.10.6 pygmt.exceptions.GMTCLibNoSessionError

exception pygmt.exceptions.**GMTCLibNoSessionError**

Tried to access GMT API without a currently open GMT session.

8.10.7 pygmt.exceptions.GMTCLibNotFoundError

exception pygmt.exceptions.**GMTCLibNotFoundError**

Could not find the GMT shared library.

8.11 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.

8.11.1 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.

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.

Examples

```
>>> from pygmt.helpers.testing import load_static_earth_relief
>>> from pygmt.helpers import GMTTempFile
>>> grid = load_static_earth_relief()
>>> type(grid)
<class 'xarray.core.dataarray.DataArray'>
>>> # Create a session and destroy it automatically when exiting the "with" block.
>>> with Session() as lib:
...     # Create a virtual file and link to the memory block of the grid.
...     with lib.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.
...             lib.call_module("grdinfo", [fin, "-C", f"->{fout.name}"])
...             # Read the contents of the temp file before it's deleted.
...             print(fout.read().strip())
-55 -47 -24 -10 190 981 1 1 8 14 1 1
```

Attributes

property Session.**info**: **dict**[**str**, **str**]

Dictionary with the GMT version and default paths and parameters.

property Session.**session_pointer**: **c_void_p**

The `ctypes.c_void_p` pointer to the current open GMT session.

Raises

GMTCLibNoSessionError – If trying to access without a currently open GMT session (i.e., outside of the context manager).

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 and allocate space to hold data.
Session.destroy()	Destroy the currently open GMT API session.
Session.extract_region()	Extract the region of the currently active figure.
Session.get_common(option)	Inquire if a GMT common option has been set and return its current value if possible.
Session.get_default(name)	Get the value of a GMT configuration parameter or a GMT API parameter.
Session.get_enum(name)	Get the value of a GMT constant (C enum) from `gmt_resources.h`.
Session.get_libgmt_func(name[, argtypes, ...])	Get a ctypes function from the libgmt shared library.
Session.inquire_virtualfile(vfname)	Get the family of a virtual file.
Session.open_virtualfile(family, geometry, ...)	Open a GMT virtual file associated with a data object for reading or writing.

continues on next page

Table 27 – continued from previous page

Session.put_matrix(dataset, matrix[, pad])	Attach a 2-D numpy array to a GMT dataset.
Session.put_strings(dataset, family, strings)	Attach a 1-D numpy array of dtype str as a column on a GMT dataset.
Session.put_vector(dataset, column, vector)	Attach a 1-D numpy array as a column on a GMT dataset.
Session.read_data(infile, kind[, family, ...])	Read a data file into a GMT data container.
Session.read_virtualfile(vfname[, kind])	Read data from a virtual file and optionally cast into a GMT data container.
Session.virtualfile_from_grid(grid)	Store a grid in a virtual file.
Session.virtualfile_from_matrix(matrix)	Store a 2-D numpy array as a matrix inside a virtual file.
Session.virtualfile_from_stringio(stringio)	Store a `io.StringIO` object in a virtual file.
Session.virtualfile_from_vectors(vectors)	Store a sequence of 1-D vectors as columns of a dataset inside a virtual file.
Session.virtualfile_in([check_kind, data, ...])	Store any data inside a virtual file.
Session.virtualfile_out([kind, fname])	Create a virtual file or an actual file for storing output data.
Session.virtualfile_to_dataset(vfname[, ...])	Output a tabular dataset stored in a virtual file to a different format.
Session.virtualfile_to_raster(vfname[, ...])	Output raster data stored in a virtual file to an `xarray.DataArray` object.
Session.write_data(family, geometry, mode, ...)	Write a GMT data container to a file.

pygmt.clib.Session.call_module

`Session.`**`call_module`**(*module*, *args*)

Call a GMT module with the given arguments.

Wraps `GMT_Call_Module`.

The `GMT_Call_Module` API function supports passing module arguments in three different ways:

1. Pass a single string that contains whitespace-separated module arguments.
2. Pass a list of strings and each string contains a module argument.
3. Pass a list of `GMT_OPTION` data structure.

Both options 1 and 2 are implemented in this function, but option 2 is preferred because it can correctly handle special characters like whitespaces and quotation marks in module arguments.

Parameters

- **`module`** (`str`) – The GMT module name to be called (`"coast"`, `"basemap"`, etc).
- **`args`** (`str` | `list`[`str`]) – Module arguments that will be passed to the GMT module. It can be either a single string (e.g., `"-R0/5/0/10 -JX10c -BWSen+t'My Title'"`) or a list of strings (e.g., `["-R0/5/0/10", "-JX10c", "-BWSEN+tMy Title"]`).

Raises

- ***GMTInvalidInput*** – If the `args` argument is not a string or a list of strings.
- ***GMTCLibError*** – If the returned status code of the function is non-zero.

Return type

`None`

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.

Return type

`None`

pygmt.clib.Session.create_data

Session.**create_data**(*family*, *geometry*, *mode*, *dim=None*, *ranges=None*, *inc=None*, *registration='GMT_GRID_NODE_REG'*, *pad=None*)

Create an empty GMT data container and allocate space to hold data.

Valid data families and geometries are in `FAMILIES` and `GEOMETRIES`.

There are two ways to define the dimensions needed to actually allocate memory:

1. Via `ranges`, `inc` and `registration`.
2. Via `dim` and `registration`.

   `dim` contains up to 4 values and they have different meanings for different GMT data families:

   For `GMT_DATASET`:

   - 0: number of tables
   - 1: number of segments per table
   - 2: number of rows per segment
   - 3: number of columns per row

   For `GMT_VECTOR`:

   - 0: number of columns
   - 1: number of rows [optional, can be 0 if unknown]
   - 2: data type (e.g., `GMT_DOUBLE`) [Will be overwritten by `put_vector`]

   For `GMT_GRID`/`GMT_IMAGE`/`GMT_CUBE`/`GMT_MATRIX`:

   - 0: number of columns
   - 1: number of rows
   - 2: number of bands or layers [Ignored for `GMT_GRID`]

- 3: data type (e.g., GMT_DOUBLE) [For GMT_MATRIX only, but will be overwritten by put_matrix]

In other words, inc is assumed to be 1.0, and ranges is [0, dim[0], 0, dim[1]] for pixel registration or [0, dim[0]-1.0, 0, dim[1]-1.0] for grid registration.

When creating a grid/image/cube, you can do it in one or two steps:

1. Call this function with mode="GMT_CONTAINER_AND_DATA". This creates a header and allocates a grid or an image
2. Call this function twice:
   1. First with mode="GMT_CONTAINER_ONLY", to create a header only and compute the dimensions based on other parameters
   2. Second with mode="GMT_DATA_ONLY", to allocate the grid/image/cube array based on the dimensions already set. This time, you pass NULL for dim/ranges/inc/registration/pad and let data be the void pointer returned in the first step.

Note: This is not implemented yet, since this function doesn't have the data parameter.

Parameters

- **family** (str) – A valid GMT data family name (e.g., "GMT_IS_DATASET"). See FAMILIES for valid names.
- **geometry** (str) – A valid GMT data geometry name (e.g., "GMT_IS_POINT"). See GEOMETRIES for valid names.
- **mode** (str) – A valid GMT data mode. See MODES for valid names. For GMT_IS_DATASET/GMT_IS_MATRIX/GMT_IS_VECTOR, adding GMT_WITH_STRINGS to the mode will allocate the corresponding arrays of string pointers.
- **dim** (Sequence[int] | None, default: None) – The dimensions of the dataset, as explained above. If None, will pass in the NULL pointer.
- **ranges** (Sequence[float] | None, default: None) – The data extent.
- **inc** (Sequence[float] | None, default: None) – The increments between points of the dataset.
- **registration** (Literal['GMT_GRID_NODE_REG', 'GMT_GRID_PIXEL_REG'], default: 'GMT_GRID_NODE_REG') – The node registration. Can be "GMT_GRID_PIXEL_REG" or "GMT_GRID_NODE_REG".
- **pad** (int | None, default: None) – The padding for GMT_IS_GRID/GMT_IS_IMAGE/GMT_IS_CUBE. If None, defaults to "GMT_PAD_DEFAULT".

  For GMT_IS_MATRIX, it can be:

  – 0: default row/col orientation [Default]

  – 1: row-major format (C)

  – 2: column-major format (FORTRAN)

Return type
c_void_p

Returns
data_ptr – A ctypes pointer (an integer) to the allocated GMT data container.

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.extract_region

Session.**extract_region**()

Extract the region of the currently active figure.

Retrieves the information from the PostScript file, so it works for country codes as well.

Return type
ndarray

Returns
region – A numpy 1-D 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="M12c", frame=True, land="black"
... )
>>> with Session() as lib:
...     region = lib.extract_region()
>>> print(", ".join([f"{x:.2f}" for x in region]))
0.00, 10.00, -20.00, -10.00
```

Using ISO country codes for the regions (for example `"US.HI"` for Hawai'i):

```
>>> fig = pygmt.Figure()
>>> fig.coast(region="US.HI", projection="M12c", frame=True, land="black")
>>> with Session() as lib:
...     region = lib.extract_region()
>>> print(", ".join([f"{x:.2f}" for x in region]))
-164.71, -154.81, 18.91, 23.58
```

The country codes can have an extra argument that rounds the region to multiples 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="M12c", frame=True, land="black")
>>> with Session() as lib:
...     region = lib.extract_region()
>>> print(", ".join([f"{x:.2f}" for x in region]))
-165.00, -150.00, 15.00, 25.00
```

pygmt.clib.Session.get_common

`Session.`**`get_common`**(*option*)

Inquire if a GMT common option has been set and return its current value if possible.

Parameters

option (str) – The GMT common option to check. Valid options are "B", "I", "J", "R", "U", "V", "X", "Y", "a", "b", "f", "g", "h", "i", "n", "o", "p", "r", "s", "t", and ":".

Return type

bool | int | float | ndarray

Returns

value – Whether the option was set or its value. If the option was not set, return False. Otherwise, the return value depends on the choice of the option.

- options "B", "J", "U", "g", "n", "p", and "s": return True if set, else False (bool)
- "I": 2-element array for the increments (float)
- "R": 4-element array for the region (float)
- "V": the verbose level (int)
- "X": the xshift (float)
- "Y": the yshift (float)
- "a": geometry of the dataset (int)
- "b": return 0 if -bi was set and 1 if -bo was set (int)
- "f": return 0 if -fi was set and 1 if -fo was set (int)
- "h": whether to delete existing header records (int)
- "i": number of input columns (int)
- "o": number of output columns (int)
- "r": registration type (int)
- "t": 2-element array for the transparency (float)
- ":": return 0 if -:i was set and 1 if -:o was set (int)

Examples

```
>>> with Session() as lib:
...     lib.call_module(
...         "basemap", ["-R0/10/10/15", "-JX5i/2.5i", "-Baf", "-Ve"]
...     )
...     region = lib.get_common("R")
```

(continues on next page)

(continued from previous page)

```
...     projection = lib.get_common("J")
...     timestamp = lib.get_common("U")
...     verbose = lib.get_common("V")
...     lib.call_module("plot", ["-T", "-Xw+1i", "-Yh-1i"])
...     xshift = lib.get_common("X")  # xshift/yshift are in inches
...     yshift = lib.get_common("Y")
>>> print(region, projection, timestamp, verbose, xshift, yshift)
[ 0. 10. 10. 15.] True False 3 6.0 1.5
>>> with Session() as lib:
...     lib.call_module("basemap", ["-R0/10/10/15", "-JX5i/2.5i", "-Baf"])
...     lib.get_common("A")
Traceback (most recent call last):
...
pygmt.exceptions.GMTInvalidInput: Unknown GMT common option flag 'A'.
```

pygmt.clib.Session.get_default

Session.**get_default**(*name*)

Get the value of a GMT configuration parameter or a GMT API parameter.

In addition to the long list of GMT configuration parameters, the following API parameter names are also supported:

- "API_VERSION": The GMT API 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
- "API_BIN_VERSION": The GMT binary version (with git information)

Parameters

name (str) – The name of the GMT configuration parameter (e.g., "PROJ_LENGTH_UNIT") or a GMT API parameter (e.g., "API_VERSION").

Return type

str

Returns

value – The current value for the parameter.

Raises

GMTCLibError – If the parameter doesn't exist.

pygmt.clib.Session.get_enum

Session.**get_enum**(*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"`).

Return type
int

Returns
value – 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.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 | None, default: None) – 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.

Return type
Callable

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'>
```

pygmt.clib.Session.inquire_virtualfile

Session.**inquire_virtualfile**(*vfname*)

Get the family of a virtual file.

Parameters
vfname (str) – Name of the virtual file to inquire.

Return type
int

Returns
family – The integer value for the family of the virtual file.

Examples

```
>>> from pygmt.clib import Session
>>> with Session() as lib:
...     with lib.virtualfile_out(kind="dataset") as vfile:
...         family = lib.inquire_virtualfile(vfile)
...         assert family == lib["GMT_IS_DATASET"]
```

pygmt.clib.Session.open_virtualfile

Session.**open_virtualfile**(*family*, *geometry*, *direction*, *data*)

Open a GMT virtual file associated with a data object for reading or writing.

GMT uses a virtual file scheme to pass in data or get data from 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 file, or get the output from a module that writes to a 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** (c_void_p | None) – The ctypes void pointer to the GMT data structure. For output (i.e., `direction="GMT_OUT"`), it can be `None` to have GMT automatically allocate the output GMT data structure.

Yields
vfname – The name of the virtual file that you can pass to a GMT module.

Return type
Generator[str, None, None]

Examples

```
>>> from pygmt.helpers import GMTTempFile
>>> 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"
```

(continues on next page)

(continued from previous page)

```
...     geometry = "GMT_IS_POINT"
...     dataset = lib.create_data(
...         family=family,
...         geometry=geometry,
...         mode="GMT_CONTAINER_ONLY",
...         dim=[2, 5, lib["GMT_INT"], 0],  # ncolumns, nrows, dtype, unused
...     )
...     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_virtualfile(*vfargs) as vfile:
...         # Send the output to a temp file so that we can read it
...         with GMTTempFile() as ofile:
...             lib.call_module("info", [vfile, f"->{ofile.name}"])
...             print(ofile.read().strip())
<vector memory>: N = 5 <0/4> <5/9>
```

pygmt.clib.Session.put_matrix

`Session.`**`put_matrix`**(*dataset*, *matrix*, *pad=0*)

Attach a 2-D numpy 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 with `family="GMT_IS_DATASET|GMT_VIA_MATRIX"`.

Not all numpy dtypes are supported, only: int8, int16, int32, int64, longlong, uint8, uint16, uint32, uint64, ulonglong, float32, and float64.

Warning

The numpy array must be C contiguous in memory. Use `numpy.ascontiguousarray` to make sure your matrix is contiguous (it won't copy if it already is).

Parameters

- **`dataset`** (`c_void_p`) – The ctypes void pointer to a `GMT_MATRIX` data container. Create it with *pygmt.clib.Session.create_data*.
- **`matrix`** (`ndarray`) – The array that will be attached to the dataset. Must be a 2-D C contiguous array.
- **`pad`** (`int`, default: `0`) – 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 a non-zero status.

Return type

`None`

pygmt.clib.Session.put_strings

`Session.`**`put_strings`**(*dataset*, *family*, *strings*)

Attach a 1-D numpy 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 2-D 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** (`c_void_p`) – The ctypes void pointer to a `GMT_VECTOR`/`GMT_MATRIX` data container. 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** (`ndarray`) – The array that will be attached to the dataset. Must be a 1-D C contiguous array.

Raises

GMTCLibError – If given invalid input or `GMT_Put_Strings` exits with a non-zero status.

Return type

`None`

pygmt.clib.Session.put_vector

`Session.`**`put_vector`**(*dataset*, *column*, *vector*)

Attach a 1-D numpy 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 with `family="GMT_IS_DATASET|GMT_VIA_VECTOR"`.

Not all numpy dtypes are supported, only: int8, int16, int32, int64, longlong, uint8, uint16, uint32, uint64, ulonglong, float32, float64, str_, datetime64, and timedelta64.

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** (`c_void_p`) – The ctypes void pointer to a `GMT_VECTOR` data container. Create it with *pygmt.clib.Session.create_data*.
- **column** (`int`) – The column number of this vector in the dataset (starting from 0).

- **vector** (ndarray) – The array that will be attached to the dataset. Must be a 1-D C contiguous array.

Raises

GMTCLibError – If given invalid input or GMT_Put_Vector exits with a non-zero status.

Return type

None

pygmt.clib.Session.read_data

Session.**read_data**(*infile*, *kind*, *family=None*, *geometry=None*, *mode='GMT_READ_NORMAL'*, *region=None*, *data=None*)

Read a data file into a GMT data container.

Wraps GMT_Read_Data but only allows reading from a file. The function definition is different from the original C API function.

Parameters

- **infile** (str) – The input file name.
- **kind** (Literal['dataset', 'grid', 'image']) – The data kind of the input file. Valid values are "dataset", "grid" and "image".
- **family** (str | None, default: None) – A valid GMT data family name (e.g., "GMT_IS_DATASET"). See the FAMILIES attribute for valid names. If None, will determine the data family from the kind parameter.
- **geometry** (str | None, default: None) – A valid GMT data geometry name (e.g., "GMT_IS_POINT"). See the GEOMETRIES attribute for valid names. If None, will determine the data geometry from the kind parameter.
- **mode** (str, default: 'GMT_READ_NORMAL') – How the data is to be read from the file. This option varies depending on the given family. See the GMT API documentation for details. Default is GMT_READ_NORMAL which corresponds to the default read mode value of 0 in the GMT_enum_read enum.
- **region** (Sequence[float] | None, default: None) – Subregion of the data, in the form of [xmin, xmax, ymin, ymax, zmin, zmax]. If None, the whole data is read.
- **data** (default: None) – None or the pointer returned by this function after a first call. It's useful when reading grids/images/cubes in two steps (get a grid/image/cube structure with a header, then read the data).

Returns

Pointer to the data container, or None if there were errors.

Raises

GMTCLibError – If the GMT API function fails to read the data.

pygmt.clib.Session.read_virtualfile

Session.**read_virtualfile**(*vfname*, *kind=None*)

Read data from a virtual file and optionally cast into a GMT data container.

Parameters

- **vfname** (str) – Name of the virtual file to read.

- **kind** (Literal['dataset', 'grid', 'image', 'cube', None], default: None) – Cast the data into a GMT data container. Valid values are "dataset", "grid", "image" and None. If None, will return a ctypes void pointer.

Returns

pointer – Pointer to the GMT data container. If kind is None, returns a ctypes void pointer instead.

Examples

```
>>> from pathlib import Path
>>> from pygmt.clib import Session
>>> from pygmt.helpers import GMTTempFile
>>>
>>> # Read dataset from a virtual file
>>> with Session() as lib:
...     with GMTTempFile(suffix=".txt") as tmpfile:
...         with Path(tmpfile.name).open(mode="w") as fp:
...             print("1.0 2.0 3.0 TEXT", file=fp)
...         with lib.virtualfile_out(kind="dataset") as vouttbl:
...             lib.call_module("read", [tmpfile.name, vouttbl, "-Td"])
...             # Read the virtual file as a void pointer
...             void_pointer = lib.read_virtualfile(vouttbl)
...             assert isinstance(void_pointer, int)  # void pointer is an int
...             # Read the virtual file as a dataset
...             data_pointer = lib.read_virtualfile(vouttbl, kind="dataset")
...             assert isinstance(data_pointer, ctp.POINTER(_GMT_DATASET))
>>>
>>> # Read grid from a virtual file
>>> with Session() as lib:
...     with lib.virtualfile_out(kind="grid") as voutgrd:
...         lib.call_module("read", ["@earth_relief_01d_g", voutgrd, "-Tg"])
...         # Read the virtual file as a void pointer
...         void_pointer = lib.read_virtualfile(voutgrd)
...         assert isinstance(void_pointer, int)  # void pointer is an int
...         data_pointer = lib.read_virtualfile(voutgrd, kind="grid")
...         assert isinstance(data_pointer, ctp.POINTER(_GMT_GRID))
```

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 data container 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_virtualfile*.

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 (DataArray) – The grid that will be included in the virtual file.

Yields
fname – The name of virtual file. Pass this as a file name argument to a GMT module.

Return type
Generator[str, None, None]

Examples

```
>>> from pygmt.helpers.testing import load_static_earth_relief
>>> from pygmt.helpers import GMTTempFile
>>> data = load_static_earth_relief()
>>> print(data.shape)
(14, 8)
>>> print(data.lon.values.min(), data.lon.values.max())
-54.5 -47.5
>>> print(data.lat.values.min(), data.lat.values.max())
-23.5 -10.5
>>> print(data.values.min(), data.values.max())
190.0 981.0
>>> with Session() as ses:
...     with ses.virtualfile_from_grid(data) as fin:
...         # Send the output to a file so that we can read it
...         with GMTTempFile() as fout:
...             ses.call_module(
...                 "grdinfo", [fin, "-L0", "-Cn", f"->{fout.name}"]
...             )
...             print(fout.read().strip())
-55 -47 -24 -10 190 981 1 1 8 14 1 1
>>> # The output is: w e s n z0 z1 dx dy n_columns n_rows reg gtype
```

pygmt.clib.Session.virtualfile_from_matrix

Session.**virtualfile_from_matrix**(*matrix*)

Store a 2-D numpy array as a matrix 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` data container pretending to be a `GMT_DATASET` data container.

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_virtualfile*.

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 (ndarray) – The matrix that will be included in the GMT data container.

Yields

fname – The name of virtual file. Pass this as a file name argument to a GMT module.

Return type

Generator[str, None, None]

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", [fin, f"->{fout.name}"])
...             print(fout.read().strip())
<matrix memory>: N = 4 <0/9> <1/10> <2/11>
```

pygmt.clib.Session.virtualfile_from_stringio

Session.**virtualfile_from_stringio**(*stringio*)

Store a io.StringIO object in a virtual file.

Store the contents of a io.StringIO object in a GMT_DATASET container and create a virtual file to pass to a GMT module.

For simplicity, currently we make following assumptions in the StringIO object

- "#" indicates a comment line.
- ">" indicates a segment header.

Parameters

stringio (StringIO) – The io.StringIO object containing the data to be stored in the virtual file.

Yields

fname – The name of the virtual file.

Return type

Generator[str, None, None]

Examples

```
>>> import io
>>> from pygmt.clib import Session
>>> # A StringIO object containing legend specifications
>>> stringio = io.StringIO(
...     "# Comment\n"
...     "H 24p Legend\n"
```

(continues on next page)

(continued from previous page)

```
...     "N 2\n"
...     "S 0.1i c 0.15i p300/12 0.25p 0.3i My circle\n"
... )
>>> with Session() as lib:
...     with lib.virtualfile_from_stringio(stringio) as fin:
...         lib.virtualfile_to_dataset(vfname=fin, output_type="pandas")
                                          0
0                                 H 24p Legend
1                                        N 2
2  S 0.1i c 0.15i p300/12 0.25p 0.3i My circle
```

pygmt.clib.Session.virtualfile_from_vectors

Session.**virtualfile_from_vectors**(*vectors*)

Store a sequence of 1-D vectors as columns of a dataset inside a virtual file.

Use the virtual file name to pass the dataset with 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_virtualfile*.

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 2-D array), they will need to be copied to a contiguous block.

Parameters

vectors (Sequence) – A sequence of vectors that will be stored in the dataset. All must be of the same size.

Yields

fname – The name of virtual file. Pass this as a file name argument to a GMT module.

Return type

Generator[str, None, None]

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", [fin, f"->{fout.name}"])
...             print(fout.read().strip())
<vector memory>: N = 3 <1/3> <4/6> <7/9>
```

pygmt.clib.Session.virtualfile_in

Session.**virtualfile_in**(*check_kind=None*, *data=None*, *x=None*, *y=None*, *z=None*, *mincols=2*, *required_data=True*, *required_z=False*, *extra_arrays=None*)

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 or None*) – Used to validate the type of data that can be passed in. Choose from 'raster', 'vector', or None. Default is None (no validation).
- **data** (default: 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** (*1-D arrays or None*) – x, y, and z columns as numpy arrays.
- **mincols** (default: 2) – Number of minimum required columns. Default is 2 (i.e. require x and y columns).
- **required_data** (*bool*) – Set to True when 'data' is required, or False when dealing with optional virtual files. [Default is True].
- **required_z** (*bool*) – State whether the 'z' column is required.

  Deprecated since version v0.16.0: The parameter 'required_z' will be removed in v0.20.0. Use parameter 'mincols' instead. E.g., required_z=True is equivalent to mincols=3.
- **extra_arrays** (*list of 1-D arrays*) – 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.

  Deprecated since version v0.16.0: The parameter 'extra_arrays' will be removed in v0.20.0. Prepare and pass a dictionary of arrays instead to the *data* parameter. E.g., data={"x": x, "y": y, "size": size}.

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_in(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, f"->{fout.name}"])
...             print(fout.read().strip())
<vector memory>: N = 3 <7/9> <4/6> <1/3>
```

pygmt.clib.Session.virtualfile_out

Session.**virtualfile_out**(*kind='dataset'*, *fname=None*)

Create a virtual file or an actual file for storing output data.

If `fname` is not given, a virtual file will be created to store the output data into a GMT data container and the function yields the name of the virtual file. Otherwise, the output data will be written into the specified file and the function simply yields the actual file name.

Parameters

- **kind** (`Literal['dataset', 'grid', 'image']`, default: `'dataset'`) – The data kind of the virtual file to create. Valid values are `"dataset"`, `"grid"`, and `"image"`. Ignored if `fname` is specified.
- **fname** (`str | None`, default: `None`) – The name of the actual file to write the output data. No virtual file will be created.

Yields

vfile – Name of the virtual file or the actual file.

Return type

`Generator[str, None, None]`

Examples

```
>>> from pathlib import Path
>>> from pygmt.clib import Session
>>> from pygmt.datatypes import _GMT_DATASET
>>> from pygmt.helpers import GMTTempFile
>>>
>>> with GMTTempFile(suffix=".txt") as tmpfile:
...     with Path(tmpfile.name).open(mode="w") as fp:
...         print("1.0 2.0 3.0 TEXT", file=fp)
...
...     # Create a virtual file for storing the output table.
...     with Session() as lib:
...         with lib.virtualfile_out(kind="dataset") as vouttbl:
...             lib.call_module("read", [tmpfile.name, vouttbl, "-Td"])
...             ds = lib.read_virtualfile(vouttbl, kind="dataset")
...             assert isinstance(ds.contents, _GMT_DATASET)
...
...     # Write data to an actual file without creating a virtual file.
...     with Session() as lib:
...         with lib.virtualfile_out(fname=tmpfile.name) as vouttbl:
...             assert vouttbl == tmpfile.name
...             lib.call_module("read", [tmpfile.name, vouttbl, "-Td"])
...         line = Path(vouttbl).read_text()
...         assert line == "1\t2\t3\tTEXT\n"
```

pygmt.clib.Session.virtualfile_to_dataset

Session.**virtualfile_to_dataset**(*vfname*, *output_type='pandas'*, *header=None*, *column_names=None*, *dtype=None*, *index_col=None*)

Output a tabular dataset stored in a virtual file to a different format.

The format of the dataset is determined by the `output_type` parameter.

Parameters

- **vfname** (str) – The virtual file name that stores the result data.
- **output_type** (Literal['pandas', 'numpy', 'file', 'strings'], default: 'pandas') – Desired output type of the result data.
  - "pandas" will return a pandas.DataFrame object.
  - "numpy" will return a numpy.ndarray object.
  - "file" means the result was saved to a file and will return None.
  - "strings" will return the trailing text only as an array of strings.
- **header** (int | None, default: None) – Row number containing column names for the pandas.DataFrame output. header=None means not to parse the column names from table header. Ignored if the row number is larger than the number of headers in the table.
- **column_names** (list[str] | None, default: None) – The column names for the pandas.DataFrame output.
- **dtype** (type | dict[str, type] | None, default: None) – Data type for the columns of the pandas.DataFrame output. Can be a single type for all columns or a dictionary mapping column names to types.
- **index_col** (str | int | None, default: None) – Column to set as the index of the pandas.DataFrame output.

Return type

DataFrame | ndarray | None

Returns

result – The result dataset. If output_type="file" returns None.

Examples

```
>>> from pathlib import Path
>>> import numpy as np
>>> import pandas as pd
>>>
>>> from pygmt.helpers import GMTTempFile
>>> from pygmt.clib import Session
>>>
>>> with GMTTempFile(suffix=".txt") as tmpfile:
...     # prepare the sample data file
...     with Path(tmpfile.name).open(mode="w") as fp:
...         print(">", file=fp)
...         print("1.0 2.0 3.0 TEXT1 TEXT23", file=fp)
...         print("4.0 5.0 6.0 TEXT4 TEXT567", file=fp)
...         print(">", file=fp)
...         print("7.0 8.0 9.0 TEXT8 TEXT90", file=fp)
...         print("10.0 11.0 12.0 TEXT123 TEXT456789", file=fp)
...
...     # file output
...     with Session() as lib:
...         with GMTTempFile(suffix=".txt") as outtmp:
...             with lib.virtualfile_out(
...                 kind="dataset", fname=outtmp.name
```

(continues on next page)

(continued from previous page)

```
...             ) as vouttbl:
...                 lib.call_module("read", [tmpfile.name, vouttbl, "-Td"])
...                 result = lib.virtualfile_to_dataset(
...                     vfname=vouttbl, output_type="file"
...                 )
...                 assert result is None
...                 assert Path(outtmp.name).stat().st_size > 0
...
...     # strings, numpy and pandas outputs
...     with Session() as lib:
...         with lib.virtualfile_out(kind="dataset") as vouttbl:
...             lib.call_module("read", [tmpfile.name, vouttbl, "-Td"])
...
...             # strings output
...             outstr = lib.virtualfile_to_dataset(
...                 vfname=vouttbl, output_type="strings"
...             )
...             assert isinstance(outstr, np.ndarray)
...             assert outstr.dtype.kind in ("S", "U")
...
...             # numpy output
...             outnp = lib.virtualfile_to_dataset(
...                 vfname=vouttbl, output_type="numpy"
...             )
...             assert isinstance(outnp, np.ndarray)
...
...             # pandas output
...             outpd = lib.virtualfile_to_dataset(
...                 vfname=vouttbl, output_type="pandas"
...             )
...             assert isinstance(outpd, pd.DataFrame)
...
...             # pandas output with specified column names
...             outpd2 = lib.virtualfile_to_dataset(
...                 vfname=vouttbl,
...                 output_type="pandas",
...                 column_names=["col1", "col2", "col3", "coltext"],
...             )
...             assert isinstance(outpd2, pd.DataFrame)
>>> outstr
array(['TEXT1 TEXT23', 'TEXT4 TEXT567', 'TEXT8 TEXT90',
   'TEXT123 TEXT456789'], dtype='<U18')
>>> outnp
array([[1.0, 2.0, 3.0, 'TEXT1 TEXT23'],
       [4.0, 5.0, 6.0, 'TEXT4 TEXT567'],
       [7.0, 8.0, 9.0, 'TEXT8 TEXT90'],
       [10.0, 11.0, 12.0, 'TEXT123 TEXT456789']], dtype=object)
>>> outpd
     0    1    2                   3
0  1.0  2.0  3.0         TEXT1 TEXT23
1  4.0  5.0  6.0        TEXT4 TEXT567
2  7.0  8.0  9.0         TEXT8 TEXT90
```

(continues on next page)

(continued from previous page)

```
3  10.0  11.0  12.0  TEXT123 TEXT456789
>>> outpd2
   col1  col2  col3             coltext
0   1.0   2.0   3.0        TEXT1 TEXT23
1   4.0   5.0   6.0       TEXT4 TEXT567
2   7.0   8.0   9.0        TEXT8 TEXT90
3  10.0  11.0  12.0  TEXT123 TEXT456789
```

pygmt.clib.Session.virtualfile_to_raster

Session.**virtualfile_to_raster**(*vfname*, *kind='grid'*, *outgrid=None*)

Output raster data stored in a virtual file to an `xarray.DataArray` object.

The raster data can be a grid, an image or a cube.

Parameters

- **vfname** (`str`) – The virtual file name that stores the result grid/image/cube.
- **kind** (`Literal['grid', 'image', 'cube', None]`, default: `'grid'`) – Type of the raster data. Valid values are `"grid"`, `"image"`, `"cube"` or `None`. If `None`, will inquire the data type from the virtual file name.
- **outgrid** (`str` | `None`, default: `None`) – Name of the output grid/image/cube. If specified, it means the raster data was already saved into an actual file and will return `None`.

Return type

`DataArray` | `None`

Returns

result – The result grid/image/cube. If `outgrid` is specified, return `None`.

Examples

```
>>> from pathlib import Path
>>> from pygmt.clib import Session
>>> from pygmt.helpers import GMTTempFile
>>> with Session() as lib:
...     # file output
...     with GMTTempFile(suffix=".nc") as tmpfile:
...         outgrid = tmpfile.name
...         with lib.virtualfile_out(kind="grid", fname=outgrid) as voutgrd:
...             lib.call_module("read", ["@earth_relief_01d_g", voutgrd, "-Tg"])
...             result = lib.virtualfile_to_raster(
...                 vfname=voutgrd, outgrid=outgrid
...             )
...             assert result == None
...             assert Path(outgrid).stat().st_size > 0
...
...     # xarray.DataArray output
...     outgrid = None
...     with lib.virtualfile_out(kind="grid", fname=outgrid) as voutgrd:
...         lib.call_module("read", ["@earth_relief_01d_g", voutgrd, "-Tg"])
...         result = lib.virtualfile_to_raster(vfname=voutgrd, outgrid=outgrid)
...         assert isinstance(result, xr.DataArray)
```

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.

Return type

None

GMT modules are executed through the *call_module* method:

clib.Session.call_module(module, args)	Call a GMT module with the given arguments.

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 objects to and from GMT virtual files:

clib.Session.virtualfile_in([check_kind, ...])	Store any data inside a virtual file.
clib.Session.virtualfile_out([kind, fname])	Create a virtual file or an actual file for storing output data.
clib.Session.virtualfile_to_dataset(vfname)	Output a tabular dataset stored in a virtual file to a different format.
clib.Session.virtualfile_to_raster(vfname[, ...])	Output raster data stored in a virtual file to an xarray.DataArray object.

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.

continues on next page

Table 30 – continued from previous page

clib.Session.__enter__()	Create a GMT API session.
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 configuration parameter or a GMT API parameter.
clib.Session.get_common(option)	Inquire if a GMT common option has been set and return its current value if possible.
clib.Session.create_data(family, geometry, mode)	Create an empty GMT data container and allocate space to hold data.
clib.Session.put_matrix(dataset, matrix[, pad])	Attach a 2-D numpy array to a GMT dataset.
clib.Session.put_strings(dataset, family, ...)	Attach a 1-D numpy array of dtype str as a column on a GMT dataset.
clib.Session.put_vector(dataset, column, vector)	Attach a 1-D numpy array as a column on a GMT dataset.
clib.Session.read_data(infile, kind[, ...])	Read a data file into a GMT data container.
clib.Session.write_data(family, geometry, ...)	Write a GMT data container to a file.
clib.Session.open_virtualfile(family, ...)	Open a GMT virtual file associated with a data object for reading or writing.
clib.Session.read_virtualfile(vfname[, kind])	Read data from a virtual file and optionally cast into a GMT data container.
clib.Session.extract_region()	Extract the region of the currently active figure.
clib.Session.get_libgmt_func(name[, ...])	Get a ctypes function from the libgmt shared library.
clib.Session.virtualfile_from_grid(grid)	Store a grid in a virtual file.
clib.Session.virtualfile_from_stringio(string	Store a `io.StringIO` object in a virtual file.
clib.Session.virtualfile_from_matrix(matrix)	Store a 2-D numpy array as a matrix inside a virtual file.
clib.Session.virtualfile_from_vectors(vectors	Store a sequence of 1-D vectors as columns of a dataset inside a virtual file.

8.11.2 pygmt.clib.Session.__getitem__

Session.**__getitem__**(*name*)

Get the value of a GMT constant.

Parameters
name (`str`) – The name of the constant (e.g., `"GMT_SESSION_EXTERNAL"`).

Return type
`int`

Returns
value – Integer value of the constant. Do not rely on this value because it might change.

8.11.3 pygmt.clib.Session.__enter__

Session.**__enter__**()

Create a GMT API session.

Calls *pygmt.clib.Session.create*.

8.11.4 pygmt.clib.Session.__exit__

Session.**__exit__**(*exc_type*, *exc_value*, *traceback*)

Destroy the currently open GMT API session.

Calls *pygmt.clib.Session.destroy*.

CHAPTER

NINE

TECHNICAL REFERENCE

The Technical Reference section provides detailed information on the technical aspects of GMT and PyGMT, including supported encodings, fonts, bit and hachure patterns, and other essential components for creating high-quality visualizations. For additional details, visit the GMT Technical Reference.

9.1 Common Parameters

distcalc
Determine how spherical distances are calculated. Valid values are:

- `"g"`: Perform great circle distance calculations, with parameters such as distance increments or radii compared against calculated great circle distances [Default].
- `"e"`: Select ellipsoidal (or geodesic) mode for the highest precision but slowest calculation time.
- `"f"`: Select Flat Earth mode, which gives a more approximate but faster result.

Note: (1) All spherical distance calculations depend on the current ellipsoid (PROJ_ELLIPSOID), the definition of the mean radius (PROJ_MEAN_RADIUS), and the specification of the latitude type (PROJ_AUX_LATITUDE). Geodesic distance calculations are also controlled by the algorithm to use for geodesic calculations (PROJ_GEODESIC). (2) Coordinate transformations that can use ellipsoidal or spherical forms will first consult this parameter if given.

verbose
Select verbosity level, which modulates the messages written to stderr.

Choose among 7 levels of verbosity [Default is `"w"`]:

- `"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

9.2 GMT Map Projections

The table below shows the projection codes for the 31 GMT map projections:

PyGMT Projection Argument	Projection Name
Alon$_0$/lat$_0$[/*horizon*]/*width*	*Lambert azimuthal equal-area projection*
Blon$_0$/lat$_0$/lat$_1$/lat$_2$/*width*	*Albers conic equal-area projection*
Clon$_0$/lat$_0$/*width*	*Cassini cylindrical projection*
Cyl_stere/[lon$_0$/[lat$_0$/]]*width*	*Cylindrical stereographic projection*
Dlon$_0$/lat$_0$/lat$_1$/lat$_2$/*width*	*Equidistant conic projection*
Elon$_0$/lat$_0$[/*horizon*]/*width*	*Azimuthal equidistant projection*
Flon$_0$/lat$_0$[/*horizon*]/*width*	*Gnomonic projection*
Glon$_0$/lat$_0$[/*horizon*]/*width*	*Orthographic projection*
Glon$_0$/lat$_0$/*width*[**+a***azimuth*][**+t***tilt*][**+v***vwidth*/*vheight*][**+w***twist*][**+z***altitude*]	*Perspective projection*
H[lon$_0$/]*width*	*Hammer projection*
I[lon$_0$/]*width*	*Sinusoidal projection*
J[lon$_0$/]*width*	*Miller cylindrical projection*
Kf[lon$_0$/]*width*	*Eckert IV equal-area projection*
Ks[lon$_0$/]*width*	*Eckert VI equal-area projection*
Llon$_0$/lat$_0$/lat$_1$/lat$_2$/*width*	*Lambert conic conformal projection*
M[lon$_0$/[lat$_0$/]]*width*	*Mercator projection*
N[lon$_0$/]*width*	*Robinson projection*
Oalon$_0$/lat$_0$/*azimuth*/*width*[**+v**]	Oblique Mercator projection: *1. origin and azimuth*
Oblon$_0$/lat$_0$/lon$_1$/lat$_1$/*width*[**+v**]	Oblique Mercator projection: *2. two points*
Oclon$_0$/lat$_0$/lon$_p$/lat$_p$/*width*[**+v**]	Oblique Mercator projection: *3. origin and projection pol*
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 projection*
Q[lon$_0$/[lat$_0$/]]*width*	*Cylindrical equidistant projection*
R[lon$_0$/]*width*	*Winkel Tripel projection*
Slon$_0$/lat$_0$[/*horizon*]/*width*	*General stereographic projection*
Tlon$_0$[/lat$_0$]/*width*	*Transverse Mercator projection*
U*zone*/*width*	*Universal Transverse Mercator projection*
V[lon$_0$/]*width*	*Van der Grinten projection*
W[lon$_0$/]*width*	*Mollweide projection*
X*width*[**l**|**p***exp*|**T**|**t**][/*height*[**l**|**p***exp*|**T**|**t**]][**d**]	Cartesian *linear*, *logarithmic*, *power*, and time
Ylon$_0$/lat$_0$/*width*	*Cylindrical equal-area projection*

9.3 Supported Fonts

PyGMT supports the 35 standard PostScript fonts. The table below lists them with their font numbers and font names. When specifying fonts in PyGMT, you can either give the font name or just the font number. For example, to use the font "Helvetica", you can use either `"Helvetica"` or `"0"`. For the special fonts "Symbol" (**12**) and "ZapfDingbats" (**34**), see the *Supported Encodings and Non-ASCII Characters* for the character set. The image below the table shows a visual sample for each font.

Font No.	Font Name	Font No.	Font Name
0	Helvetica	17	Bookman-Demi
1	Helvetica-Bold	18	Bookman-DemiItalic
2	Helvetica-Oblique	19	Bookman-Light
3	Helvetica-BoldOblique	20	Bookman-LightItalic
4	Times-Roman	21	Helvetica-Narrow
5	Times-Bold	22	Helvetica-Narrow-Bold
6	Times-Italic	23	Helvetica-Narrow-Oblique
7	Times-BoldItalic	24	Helvetica-Narrow-BoldOblique
8	Courier	25	NewCenturySchlbk-Roman
9	Courier-Bold	26	NewCenturySchlbk-Italic
10	Courier-Oblique	27	NewCenturySchlbk-Bold
11	Courier-BoldOblique	28	NewCenturySchlbk-BoldItalic
12	Symbol	29	Palatino-Roman
13	AvantGarde-Book	30	Palatino-Italic
14	AvantGarde-BookOblique	31	Palatino-Bold
15	AvantGarde-Demi	32	Palatino-BoldItalic
16	AvantGarde-DemiOblique	33	ZapfChancery-MediumItalic
		34	ZapfDingbats

#	Font Name	#	Font Name
0	Helvetica	**17**	**Bookman-Demi**
1	**Helvetica-Bold**	***18***	***Bookman-DemiItalic***
2	*Helvetica-Oblique*	19	Bookman-Light
3	***Helvetica-BoldOblique***	*20*	*Bookman-LightItalic*
4	Times-Roman	21	Helvetica-Narrow
5	**Times-Bold**	**22**	**Helvetica-Narrow-Bold**
6	*Times-Italic*	*23*	*Helvetica-Narrow-Oblique*
7	***Times-BoldItalic***	***24***	***Helvetica-Narrow-BoldOblique***
8	Courier	25	NewCenturySchlbk-Roman
9	**Courier-Bold**	*26*	*NewCenturySchlbk-Italic*
10	*Courier-Oblique*	**27**	**NewCenturySchlbk-Bold**
11	***Courier-BoldOblique***	***28***	***NewCenturySchlbk-BoldItalic***
12	Σψμβολ (Symbol)	29	Palatino-Roman
13	AvantGarde-Book	*30*	*Palatino-Italic*
14	*AvantGarde-BookOblique*	**31**	**Palatino-Bold**
15	**AvantGarde-Demi**	***32***	***Palatino-BoldItalic***
16	***AvantGarde-DemiOblique***	*33*	*ZapfChancery-MediumItalic*
		34	✺❁❒❄✤❇■✳✪❁▼▲ (ZapfDingbats)

9.4 Bit and Hachure Patterns

PyGMT supports a variety of bit and hachure patterns that can be used to fill polygons.

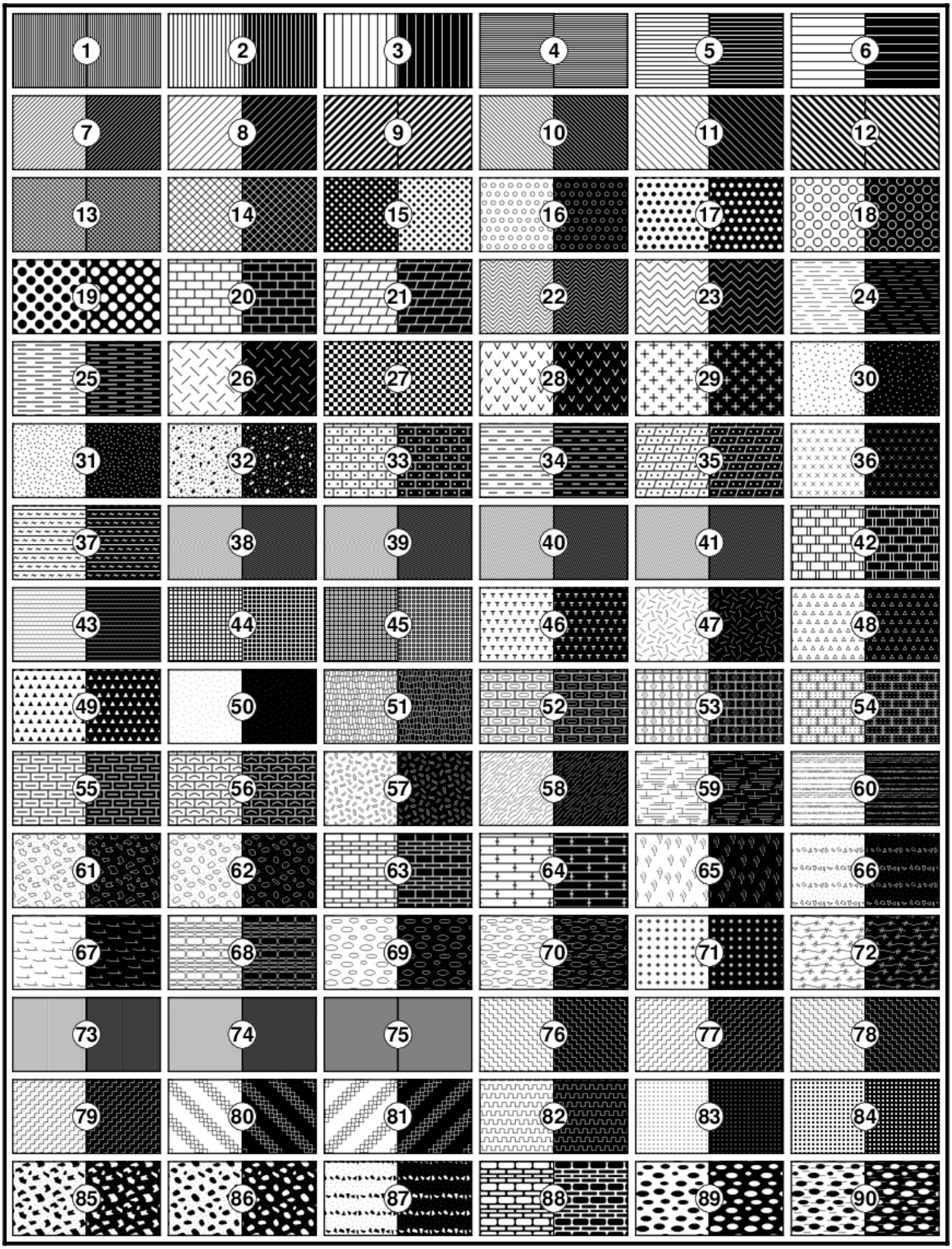
These patterns can be defined using the following syntax:

P|**p***pattern*[**+b***color*][**+f***color*][**+r***dpi*]

pattern can either be a number in the range 1-90 or the name of a 1-, 8-, or 24-bit image raster file. The former will result in one of the 90 predefined 64x64 bit-patterns provided by GMT (see the figure below). The latter allows the user to create customized, repeating images using image raster files.

By specifying uppercase **P** instead of **p** the image will be bit-reversed, i.e., white and black areas will be interchanged (only applies to 1-bit images or predefined bit-image patterns). For these patterns and other 1-bit images one may specify alternative **b**ackground and **f**oreground colors (by appending **+b***color* and/or **+f***color*) that will replace the default white and black pixels, respectively. Excluding *color* from a fore- or background specification yields a transparent image where only the back- or foreground pixels will be painted. The **+r***dpi* modifier sets the resolution in dpi.

The image below shows the 90 predefined bit patterns that can be used in PyGMT.


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90

9.5 Supported Encodings and Non-ASCII Characters

PyGMT supports a number of encodings and each encoding contains a set of ASCII and non-ASCII characters. In PyGMT, you can use any of these ASCII and non-ASCII characters in arguments and text strings. When using non-ASCII characters in PyGMT, the easiest way is to copy and paste the character from the encoding tables below.

Note: The special character  (REPLACEMENT CHARACTER) is used to indicate that the character is undefined in the encoding.

9.5.1 Adobe ISOLatin1+ Encoding

Octal	0	1	2	3	4	5	6	7
\03x		•	…	™	—	–	fi	ž
\04x		!	“	#	$	%	&	’
\05x	(	)	*	+	,	−	.	/
\06x	0	1	2	3	4	5	6	7
\07x	8	9	:	;	<	=	>	?
\10x	@	A	B	C	D	E	F	G
\11x	H	I	J	K	L	M	N	O
\12x	P	Q	R	S	T	U	V	W
\13x	X	Y	Z	[	\	]	^	_
\14x	‘	a	b	c	d	e	f	g
\15x	h	i	j	k	l	m	n	o
\16x	p	q	r	s	t	u	v	w
\17x	x	y	z	{	|	}	~	š
\20x	Œ	†	‡	Ł	⁄	‹	Š	›
\21x	œ	Ÿ	Ž	ł	‰	„	“	”
\22x	ı	`	´	ˆ	˜	¯	˘	˙
\23x	¨	¸	°	˛	‚	″	˝	ˇ
\24x		¡	¢	£	¤	¥	¦	§
\25x	¨	©	ª	«	¬	-	®	¯
\26x	°	±	2	3	´	µ	¶	·
\27x	¸	1	º	»	¼	½	¾	¿
\30x	À	Á	Â	Ã	Ä	Å	Æ	Ç
\31x	È	É	Ê	Ë	Ì	Í	Î	Ï
\32x	Đ	Ñ	Ò	Ó	Ô	Õ	Ö	×
\33x	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
\34x	à	á	â	ã	ä	å	æ	ç
\35x	è	é	ê	ë	ì	í	î	ï
\36x	ð	ñ	ò	ó	ô	õ	ö	÷
\37x	ø	ù	ú	û	ü	ý	þ	ÿ

9.5.2 Adobe Symbol Encoding

Octal	0	1	2	3	4	5	6	7
\04x		!	∀	#	∃	%	&	∋
\05x	(	)	*	+	,	−	.	/
\06x	0	1	2	3	4	5	6	7
\07x	8	9	:	;	<	=	>	?
\10x	≅	Α	Β	Χ	Δ	Ε	Φ	Γ
\11x	Η	Ι	ϑ	Κ	Λ	Μ	Ν	Ο
\12x	Π	Θ	Ρ	Σ	Τ	ϒ	ς	Ω
\13x	Ξ	Ψ	Ζ	[	∴	]	⊥	_
\14x	⍰	α	β	χ	δ	ε	φ	γ
\15x	η	ι	ϕ	κ	λ	μ	ν	ο
\16x	π	θ	ρ	σ	τ	υ	ϖ	ω
\17x	ξ	ψ	ζ	{	|	}	~	□
\24x	€	ϒ	′	≤	⁄	∞	ƒ	♣
\25x	♦	♥	♠	↔	←	↑	→	↓
\26x	°	±	″	≥	×	∝	∂	•
\27x	÷	≠	≡	≈	…	⍰	⍰	↵
\30x	ℵ	ℑ	ℜ	℘	⊗	⊕	∅	∩
\31x	∪	⊃	⊇	⊄	⊂	⊆	∈	∉
\32x	∠	∇	®	©	™	∏	√	·
\33x	¬	∧	∨	⇔	⇐	⇑	⇒	⇓
\34x	◊	〈	®	©	™	∑	⎛	⎜
\35x	⎝	⎡	⎢	⎣	⎧	⎨	⎩	⎪
\36x	□	〉	∫	⌠	⎮	⌡	⎞	⎟
\37x	⎠	⎤	⎥	⌡	⎫	⎬	⎭	□

Note: The octal code `\140` represents the RADICAL EXTENDER character, which is not available in the Unicode character set.

9.5.3 Adobe ZapfDingbats Encoding

Octal	0	1	2	3	4	5	6	7
\04x		✁	✂	✃	✄	☎	✆	✇
\05x	✈	✉	☛	☞	✌	✍	✎	✏
\06x	✐	✑	✒	✓	✔	✕	✖	✗
\07x	✘	✙	✚	✛	✜	✝	✞	✟
\10x	✠	✡	✢	✣	✤	✥	✦	✧
\11x	★	✩	✪	✫	✬	✭	✮	✯
\12x	✰	✱	✲	✳	✴	✵	✶	✷
\13x	✸	✹	✺	✻	✼	✽	✾	✿
\14x	❀	❁	❂	❃	❄	❅	❆	❇
\15x	❈	❉	❊	❋	●	❍	■	❑
\16x	❒	❑	❒	▲	▼	◆	❖	◗
\17x	❘	❙	❚	❛	❜	❝	❞	❟
\20x	❨	❩	❪	❫	❬	❭	❮	❯
\21x	❰	❱	❲	❳	❴	❵	❟	❟
\24x	❟	❡	❢	❣	❤	❥	❦	❧
\25x	♣	♦	♥	♠	①	②	③	④
\26x	⑤	⑥	⑦	⑧	⑨	⑩	❶	❷
\27x	❸	❹	❺	❻	❼	❽	❾	❿
\30x	①	②	③	④	⑤	⑥	⑦	⑧
\31x	⑨	⑩	❶	❷	❸	❹	❺	❻
\32x	❼	❽	❾	❿	➔	→	↔	↕
\33x	➘	➙	➚	➛	➜	→	→	➟
\34x	➠	➡	➢	➢	➤	➥	➦	➧
\35x	➨	➩	➪	➫	➬	➭	➮	➯
\36x	❟	➱	➲	➳	➴	➵	➶	➷
\37x	➸	➹	➺	➻	➼	➽	⇛	❟

9.5.4 ISO/IEC 8859

PyGMT also supports the ISO/IEC 8859 standard for 8-bit character encodings. Refer to ISO/IEC 8859 for descriptions of the different parts of the standard.

For a list of the characters in each part of the standard, refer to the following links:

- ISO/IEC 8859-1
- ISO/IEC 8859-2
- ISO/IEC 8859-3
- ISO/IEC 8859-4
- ISO/IEC 8859-5
- ISO/IEC 8859-6
- ISO/IEC 8859-7
- ISO/IEC 8859-8
- ISO/IEC 8859-9
- ISO/IEC 8859-10

- ISO/IEC 8859-11
- ISO/IEC 8859-13
- ISO/IEC 8859-14
- ISO/IEC 8859-15
- ISO/IEC 8859-16

9.6 Environment Variables

PyGMT's behavior can be controlled through various environment variables. These variables can be set either in your shell environment or within your Python script using the `os.environ` dictionary.

Here we list the environment variables used by PyGMT which are categorized into three groups:

1. System environment variables
2. GMT/PyGMT environment variables
3. Module-specific environment variables

9.6.1 System Environment Variables

TZ
: Specify the time zone for the current calendar time. It can be set to a string that defines the timezone, such as `"UTC"`, `"America/New_York"`, or `"Europe/London"`. Refer to Specifying the Time Zone with TZ for the valid format. If not set, the system's default timezone is used.

9.6.2 GMT/PyGMT Environment Variables

GMT_LIBRARY_PATH
: Specify the directory where the GMT shared library is located. This is useful when GMT is installed in a non-standard location or when you want to use a specific version of GMT. If not set, PyGMT will attempt to find the GMT library in standard system locations.

PYGMT_USE_EXTERNAL_DISPLAY
: Whether to use external viewers for displaying images. If set to `"false"`, PyGMT will not attempt to open images in external viewers. This can be useful when running tests or building the documentation to avoid popping up windows.

9.6.3 Module-Specific Environment Variables

X2SYS_HOME
: Specify the directory where x2sys databases and related settings will be stored. This environment variable is used by x2sys-related functions (e.g., *`pygmt.x2sys_init`*) to manage and access x2sys data. If not set, these functions will use a default directory or prompt for a location.

CHAPTER

TEN

CHANGELOG

10.1 Release v0.15.0 (2025/03/31)

DOI 10.5281/zenodo.15071586

10.1.1 Highlights

- **Fifteenth minor release of PyGMT**
- One new gallery example and two new tutorials
- Figure.shift_origin: Support shifting origins temporarily when used as a context manager (#2509)
- Documentation as HTML ZIP archive and in PDF format for offline reference

10.1.2 Enhancements

- **BREAKING** Support typesetting apostrophe (') and backtick (`) (#3105)
- **BREAKING** pygmt.grdcut: Refactor to store output in virtualfiles for grids (#3115)
- GMTDataArrayAccessor: Support passing values using enums GridRegistration and GridType for grid registration and type (#3696)
- pygmt.grdfill: Add new parameters 'constantfill'/'gridfill'/'neighborfill'/'splinefill' for filling holes (#3855)
- pygmt.grdfill: Add new parameter 'inquire' to inquire the bounds of holes (#3880)
- pygmt.grdfill: Add alias 'coltypes' (-f) (#3869)

10.1.3 Deprecations

- pygmt.grdfill: Deprecate parameter 'no_data' to 'hole' (remove in v0.19.0) (#3852)
- pygmt.grdfill: Deprecate parameter 'mode', use parameters 'constantfill'/'gridfill'/'neighborfill'/'splinefill' instead (remove in v0.19.0) (#3855)
- pygmt.grdclip: Deprecate parameter 'new' to 'replace' (remove in v0.19.0) (#3884)
- clib.Session: Remove deprecated open_virtual_file method, use open_virtualfile instead (Deprecated since v0.11.0) (#3738)
- clib.Session: Remove deprecated virtualfile_from_data method, use virtualfile_in instead (Deprecated since v0.13.0) (#3739)

10.1.4 Documentation

- Add an advanced tutorial for plotting focal mechanisms (beachballs) (#2550)
- Add an advanced tutorial for creating legends (#3594)
- Add a gallery example for Figure.hlines and Figure.vlines (#3755)

10.1.5 Maintenance

- Use the 'release-branch-semver' version scheme for setuptools_scm (#3828)
- Rename _GMT_DATASET.to_dataframe to .to_pandas and _GMT_GRID.to_dataarray/_GMT_IMAGE.to_dataarray to .to_xarray (#3798)
- Bump to ruff 0.9.0, apply ruff 2025 style, and ignore A005 (stdlib-module-shadowing) violations (#3763)
- Use well-known labels in project URLs following PEP753 (#3743)
- clib.conversion: Remove the unused array_to_datetime function (#3507)
- CI: Test on Linux arm64 runners (#3778)
- CI: Build PDF documentation using tectonic (#3765)

Full Changelog: https://github.com/GenericMappingTools/pygmt/compare/v0.14.0…v0.15.0

10.1.6 Contributors

- Dongdong Tian
- Yvonne Fröhlich
- Wei Ji Leong
- Michael Grund

10.2 Release v0.14.2 (2025/02/15)



10.2.1 Bug Fixes

- **Patch release fixing a critical bug introduced in PyGMT v0.14.1**
- Fix the bug for passing text strings with numeric values (#3804)

Full Changelog: https://github.com/GenericMappingTools/pygmt/compare/v0.14.1…v0.14.2

10.2.2 Contributors

- Dongdong Tian

10.3 Release v0.14.1 (2025/02/01)

DOI 10.5281/zenodo.14742338

10.3.1 Highlights

- **Patch release fixing critical bugs in PyGMT v0.14.0**
- Fix the bug of converting Python sequence of datetime-like objects (#3760)

10.3.2 Maintenance

- CI: Separate jobs for publishing to TestPyPI and PyPI (#3742)
- clib.conversion._to_numpy: Add tests for Python sequence of datetime-like objects (#3758)
- Fix an image in README.md (broken on PyPI) and rewrap to 88 characters (#3740)
- Fix the dataset link in the RGB image gallery example (#3781)
- Update License year to 2025 (#3737)

Full Changelog: https://github.com/GenericMappingTools/pygmt/compare/v0.14.0…v0.14.1

10.3.3 Contributors

- Dongdong Tian
- Wei Ji Leong

10.4 Release v0.14.0 (2024/12/31)

DOI 10.5281/zenodo.14535921

10.4.1 Highlights

- **Fourteenth minor release of PyGMT**
- Bump minimum supported version to GMT>=6.4.0 (#3450)
- Two new plotting methods and six new functions to access more GMT remote datasets
- PyArrow as an optional dependency and improved support of PyArrow data types (#3592)

10.4.2 New Features

- Add Figure.hlines for plotting horizontal lines (#923)
- Add Figure.vlines for plotting vertical lines (#3726)
- Add load_black_marble to load “Black Marble” dataset (#3469)
- Add load_blue_marble to load “Blue Marble” dataset (#2235)
- Add load_earth_deflection to load “IGPP Earth east-west and north-south deflection” datasets (#3728)
- Add load_earth_dist to load “GSHHG Earth distance to shoreline” dataset (#3706)

- Add load_earth_mean_dynamic_topography to load "CNES Earth Mean Dynamic Topography" dataset (#3718)
- Add load_earth_mean_sea_surface to load "CNES Earth Mean Sea Surface" dataset (#3717)
- load_earth_free_air_anomaly: Add "uncertainty" parameter to load the "IGPP Earth free-air anomaly uncertainty" dataset (#3727)

10.4.3 Enhancements

- Figure.plot: Add the "symbol" parameter to support plotting data points with varying symbols (#1117)
- Figure.plot3d: Add the "symbol" parameter to support plotting data points with varying symbols (#3559)
- Figure.legend: Support passing a StringIO object as the legend specification (#3438)
- load_tile_map: Add parameter "crs" to set the CRS of the returned dataarray (#3554)
- PyArrow: Support pyarrow arrays with string/large_string/string_view types (#3619)
- Support 1-D/2-D numpy arrays with longlong and ulonglong dtype (#3566)
- GMT_IMAGE: Implement the to_dataarray method for 3-band images (#3128)
- Ensure non-ASCII characters are typeset correctly even if PS_CHAR_ENCODING is not "ISOLatin1+" (#3611)
- Add enums GridRegistration and GridType for grid registration and type (#3693)

10.4.4 Deprecations

- SPEC 0: Bump minimum supported versions to Python 3.11, NumPy 1.25, pandas>=2.0 and xarray>=2023.04 (#3460, #3606, #3697)
- clib.Session.virtualfile_from_vectors: Now takes a sequence of vectors as its single argument (Passing multiple arguments will be unsupported in v0.16.0) (#3522)
- Remove the deprecated build_arg_string function (deprecated since v0.12.0) (#3427)
- Figure.grdcontour: Remove the deprecated syntax for the 'annotation' parameter (deprecated since v0.12.0) (#3428)

10.4.5 Bug Fixes

- launch_external_viewer: Use full path when opening the file in a web browser (#3647)
- PyArrow: Map date32[day]/date64[ms] dtypes in pandas objects to np.datetime64 with correct date/time units (#3617)
- clib.session: Add the GMT_SESSION_NOGDALCLOSE flag to keep GDAL open (#3672)
- Set the "Conventions" attribute to "CF-1.7" for netCDF grids only (#3463)
- Fix the conversion error for pandas.Series with missing values in pandas<=2.1 (#3505, #3596)
- GeoPandas: Explicitly convert columns with overflow integers to avoid OverflowError with fiona 1.10 (#3455)
- Figure.plot/Figure.plot3d: Improve the check of the "style" parameter for "v" or "V" (#3603)
- Correctly reserve the grid data dtype by converting ctypes array to numpy array with np.ctypeslib.as_array (#3446)
- **Breaking**: Figure.text: Fix typesetting of integers when mixed with floating-point values (#3493)

10.4.6 Documentation

- Add basic tutorial "Plotting polygons" (#3593)
- Update the gallery example for plotting lines with LineString/MultiLineString geometry (#3711)
- Add the PyGMT ecosystem page (#3475)
- Document the support policy for optional packages (#3616)
- Document the environment variables that can affect the behavior of PyGMT (#3432)
- Document the built-in patterns in the Technical Reference section (#3466)
- Document Continuous Benchmarking in Maintainers Guides (#3631)
- Add instructions for installing optional dependencies (#3506)
- Update "PyData Ecosystem" to "Scientific Python Ecosystem" (#3447)
- Figure.savefig: Clarify that the "transparent" parameter also works for the PNG file associated with the KML format (#3579)
- Add the PyGMT talk at AGU24 to the "Overview" section (#3685)
- Add the GMT/PyGMT pre-conference workshop at AGU24 to the "External resources" section (#3689)
- Add TODO comments in the maintainers guides and update the release checklist (#3724)

10.4.7 Maintenance

- **Breaking**: data_kind: data is None and required now returns the "empty" kind (#3482)
- **Breaking**: data_kind: Now "matrix" represents a 2-D numpy array and unrecognized data types fall back to "vectors" (#3351)
- Add Support for Python 3.13 (#3490)
- Add the Session.virtualfile_from_stringio method to allow StringIO input for certain functions/methods (#3326)
- Add "geodatasets" as a dependency for docs and update the choropleth example (#3719)
- PyArrow: Check compatibility of pyarrow.array with string type (#2933)
- Rename sphinx-gallery's README.txt to GALLERY_HEADER.rst and require Sphinx-Gallery>=0.17.0 (#3348)
- clib.conversion: Remove the as_c_contiguous function and use np.ascontiguousarray instead (#3492)
- Use TODO comments to track deprecations and workarounds (#3722)
- Move Figure.psconvert into a separate file (#3553)
- Improve the data type checking for 2-D arrays passed to the GMT C API (#3563)
- Enable ruff's TD (flake8-todos), COM (flake8-commas), TRY (tryceratops), and EM (flake8-errmsg) rules (#3723, #3531, #3665, #3661)
- CI: Install pyarrow-core instead of pyarrow from conda-forge (#3698)
- CI: Ensure no hyphens in Python file and directory names in the "Style Checks" workflow (#3703)
- Bump to ruff>=0.8.0 and rename rule TCH to TC (#3662)
- Bump to Ghostscript 10.04.0 (#3443)
- Add enums GridFormat for GMT grid format ID (#3449)

Full Changelog: https://github.com/GenericMappingTools/pygmt/compare/v0.13.0…v0.14.0

10.4.8 Contributors

- Dongdong Tian
- Yvonne Fröhlich
- Wei Ji Leong
- Michael Grund
- Will Schlitzer
- Jiayuan Yao

10.5 Release v0.13.0 (2024/09/05)

DOI 10.5281/zenodo.13679420

10.5.1 Highlights

- **Thirteenth minor release of PyGMT**
- Add new documentation section "Technical Reference" and document the supported encodings and fonts
- Tutorial for "Draping a dataset on top of a topographic surface" (#3316)
- Tutorial for "Typesetting non-ASCII characters" (#3389)

10.5.2 New Features

- Wrap the GMT API function GMT_Read_Data to read data into GMT data containers (#3324)
- Wrap GMT's standard data type GMT_IMAGE for images (#3338)

10.5.3 Enhancements

- **Breaking**: pygmt.x2sys_cross: Refactor to use virtualfiles for output tables (#3182)
- pygmt.show_versions: Warn about incompatible Ghostscript versions (#3244)
- pygmt.show_versions: Show GDAL version (#3364, #3376)
- pygmt.datasets.load_*: Add autocompletion support for the 'resolution' parameter (#3260)
- clib.Session: Refactor the `__getitem__` special method to avoid calling API function GMT_Get_Enum repeatedly (#3261)
- clib: Refactor to avoid checking GMT version repeatedly and only check once when loading the GMT library (#3254)
- Support non-ASCII characters in ISO-8859-x charsets (#3310)
- Refactor to improve the user experience with non-ASCII characters (#3206)

10.5.4 Deprecations

- SPEC 0: Bump minimum supported version to xarray 2022.09 (#3372)
- SPEC 0: Bump minimum supported version to NumPy 1.24 (#3286)
- clib: Deprecate API function 'Session.virtualfile_from_data', use 'Session.virtualfile_in' instead (will be removed in v0.15.0) (#3225)
- Remove the unused pygmt.print_clib_info function (#3257)
- Figure.timestamp: Remove deprecated parameter 'justification', use 'justify' instead (deprecated since v0.11.0) (#3222)

10.5.5 Bug Fixes

- pygmt.set_display: Fix the bug that `method=None` doesn't reset to the default display method (#3396)
- load_tile_map: Register the rio accessor by importing rioxarray, so the returned raster has CRS (#3323)
- load_tile_map: Fix the raster band indexing, should start from 1 (#3322)
- load_tile_map: Replace deprecated rio.set_crs with rio.write_crs (#3321)
- PYGMT_USE_EXTERNAL_DISPLAY should NOT disable image display in Jupyter notebook environment (#3418)

10.5.6 Documentation

- External Resources: Add tutorial in Portuguese and using PyGMT in Google Colab (#3360)
- Remove the non-official GMT wrappers from README (#3413)
- Give recommendations about GMT-Ghostscript incompatibility and the testing example (#3249)
- Document the supported 35 standard Postscript fonts in the Technical Reference section (#3378)
- Add an offboarding access checklist for maintainers (#3411)
- Update the onboarding access checklist in Maintainers Guides (#3404)
- Add sphinx extension myst-nb to enable writing executable Markdown notebooks (#3379)

10.5.7 Maintenance

- pygmt.grd2cpt & pygmt.makecpt: Simplify the logic for dealing with CPT output (#3334)
- geopandas: Use io.StringIO to read geojson data and handle compatibility with geopandas v0.x and v1.x (#3247)
- Simplify the "Minimum Supported Versions" page using MyST customized URL schemes (#3383)
- build_arg_list: Raise an exception if an invalid output file name is given (#3336)
- sphinx-gallery: Temporarily pin to < 0.17.0 (#3350)
- Run pytest with `--color=yes` to force GitHub Actions logs to have color (#3330)
- Patch the callback print function to suppress the UnicodeDecodeError (#3367)
- Move Will from Active Maintainers to Distinguished Contributors (#3388)
- Enable ruff's unspecified-encoding (PLW1514) rule and fix violations (#3319)
- Enable ruff's literal-membership (PLR6201) rule and fix violations (#3317)
- Determine the minimum required versions of dependencies from package metadata for docs (#3380)

- CI: Use OIDC token for codecov uploading (#3163)
- CI: Test NumPy 2.1 in the GMT Tests workflow (#3401)
- CI: Set GMT_ENABLE_OPENMP to TRUE to enable OpenMP support on macOS (#3266)
- CI: Fix the name of the 'build' package to 'python-build' on conda-forge (#3408)
- CI: Bump to ubuntu-24.04 and mambaforge-23.11 in ReadTheDocs (#3296)
- CI: Build GMT dev source code with OpenMP enabled on Linux and GThreads enabled on Linux/macOS (#3011)
- CI: Add pytest plugins pytest-xdist and pytest-rerunfailures (#3193, #3267)
- Add pre-commit config with pre-commit-hooks and enable pre-commit.ci to update hooks quarterly (#3283, #3414)
- Add a test to make sure PyGMT works with paths that contain non-ASCII characters (#3280)

Full Changelog: https://github.com/GenericMappingTools/pygmt/compare/v0.12.0…v0.13.0

10.5.8 Contributors

- Dongdong Tian
- Yvonne Fröhlich
- Wei Ji Leong
- Michael Grund
- Andre L. Belem

10.6 Release v0.12.0 (2024/05/01)

DOI 10.5281/zenodo.11062720

10.6.1 Highlights

- **Twelfth minor release of PyGMT**
- Almost all module wrappers (with a few exceptions) now use in-memory GMT *virtual files* instead of intermediate temporary files to improve performance (#2730)
- Almost all module wrappers (with a few exceptions) now have consistent behavior for table-like output (#1318)
- Adopt SPEC 0 policy for minimum supported versions of GMT, Python, and other core dependencies

10.6.2 Enhancements

- **Breaking**: (Unneeded) extra double quotes around text strings (containing whitespaces) are now considered as part of the text string (#3132, #3457)
- pygmt.project: Add 'output_type' parameter for output in pandas/numpy/file formats (#3110)
- pygmt.grdtrack: Add 'output_type' parameter for output in pandas/numpy/file formats (#3106)
- pygmt.blockm*: Add 'output_type' parameter for output in pandas/numpy/file formats (#3103)
- Figure.grdcontour: Adjust processing arguments passed to "annotation" and "interval" parameters (#3116)
- Figure.contour: Adjust processing arguments passed to "annotation" and "levels" parameters (#2706)

- clib: Wrap the GMT API function GMT_Read_VirtualFile (#2993)
- clib: Add virtualfile_to_dataset method for converting virtualfile to a dataset (#3083, #3140, #3157, #3117)
- clib: Add the virtualfile_out method for creating output virtualfile (#3057)
- Wrap GMT_Inquire_VirtualFile to get the family of virtualfiles (#3152)
- Wrap GMT's standard data type GMT_GRID for grids (#2398)
- Wrap GMT's standard data type GMT_DATASET for table inputs (#2729, #3131, #3174)
- Wrap GMT's data structure GMT_GRID_HEADER for grid/image/cube headers (#3127, #3134)
- Session.call_module: Support passing a list of argument strings (#3139)
- Refactor the _load_remote_dataset function to load tiled and non-tiled grids in a consistent way (#3120)
- Refactor all wrappers to pass an argument list to Session.call_module (#3132)
- Add function build_arg_list for building arguments list from keyword dictionaries (#3149)
- Support left/right single quotation marks in text and arguments (#3192)
- non_ascii_to_octal: Return the input string if it only contains printable ASCII characters (#3199)

10.6.3 Deprecations

- SPEC 0: Set minimum supported versions to Python>=3.10, pandas>=1.5 and xarray>=2022.06 (#3043, #3039, #3151)
- Figure.plot/plot3d/rose: Remove deprecated parameter "color", use "fill" instead (deprecated since v0.8.0) (#3032)
- Figure.velo: Remove deprecated parameters "color"/"uncertaintycolor", use "fill"/"uncertaintyfill" instead (deprecated since v0.8.0) (#3034)
- Figure.wiggle: Remove deprecated parameter "color", use "fillpositive"/"fillnegative" instead (deprecated since v0.8.0) (#3035)
- Figure.grdimage: Remove deprecated parameter "bit_color", use "bitcolor" instead (deprecated since v0.8.0) (#3036)
- Figure: Remove deprecated "xshift" ("X") and "yshift" ("Y") parameters, use "Figure.shift_origin" instead (deprecated since v0.8.0) (#3044)
- Figure: Remove deprecated "timestamp" ("U") parameter, use "Figure.timestamp" instead (deprecated since v0.9.0) (#3045)
- clib: Rename the "virtualfile_from_data" method to "virtualfile_in" (#3068)
- Deprecate the "build_arg_string" function, use build_arg_list instead (deprecated since v0.12.0, will be removed in v0.14.0) (#3184)
- Deprecate the "sequence_plus" converter, only used for the "annotation" parameter of Figure.grdcontour (deprecated since v0.12.0, will be removed in v0.14.0) (#3207)
- Figure.grdcontour: Deprecate parameter "interval" to "levels" (FutureWarning since v0.12.0, will be removed in v0.16.0) (#3209)

10.6.4 Documentation

- External Resources: Add repository "gmt-pygmt-plotting" (#3213)
- Gallery example "Custom symbols": Mention own custom symbols (#3186)

- Intro "04 Table inputs": Document that a list of file names, pathlib.Path objects, URLs, or remote files is supported (3214)
- Tutorial "Plotting text": Rewrite to improve structure, explain more parameters, show list input (#2760)

10.6.5 Maintenance

- pygmt.filter1d: Improve performance by storing output in virtual files (#3085)
- pygmt.grdvolume: Refactor to store output in virtual files instead of temporary files (#3102)
- pygmt.grdhisteq.compute_bins: Refactor to store output in virtual files instead of temporary files (#3109)
- pygmt.grd2xyz: Improve performance by storing output in virtual files (#3097)
- pygmt.select: Improve performance by storing output in virtual files (#3108)
- pygmt.triangulate.delaunay_triples: Improve performance by storing output in virtual files (#3107)
- pygmt.which: Refactor to get rid of temporary files (#3148)
- Use consistent names (vintbl and vingrd) for input virtual files (#3082)
- Add sequence_to_ctypes_array to convert a sequence to a ctypes array (#3136)
- Add strings_to_ctypes_array to convert a sequence of strings into a ctypes array (#3137)
- Figure.psconvert: Ignore the unrecognized "metadata" parameter added by pytest-mpl v0.17.0 (#3054)
- Remote Datasets: Adjust attributes - remove "title", use default of "name" and "long_name", introduce "description" (#3048)
- Adopt SPEC 0 policy and drop NEP 29 policy (#3037)
- Document the support policy for minimum required GMT versions (#3070)
- Bump to ghostscript 10.03.0 (#3112)
- Bump to ruff 0.3.0 (#3081)
- Enable ruff's PTH (flake8-use-pathlib) rules and fix violations (#3129)
- Change the dev dependency "matplotlib" to "matplotlib-base" to reduce environment size (#3158)
- Migrate from os.path to pathlib (#3119)
- CI: Use "gh release" to upload assets to release (#3187)
- CI: Consistently use github.token instead of secrets.GITHUB_TOKEN (#3189)
- CI: Configure workflows to run on "workflow_dispatch" event (#3133)
- Switch to official GitHub action for managing app tokens (#3165)

Full Changelog: https://github.com/GenericMappingTools/pygmt/compare/v0.11.0…v0.12.0

10.6.6 Contributors

- Dongdong Tian
- Yvonne Fröhlich
- Michael Grund
- Wei Ji Leong

10.7 Release v0.11.0 (2024/02/01)

DOI 10.5281/zenodo.10578540

10.7.1 Highlights

- **Eleventh minor release of PyGMT**
- Tutorial for table inputs (#2722) and gallery example for choropleth map (#2796)
- Easy access to planetary relief datasets (Mercury, Venus, Moon, Mars, Pluto) (#3028, #2906, #2674, #2847, #3027)
- Faster PyGMT by ~0.1 seconds for each module call (#2930)

10.7.2 New Features

- Support timedelta64 dtype as input (#2884)
- Figure.text: Support passing in a list of angle/font/justify values (#2720)
- Figure.savefig: Support saving figures in PPM (.ppm) format (#2771)
- Figure.savefig: Support generating GeoTIFF file (with extension '.tiff') (#2698)
- Figure.savefig: Add the 'worldfile' parameter to write a companion world file for raster images (#2766)

10.7.3 Enhancements

- geopandas: Correctly handle columns with integer values bigger than the largest 32-bit integer (#2841)
- pyarrow: Support date32[day] and date64[ms] dtypes in pandas objects (#2845)
- datasets.load_tile_map and Figure.tilemap: Add "zoom_adjust" parameter (#2934)
- grdlandmask: Add common alias "cores" for "x" (#2944)
- Figure.coast: Add alias "box" for "-F" (#2823)
- Improve the error messages for unsupported numpy dtypes (#2856)
- Set GMT_SESSION_NAME to a unique name on Windows for multiprocessing support (#2938)
- Figure.savefig: Support .jpeg as JPEG image extension (#2691)
- Figure.savefig: Support uppercase file extensions (e.g., PNG, PDF) (#2697)

10.7.4 Deprecations

- Rename API function Session.open_virtual_file to Session.open_virtualfile (remove in v0.15.0) (#2996)
- NEP29: Set minimum required version to NumPy 1.23+ (#2991)
- Figure.timestamp: Deprecate parameter 'justification' to 'justify' (remove in v0.13.0) (#3002)
- Figure.grdimage: Remove the unsupported 'img_out'/'A' parameter (#2907)

10.7.5 Bug Fixes

- pygmt.which: Fix the bug when passing multiple files (#2726)
- pygmt.filter1d: Fix the bug that the first line is read as headers (#2780)
- clib: Fix the bug when passing multiple columns of strings with variable lengths to the GMT C API (#2719)
- Let kwargs_to_strings work with default values and positional arguments (#2826)
- Figure.meca: Fix typo pricipal_axis -> principal_axis (#2940)

10.7.6 Documentation

- Add gallery example "Scale bar" (#2822)
- Add gallery example for plotting connection lines ("connection" parameter of Figure.plot) (#2999)
- Add gallery example showing how to adjust line segment ends (caps and joints) (#3015)
- Gallery example "Legend": Update regarding input data and multi-column legends (#2762)
- Add the Japanese "PyGMT-HOWTO" tutorial to "External Resources" (#2743)
- Figure.plot: Update docstring for "connection" parameter to GMT 6.5 (#2994)
- Use consistent description for the "outgrid" parameter (#2874)
- Improve the onboarding access checklist for contributors, maintainers and administrators (#2656)
- Recommend Miniforge instead of Mambaforge (#2833)

10.7.7 Maintenance

- pyarrow: Check compatibility of pyarrow-backed pandas objects with numeric dtypes (#2774)
- Switch away from Stamen basemaps (#2717)
- Add the "validate_output_table_type" function to check the "output_type" parameter (#2772)
- Create "skip_if_no" helper function to skip tests when missing a package (#2883)
- Fix "fixture_xr_image" to open "earth_day_01d" directly with rioxarray (#2963)
- Improve the way to import optional modules (#2809)
- Move variable `__gmt_version__` to pygmt.clib to avoid cyclic-import errors (#2713)
- Refactor the internal "_load_remote_dataset function" to simplify datasets' definitions (#2917)
- Benchmark grdsample, grdfilter and sph2grd with fixed cores (#2945)
- Setup Continuous Benchmarking workflow with pytest-codspeed (#2908)
- Update output shape and mean values from some x2sys_cross tests (#2986)
- Bump the GMT version in CI to 6.5.0 (#2962)
- Bump ghostscript to 10.02.1 (#2694)
- Add support for Python 3.12 (#2711)
- NEP29: Test PyGMT on NumPy 1.26 (#2692)
- CI: Trigger the cache_data workflow in PRs if cache files are added/deleted/updated (#2939)
- CI: Test GMT dev version on Windows by building from source (#2773)
- CI: Set cache-downloads to false to speedup the "Setup Micromamba" step (#2946)

- CI: Run certain GitHub Actions workflows on official repo only (#2951)
- CI: Run benchmarks if PR is labeled with "run/benchmark" (#2958)
- CI: Run "GMT Dev Tests" if PR is labeled with "run/test-gmt-dev" (#2960)
- CI: Require at least one code block separator for example files in the Style Checks workflow (#2810)
- Add Mypy for static type checking (#2808)
- TYP: Improve the doc style for type hints (#2813)
- TYP: Add type hints support for pygmt.datasets.load_sample_data (#2859)
- TYP: Add type hints for the "registration" parameter in pygmt.datasets.load_* functions (#2867)
- TYP: Add type hints for "data_source" in load_earth_relief and load_earth_magnetic_anomaly functions (#2849)
- TYP: Add type hints for parameters of Figure.timestamp(#2890)
- TYP: Add type hints for "terminator" in Figure.solar and simplify codes (#2881)
- TYP: Add type hints for parameters of Figure.shift_origin, improve docstrings, and add inline examples (#2879)
- doc: Convert the installation guides source code from ReST to Markdown (#2992)
- doc: Convert overview source code from ReST to Markdown (#2953)
- doc: Move compatibility table from README to separate file (#2862)
- Enable the PDF format documentation in the ReadTheDocs site (#2876)
- Add the full changelog link to the release drafter template (#2838)
- Release Drafter: Automatically replace GitHub handles with names and links (#2777)
- Exclude CODE_OF_CONDUCT.md, AUTHORSHIP.md and pygmt/tests directory from distributions (#2957)
- Add Zenodo's GMT community to the maintainer's onboarding list (#2761)
- Use ruff to lint and format codes, and remove flakeheaven/isort/black/blackdoc (#2741)
- Use codespell to check common misspellings (#2673)
- Use "# %%" as code block separators in examples (#2662)

Full Changelog: https://github.com/GenericMappingTools/pygmt/compare/v0.10.0…v0.11.0

10.7.8 Contributors

- Dongdong Tian
- Yvonne Fröhlich
- Wei Ji Leong
- Michael Grund
- Max Jones

10.8 Release v0.10.0 (2023/09/02)



10.8.1 Highlights

- **Tenth minor release of PyGMT**
- Support non-ASCII characters in Figure.text (#2638) and other method arguments (#2584)
- Three new tutorials and seven new gallery examples

10.8.2 Enhancements

- Figure.colorbar: Add alias for "Q" (#2608)
- Figure.grdimage: Allow passing RGB xarray.DataArray images (#2590)
- Figure.image: Add alias for "G" (#2615)
- Figure.meca: Add aliases for "L", "T", and "Fr" (#2546)
- clib.Session: Wrap the GMT_Get_Common API function (#2500)
- pygmt.grdfill: Add alias for "N" (#2618)
- pygmt.select: Add aliases for "C", "F", and "L" (#2466)
- pygmt.show_versions: Show versions of IPython and rioxarray (#2492)
- Better handling of optional virtual files (e.g., shading in Figure.grdimage) (#2493)

10.8.3 Deprecations

- Remove the unused pygmt.test() function (#2652)
- Figure.grdimage: Deprecate parameter "bit_color" to "bitcolor" (remove in v0.12.0) (#2635)
- Figure.text: Remove the deprecated "incols" parameter (deprecated since v0.8.0) (#2473)
- NEP29: Set minimum required version to Python 3.9+ (#2487)
- NEP29: Set minimum required version to NumPy 1.22+ (#2586)

10.8.4 Bug Fixes

- load_earth_mask: Keep data's encoding to correctly infer data's registration and gtype information (#2632)
- Geopandas integration: Mapping int/int64 to int32 for OGR_GMT format (#2592)
- Figure.meca: Let the "scale" parameter accept int/float/str values (#2566)
- Figure.meca: Fix beachball offsetting for ndarray input (requires GMT>6.4.0) (#2576)

10.8.5 Documentation

- Document the default CPT for GMT remote datasets (#2573)
- Add tutorial to explain naming of PyGMT figure elements (#2383)
- Add tutorial to show interactive data visualization via `panel` (#2498)
- Add tutorial for cartesian histograms (#2445)
- Add gallery example to show usage of dcw parameter in Figure.coast (#2428)
- Add gallery example to show usage of tile maps (#2585)
- Add gallery example showing how to build an envelope around a curve (#2587)
- Add gallery example for plotting an RGB image from an xarray.DataArray (#2641)

- Add gallery example "Quoted lines" (`style="q"`) (#2563)
- Add gallery example "Decorated lines" (`style="~"`) (#2564)
- Add gallery example "Cross-section along a transect" (#2515)

10.8.6 Maintenance

- Use substitutions to show the minimum required Python and GMT versions dynamically in installation guides (#2488)
- Use np.asarray to convert a 1-D array to datetime type in array_to_datetime (#2481)
- Use consistent docstrings for test files (#2578)
- Use concurrency to cancel previous runs (#2589)
- Set date_format to ISO8601 to silence pandas 2.0 UserWarning on read_csv (#2569)
- Remove dummy_context and use contextlib.nullcontext instead (#2491)
- NEP29: Test PyGMT on NumPy 1.25 (#2581)
- Fix tests for Aug 2023 updated remote datasets (#2636)
- Figure.meca: Refactor the two tests for offsetting beachballs (#2572)
- Figure.meca: Refactor tests for plotting multiple focal mechanisms (#2565)
- Figure.meca: Refactor tests for plotting a single focal mechanism (#2533)
- Figure.meca: Add a test for passing event names via pandas.DataFrame (#2582)
- Exclude bots from contributors in release drafter (#2484)
- Exclude DVC files from source/binary distributions (#2634)
- CI: Use mamba-org/provision-with-micromamba to setup micromamba (#2435)
- CI: Migrate provision-with-micromamba to setup-micromamba (#2536)
- CI: Run dev tests on scientific Python nightly wheels (#2612)
- CI: Remove the deprecated cml-publish command from the dvc-diff workflow (#2559)
- CI: Fix and simplify the dvc-diff workflow (#2549)
- CI: Add the "Doctests" workflow to run doctests weekly (#2456)
- CI: Add detailed descriptions in the workflow files and update maintainer guides (#2496)
- Add private function _validate_data_input to validate input data (#2595)

10.8.7 Contributors

- Dongdong Tian
- Yvonne Fröhlich
- Wei Ji Leong
- Michael Grund
- Jing-Hui Tong
- Max Jones

10.9 Release v0.9.0 (2023/03/31)

DOI 10.5281/zenodo.7772533

10.9.1 Highlights

- **Ninth minor release of PyGMT**
- Add Figure.tilemap to plot XYZ tile maps (#2394)
- Add function to load raster tile maps using contextily (#2125)
- Eleven new/updated gallery and inline examples

10.9.2 New Features

- Add load_earth_mask function for GSHHG Global Earth Mask dataset (#2310)
- Add Figure.timestamp to plot the GMT timestamp logo (#2208, #2425)

10.9.3 Enhancements

- pygmt.surface: Add aliases for "C", "L", "M", and "T" (#2321)
- Figure.meca: Add aliases for "C", "E", "G", and "W" (#2345)
- Figure.colorbar: Add aliases for "L" and "Z" (#2357)

10.9.4 Deprecations

- NEP29: Set minimum required version to NumPy 1.21+ (#2389)
- Recommend Figure.timestamp and remove timestamp (U) alias from all plotting methods (#2135)
- Remove the deprecated load_fractures_compilation function (deprecated since v0.6.0) (#2303)
- Remove the deprecated load_hotspots function (deprecated since v0.6.0) (#2309)
- Remove the deprecated load_japan_quakes function (deprecated since v0.6.0) (#2301)
- Remove the deprecated load_mars_shape function (deprecated since v0.6.0) (#2304)
- Remove the deprecated load_ocean_ridge_points function (deprecated since v0.6.0) (#2308)
- Remove the deprecated load_sample_bathymetry function (deprecated since v0.6.0) (#2305)
- Remove the deprecated load_usgs_quakes function (deprecated since v0.6.0) (#2306)
- pygmt.grdtrack: Remove the warning about the incorrect parameter order of 'points, grid' (warned since v0.7.0) (#2312)

10.9.5 Bug Fixes

- GMTDataArrayAccessor: Fallback to default grid registration and gtype if the grid source file doesn't exist (#2009)
- Figure.subplot: Fix setting "sharex", "sharey", and "frame" in combination with Figure.basemap (#2417)
- Figure.subplot: Fix strange positioning issues after exiting subplot (#2427)
- pygmt.config: Correctly reset to default values that contain whitespaces (#2331)
- pygmt.set_display: Do nothing when the display method is set to 'none' (#2450)

10.9.6 Documentation

- GMTDataArrayAccessor: Add inline examples for setting GMT specific properties (#2370)
- Document limitations of GMT xarray accessors (#2375)
- Revise the notes about registration and gtype of remote datasets (#2384)
- Add project keywords to the pyproject.toml file (#2315)
- Add inline example for colorbar (#2373)
- Add inline example for grdview (#2381)
- Add inline example for load_earth_mask (#2355)
- Add inline example for load_earth_vertical_gravity_gradient (#2356)
- Add inline examples and improve documentation for pygmt.set_display (#2458)
- Add gallery example showing how to use patterns via the "fill" parameter (or similar parameters) (#2329)
- Add gallery example for scatter plot with histograms on sides (#2410)
- Add gallery example showing how to use advanced grdgradient via the "azimuth" & "normalize" parameters (#2354)
- Add gallery example for the Figure.timestamp method (#2391)
- Expand gallery example "Colorbar" for categorical data (#2395)
- Expand gallery example "Focal mechanisms" to use "*fill" and "pen" (#2433)
- Add working example to quickstart section of README (#2369)
- Recommend Mambaforge and mamba in the installation and contributing guides (#2385)

10.9.7 Maintenance

- Add the GMTSampleData class to simplify the load_sample_data and list_sample_data functions (#2342)
- Add a new target 'doctest' to run doctests only and simplify Makefile (#2443)
- Add a package-level variable `__gmt_version__` for development use (#2366)
- Allow printing show_versions() to in-memory buffer to enable testing (#2399)
- Accept a dict containing configurable GMT parameters in build_arg_string (#2324)
- Publish to TestPyPI and PyPI via OpenID Connect token (#2453)
- Remove –sdist –wheel flags from the build command (#2420)
- Replace ModuleNotFoundError with the more general ImportError (#2441)

10.9.8 Contributors

- Dongdong Tian
- Yvonne Fröhlich
- Wei Ji Leong
- Michael Grund
- Will Schlitzer
- Jing-Hui Tong
- Max Jones

10.10 Release v0.8.0 (2022/12/30)

DOI 10.5281/zenodo.7481934

10.10.1 Highlights

- **Eighth minor release of PyGMT**
- Added support for tab auto-completion for all GMT default parameters (#2213)
- Created functions to download GMT remote datasets (#1786)
- Wrapped the ternary module (#1431)
- Added an intro tutorial for creating contour maps (#2126)

10.10.2 New Features

- Add load_earth_free_air_anomaly function for Earth free-air anomaly dataset (#2238)
- Add load_earth_geoid function for Earth Geoid dataset (#2236)
- Add load_earth_magnetic_anomaly function for Earth magnetic anomaly dataset (#2196, #2239, #2241)
- Add load_earth_vertical_gravity_gradient function for Earth vertical gravity gradient dataset (#2240)
- load_earth_relief: Add the support of data sources "gebco" , "gebcosi", and "synbath" (#1818, #2162, #2192, #2281)
- Wrap ternary (#1431)

10.10.3 Enhancements

- Set gridline (if available) as the default grid registration for remote datasets (#2266)
- Add ternary sample dataset (#2211)
- Figure.ternary: Add parameters "alabel", "blabel", and "clabel" (#2139)
- Figure.psconvert: Add a new alias "gs_path" (-G) (#2076)
- Figure.psconvert: Check if the given prefix is valid (#2170)
- Figure.savefig: Raise a FileNotFoundError if the parent directory doesn't exist (#2160)
- Figure.show: Allow keyword arguments passed to Figure.psconvert (#2078)
- pygmt.config: Support tab auto-completion for all GMT defaults (#2213)
- Rewrite the meca function to support offsetting and labeling beachballs (#1784)

10.10.4 Deprecations

- Deprecate xshift (X) and yshift (Y) aliases from all plotting modules (remove in v0.12.0) (#2071)
- Figure.plot: Deprecate parameter "color" to "fill" (remove in v0.12.0) (#2177)
- Figure.plot3d: Deprecate parameter "color" to "fill" (remove in v0.12.0) (#2178)
- Figure.rose: Deprecate parameter color to fill (remove in v0.12.0) (#2181)

- Figure.velo: Deprecate parameters "color" to "fill" and "uncertaintycolor" to "uncertaintyfill" (remove in v0.12.0) (#2206)
- Figure.wiggle: Deprecate parameter "color" (remove in v0.12.0) and add "fillpositive"/"fillnegative" (#2205)
- Figure.psconvert: Remove the deprecated parameter "icc_gray" (deprecated since v0.6.0) (#2267)
- Figure.text: Deprecate parameter "incols" to "use_word" (remove in v0.10.0) (#1964)

10.10.5 Bug Fixes

- Figure.meca: Fix line and circle of offset parameter for dict/pandas input (#2226)
- Figure.meca: Fix beachball offsetting with dict/pandas inputs (#2202)
- Figure.meca: Fix the bug when passing a dict of scalar values to the spec parameter (#2174)
- Figure.ternary: Fix the crash for pd.DataFrame input with GMT 6.3.0-6.4.0 (#2274)

10.10.6 Documentation

- Add intro tutorial section for creating contour map (#2126)
- Add gallery example for Figure.ternary method (#2138)
- Add gallery example showing the usage of vertical and horizontal bars (#1521)
- Add inline example for coast (#2142)
- Add inline example for grdcontour (#2148)
- Add inline example for grdimage (#2146)
- Add inline example for grd2cpt (#2145)
- Add inline example for solar (#2147)
- Add SciPy 2022 talk to presentations (#2053)
- Add instructions to install pygmt kernel for Jupyter users (#2153)
- Improve instructions about setting GMT_LIBRARY_PATH env variable (#2136)
- Add badges for conda package version, license, and twitter (#2081)
- Add PyOpenSci peer reviewed badge to main README (#2112)

10.10.7 Maintenance

- Add an internal function to load GMT remote datasets (#2200)
- Add support for Python 3.11 (#2172)
- NEP29: Test PyGMT on NumPy 1.24 (#2256)
- NEP29: Test PyGMT on NumPy 1.23 and 1.21 (#2057)
- Bump the GMT version in CI to 6.4.0 (#1990)
- Update baseline images for GMT 6.4.0 (#1883)
- Migrate Continuous Documentation from Vercel to Readthedocs (#1859)
- Set nested_sections to False for Sphinx-Gallery 0.11.0 regarding a correct navgation bar (#2046)
- Convert bug report, feature, and module request issue templates into yaml configured forms (#2091, #2214, #2216)
- doc: Set different html_baseurl for stable and dev versions (#2158)

- Update the instructions for checking README syntax (#2265)
- Use longname placeholders in the docstrings for common options (#1932)
- Add optional dependencies to pyproject.toml (#2069)
- Migrate project metadata from setup.py to pyproject.toml following PEP621 (#1848)
- Move blackdoc options to pyproject.toml (#2093)
- Move docformatter options from Makefile to pyproject.toml (#2072)
- Replace flake8 with flakeheaven (#1847)
- Add a workflow and Makefile target to test old GMT versions every Tuesday (#2079)
- Check if a module outputs to a temporary file using "Path().stat().st_size > 0" (#2224)
- pygmt.show_versions: Show GMT binary version and hide the Python interpreter path (#1838)
- Refactor grdview and grdimage to use virtualfile_from_data (#1988)
- Use the org-wide code of conduct (#2020)

10.10.8 Contributors

- Dongdong Tian
- Yvonne Fröhlich
- Will Schlitzer
- Michael Grund
- Wei Ji Leong
- Max Jones

10.11 Release v0.7.0 (2022/07/01)



10.11.1 Highlights

- **Seventh minor release of PyGMT**
- Wrapped 3 GMT modules
- Added two new PyGMT tutorials and EGU 2022 short course to external resources page (#1971 and #1935)

10.11.2 New Features

- Wrap binstats (#1652)
- Wrap filter1d (#1512)
- Wrap dimfilter (#1492)

10.11.3 Enhancements

- Support passing data in NumPy int8, int16, uint8 and uint16 dtypes to GMT (#1963)
- inset: Add region and projection aliases and fix two examples (#1931)
- basemap: Plotting frames if required parameters are not given (#1909)
- basemap: Added box alias for F (#1894)
- Add a sample dataset maunaLoa_co2 (#1961)
- Add a sample dataset notre_dame_topography (#1920)
- Add a sample dataset earth_relief_holes (#1921)

10.11.4 Deprecations

- NEP29: Set minimum required version to NumPy 1.20+ (#1985)
- Figure.wiggle: Remove parameter 'columns', use 'incols' instead. (#1977)
- Figure.histogram and pygmt.info: Remove parameter 'table', use 'data' instead (#1975)
- pygmt.surface: Remove parameter 'outfile', use 'outgrid' instead (#1976)
- blockm/contour/plot/plot3d/rose/surface/wiggle: Change the parameter order of data array and input arrays (#1978)

10.11.5 Bug Fixes

- grdtrack: Fix the bug when profile is given (#1867)
- Fix the grid accessor (grid registration and type) for 3D grids (#1913)

10.11.6 Documentation

- Add instructions to install PyGMT using mamba (#1967)
- Improve two gallery examples regarding categorical colormaps (#1934)
- Add inline example to dimfilter (#1956)
- Add inline example to surface (#1953)
- Add inline example to grdfill (#1954)
- Add inline code examples to contributing guidelines (#1924)
- Add thumbnail images to the external resources page (#1941)
- Redesign the team gallery using sphinx-design's card directive (#1937)

10.11.7 Maintenance

- Fix broken 'Improve this page' links using sphinx variable page_source_suffix (#1969)
- Split up functions for loading datasets (#1955)
- Set setuptools_scm fallback_version to follow PEP440 (#1945)
- Refactor text to use virtualfile_from_data (#1121)
- Run full tests only on Wednesday scheduled jobs (#1833)
- Run GMT Dev Tests on Monday, Wednesday and Friday only (#1922)

- Update GMT Dev Tests workflow to test on macOS-12 and ubuntu-22.04 (#1918)

10.11.8 Contributors

- Dongdong Tian
- Will Schlitzer
- Wei Ji Leong
- Andre L. Belem
- Yvonne Fröhlich
- Max Jones
- Jack Beagley
- Michael Grund

10.12 Release v0.6.1 (2022/04/11)

DOI 10.5281/zenodo.6426493

10.12.1 Highlights

- Patch release which allows passing None explicitly to pygmt functions (#1872, #1862, #1857, #1815)
- A new tutorial for grdhisteq (#1821)

10.12.2 Bug Fixes

- Fix pathlib support for plot and plot3d (#1831)

10.12.3 Documentation

- Add inline example for grdvolume (#1726)
- Format author affiliations in CITATION.cff and AUTHORS.md (#1844)

10.12.4 Maintenance

- NEP29: Run PyGMT tests and docs build on Python 3.10 (#1868)
- Let pygmt.show_versions() report geopandas version (#1846)
- Refactor build_arg_string to also deal with infile and outfile (#1837)
- Migrate build system settings to pyproject.toml following pep517 and pep518 (#1845)
- Use the build package to build sdist and wheel distributions (#1823)
- Let slash command /test-gmt-dev report job URL (#1866)

10.12.5 Contributors

- Dongdong Tian
- Max Jones
- Wei Ji Leong
- Michael Grund
- Will Schlitzer

10.13 Release v0.6.0 (2022/03/14)

DOI 10.5281/zenodo.6349217

10.13.1 Highlights

- **Sixth minor release of PyGMT**
- New inline examples for 14 functions!
- Single `pygmt.datasets.load_sample_data` function for loading any sample dataset (#1685)
- Minimum required GMT version is now 6.3.0 (#1649)

10.13.2 New Features

- Wrap triangulate (#731)
- Wrap grdhisteq (#1433)

10.13.3 Enhancements

- Add alias for blockmean's -S parameter (#1601)
- Allow users to set the waiting time when displaying a preview image using an external viewer (#1618)
- Raise an exception if the given parameter is not recognized and is longer than 2 characters (#1792)

10.13.4 Deprecations

- Figure.plot/plot3d: Remove parameter "sizes", use "size" instead (#1809)
- Figure.contour/plot/plot3d/rose: Remove parameter "columns", use "incols" instead (#1806)
- Figure.psconvert: Add new aliases and deprecate parameter "icc_gray" (remove in v0.8.0) (#1673)
- NEP29: Set minimum required version to Python 3.8+ (#1676)
- NEP29: Set minimum required version to NumPy 1.19+ (#1675)

10.13.5 Bug Fixes

- Allow passing arguments containing spaces into pygmt functions (#1487)
- Fix the spacing parameter processing for many modules (#1805)
- Fix missing gcmt convention keys in pygmt.meca (#1611)

- Fix the spacing parameter and check required parameters in xyz2grd (#1804)
- Fix UnicodeDecodeError with shapefiles for plot and plot3d (#1695)

10.13.6 Documentation

- Add a shorter video introduction to the home page (#1769)
- Add Liam's 2021 ROSES video to learning resources (#1760)
- Add quick conda install instructions in main README (#1717)
- Add instructions for reporting upstream bugs to contributing.md (#1610)
- List key development dependencies to install for new contributors (#1783)
- Update Code of Conduct to v2.1 (#1754)
- Update the contributing guide about pushing changes to dvc and git (#1776)
- Update dataset links to the new remote-datasets site (#1785)
- Add more sections to the API docs (#1643)
- Add an "add a title" to starter tutorial (#1688)
- Reorganize tutorial section in the documentation sidebar (#1603)
- Update the starter tutorial introduction (#1607)
- Add gallery example to showcase blockmean (#1598)
- Add gallery example to showcase project (#1696)
- Update text symbol gallery example (#1648)
- Add inline example for blockmean (#1729)
- Add inline example for blockmedian (#1730)
- Add inline example for blockmode (#1731)
- Add inline example for grd2xyz (#1713)
- Add inline example for grdclip (#1711)
- Add inline example for grdcut (#1689)
- Add inline example for grdgradient (#1720)
- Add inline example for grdlandmask (#1721)
- Add inline example for grdproject (#1722)
- Add inline example for grdsample (#1724)
- Add inline example for grdtrack (#1725)
- Add inline example for select (#1756)
- Add inline example for sph2grd (#1718)
- Add inline example for xyz2grd (#1719)

10.13.7 Maintenance

- Add a test to make sure the incols parameter works for pandas.DataFrame (#1771)
- Add load_static_earth_relief function for internal testing (#1727)
- Migrate pylint settings from .pylintrc to pyproject.toml (#1755)
- NEP29: Test PyGMT on NumPy 1.22 (#1701)
- Replace pkg_resources with importlib.metadata (#1674)
- Update deprecated -g common option syntax (#1670)
- Update deprecated -JG syntax (#1659)
- Use pytest-doctestplus to skip some inline doctests (#1790)
- Use Python 3.10 in Continuous Integration tests (#1577)

10.13.8 Contributors

- Will Schlitzer
- Max Jones
- Dongdong Tian
- Michael Grund
- Wei Ji Leong
- Julius Busecke

10.14 Release v0.5.0 (2021/10/29)

DOI 10.5281/zenodo.5607255

10.14.1 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)

10.14.2 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)

10.14.3 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)

10.14.4 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)

10.14.5 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)

10.14.6 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)

10.14.7 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)

10.14.8 Contributors

- Dongdong Tian
- Michael Grund
- Will Schlitzer
- Wei Ji Leong
- Max Jones
- Yohai Magen
- Amanda Leaman
- @daroari
- @obaney
- @srijac
- Andrés Ignacio Torres
- Becky Salvage
- Claudio Satriano
- Jamie J Quinn
- @carocamargo

10.15 Release v0.4.1 (2021/08/07)

DOI 10.5281/zenodo.5162003

10.15.1 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)

10.15.2 Enhancements

- Add common alias "verbose" (V) to grdlandmask and savefig (#1343)

10.15.3 Bug Fixes

- Change invalid input conditions in grdtrack (#1376)
- Fix bug so that x2sys_cross accepts dataframes with NaN values (#1369)

10.15.4 Documentation

- Combine documentation and compatibility sections in README (#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)

10.15.5 Maintenance

- Add tomli as a dependency in GMT Dev Tests (#1401)
- NEP29: Test PyGMT on NumPy 1.21 (#1355)

10.15.6 Contributors

- Max Jones
- Will Schlitzer
- Michael Grund
- Wei Ji Leong
- Yohai Magen
- Jiayuan Yao
- Dongdong Tian
- Kadatatlu Kishore
- @sean0921
- Soham Banerjee

10.16 Release v0.4.0 (2021/06/20)

DOI 10.5281/zenodo.4978645

10.16.1 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)

10.16.2 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)

10.16.3 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)

10.16.4 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)

10.16.5 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)

10.16.6 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)

10.16.7 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)

10.16.8 Contributors

- Dongdong Tian
- Wei Ji Leong
- Michael Grund
- Max 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

10.17 Release v0.3.1 (2021/03/14)

DOI 10.5281/zenodo.4592991

10.17.1 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

10.17.2 Enhancements

- Support passing a sequence to the spacing parameter of pygmt.info() (#1031)

10.17.3 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)

10.17.4 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)

10.17.5 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)

10.17.6 Contributors

- Dongdong Tian
- Jiayuan Yao
- Wei Ji Leong
- Max Jones
- Michael Grund
- Will Schlitzer
- Liam Toney
- Kathryn Materna
- Alicia Ngoc Diep Ha
- Tawanda Moyo

10.18 Release v0.3.0 (2021/02/15)

DOI 10.5281/zenodo.4522136

10.18.1 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

10.18.2 New Features

- Wrap grd2cpt (#803)
- Let Figure.text support record-by-record transparency (#716)
- Provide basic support for FreeBSD (#700, #878)

10.18.3 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)

10.18.4 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)

10.18.5 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)

10.18.6 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)

10.18.7 Contributors

- Will Schlitzer
- Dongdong Tian
- Wei Ji Leong
- Michael Grund
- Liam Toney
- Max Jones

10.19 Release v0.2.1 (2020/11/14)

DOI 10.5281/zenodo.4253459

10.19.1 Highlights

- **Patch release with more tutorials and gallery examples!**
- Support Python 3.9 (#689)
- Add Liam's ROSES 2020 PyGMT talk (#643)

10.19.2 New Features

- Wrap plot3d (#471)
- Wrap grdfilter (#616)

10.19.3 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)

10.19.4 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)

10.19.5 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 (#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)

10.19.6 Bug Fixes

- Disallow passing arguments like -XNone to GMT (#639)

10.19.7 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)

10.19.8 Contributors

- Dongdong Tian
- Wei Ji Leong
- Conor Bacon
- carocamargo

10.20 Release v0.2.0 (2020/09/12)

DOI 10.5281/zenodo.4025418

10.20.1 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)

10.20.2 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)

10.20.3 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)

10.20.4 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)

10.20.5 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)

10.20.6 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)

10.20.7 Contributors

- Dongdong Tian
- Wei Ji Leong
- Tyler Newton
- Liam Toney

10.21 Release v0.1.2 (2020/07/07)



10.21.1 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

10.21.2 New Features

- Wrap grdcut (#492)
- Add show_versions() function for printing debugging information used in issue reports (#466)

10.21.3 Enhancements

- Change load_earth_relief()'s default resolution to 01d (#488)
- Enhance text with extra functionality and aliases (#481)

10.21.4 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)

10.21.5 Bug Fixes

- Fix a bug when passing data to GMT in Session.open_virtual_file() (#490)

10.21.6 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)

10.21.7 Contributors

- Dongdong Tian
- Wei Ji Leong
- Liam Toney

10.22 Release v0.1.1 (2020/05/22)



10.22.1 Highlights

- Windows users rejoice, this bugfix release is for you!
- Let PyGMT work with the conda GMT package on Windows (#434)

10.22.2 Enhancements

- Handle setting special parameters without default settings for config (#411)

10.22.3 Documentation

- Update install instructions (#430)
- Add PyGMT AGU 2019 poster to website (#425)
- Redirect www.pygmt.org to latest, instead of dev (#423)

10.22.4 Bug Fixes

- Set GMT_COMPATIBILITY to 6 when pygmt session starts (#432)
- Improve how PyGMT finds the GMT library (#440)

10.22.5 Maintenance

- Finalize fixes on Windows test suite for v0.1.1 (#441)
- Cache test data on Azure Pipelines (#438)

10.22.6 Contributors

- Dongdong Tian
- Wei Ji Leong
- Jason K. Moore

10.23 Release v0.1.0 (2020/05/03)

DOI 10.5281/zenodo.3782862

10.23.1 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)

10.23.2 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)

10.23.3 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)

10.23.4 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)

10.23.5 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)

10.23.6 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)

10.23.7 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)

10.23.8 Contributors

- Dongdong Tian
- Wei Ji Leong
- Leonardo Uieda
- Liam Toney
- Brook Tozer
- Claudio Satriano
- Cody Woodson
- Mark Wieczorek
- Philipp Loose
- Kathryn Materna

CHAPTER

ELEVEN

MINIMUM SUPPORTED VERSIONS

PyGMT has adopted SPEC 0 alongside the rest of the scientific Python ecosystem, and will therefore:

- Drop support for Python versions 3 years after their initial release.
- Drop support for core package dependencies (NumPy, pandas, Xarray) 2 years after their initial release.

In addition to the above, the PyGMT team has also decided to:

- Drop support for GMT versions 3 years after their initial release, while ensuring at least two latest minor versions remain supported.
- Maintain support for *optional dependencies* for at least 1 year after their initial release. Users are encouraged to use the most up-to-date optional dependencies where possible.

Note

The SPEC 0 policy is enforced on a best-effort basis, and the PyGMT team may decide to drop support for core (and optional) package dependencies earlier than recommended for compatibility reasons.

PyGMT Version	Documentation	GMT	Python	NumPy	pandas	Xarray
Dev*	Web, HTML+ZIP, PDF	>=6.4.0	>=3.11	>=1.26	>=2.1	>=2023.07
v0.15.0	Web, HTML+ZIP, PDF	>=6.4.0	>=3.11	>=1.25	>=2.0	>=2023.04
v0.14.2	Web, HTML+ZIP	>=6.4.0	>=3.11	>=1.25	>=2.0	>=2023.04
v0.14.1	Web, HTML+ZIP	>=6.4.0	>=3.11	>=1.25	>=2.0	>=2023.04
v0.14.0	Web, HTML+ZIP	>=6.4.0	>=3.11	>=1.25	>=2.0	>=2023.04
v0.13.0	Web, HTML+ZIP	>=6.3.0	>=3.10	>=1.24	>=1.5	>=2022.09
v0.12.0	Web, HTML+ZIP	>=6.3.0	>=3.10	>=1.23	>=1.5	>=2022.06
v0.11.0	Web, HTML+ZIP	>=6.3.0	>=3.9	>=1.23		
v0.10.0	Web, HTML+ZIP	>=6.3.0	>=3.9	>=1.22		
v0.9.0	Web, HTML+ZIP	>=6.3.0	>=3.8	>=1.21		
v0.8.0	Web, HTML+ZIP	>=6.3.0	>=3.8	>=1.20		
v0.7.0	Web, HTML+ZIP	>=6.3.0	>=3.8	>=1.20		
v0.6.1	Web, HTML+ZIP	>=6.3.0	>=3.8	>=1.19		
v0.6.0	Web, HTML+ZIP	>=6.3.0	>=3.8	>=1.19		
v0.5.0	Web, HTML+ZIP	>=6.2.0	>=3.7	>=1.18		
v0.4.1	Web, HTML+ZIP	>=6.2.0	>=3.7	>=1.17		
v0.4.0	Web, HTML+ZIP	>=6.2.0	>=3.7	>=1.17		
v0.3.1	Web, HTML+ZIP	>=6.1.1	>=3.7			
v0.3.0	Web, HTML+ZIP	>=6.1.1	>=3.7			
v0.2.1	Web, HTML+ZIP	>=6.1.1	>=3.6			
v0.2.0	Web, HTML+ZIP	>=6.1.1	3.6 - 3.8			
v0.1.2	Web, HTML+ZIP	>=6.0.0	3.6 - 3.8			
v0.1.1	Web, HTML+ZIP	>=6.0.0	3.6 - 3.8			
v0.1.0	Web, HTML+ZIP	>=6.0.0	3.6 - 3.8			

*Dev reflects the main branch and is for the upcoming release.

CHAPTER

TWELVE

ECOSYSTEM

PyGMT provides a Python interface to the Generic Mapping Tools (GMT), which is a command line program that provides a wide range of tools for manipulating geospatial data and making publication-quality maps and figures. PyGMT integrates well with the scientific Python ecosystem, with NumPy for its fundamental array data structure, pandas for tabular data I/O and Xarray for raster grids/images/cubes I/O.

In addition to these core dependencies, PyGMT also relies on several optional packages to provide additional functionality for users.

This page was adapted from GeoPandas's Ecosystem page.

12.1 PyGMT dependencies

Asterisk (*) after the package name indicates the package is a required dependency of PyGMT.

12.1.1 NumPy*

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

12.1.2 pandas*

pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real-world data analysis in Python.

12.1.3 Xarray*

Xarray is an open source project and Python package that introduces labels in the form of dimensions, coordinates, and attributes on top of raw NumPy-like arrays, which allows for more intuitive, more concise, and less error-prone user experience.

12.1.4 IPython

IPython provides a rich toolkit to help you make the most of using Python interactively. Its main components are a powerful interactive Python shell and a Jupyter kernel to work with Python code in Jupyter notebooks and other interactive frontends.

PyGMT relies on IPython to provide a rich interactive experience in Jupyter notebooks.

12.1.5 GeoPandas

GeoPandas is an open source project to make working with geospatial data in Python easier. GeoPandas extends the datatypes used by pandas to allow spatial operations on geometric types. Geometric operations are performed by Shapely. GeoPandas further depends on pyogrio for file access and Matplotlib for plotting.

PyGMT doesn't directly rely on GeoPandas, but provides support of GeoPandas's two main data structure, `geopandas.GeoDataFrame` and `geopandas.GeoSeries`, which can be directly used in data processing and plotting functions/methods of PyGMT.

12.1.6 contextily

contextily is a small Python package to retrieve tile maps from the internet. It can add those tiles as basemap to matplotlib figures or write tile maps to disk into geospatial raster files.

In PyGMT, *`pygmt.datasets.load_tile_map`* and *`pygmt.Figure.tilemap`* rely on it.

12.1.7 rioxarray

rioxarray is a geospatial Xarray extension powered by rasterio. Built on top of rasterio, it enables seamless reading, writing, and manipulation of multi-dimensional arrays with geospatial attributes such as coordinate reference systems (CRS) and spatial extent (bounds).

PyGMT relies on rioxarray in several aspects:

1. To save multi-band rasters to temporary files in GeoTIFF format, to support processing and plotting 3-D `xarray.DataArray` images.
2. To write CRS information to the `xarray.DataArray` objects.
3. To reproject raster tiles to the target CRS in *`pygmt.datasets.load_tile_map`*.

Note

We're working towards avoiding temporary files when processing/plotting multi-band rasters in PR #3468.

12.1.8 PyArrow

Apache Arrow is a development platform for in-memory analytics. It contains a set of technologies that enable big data systems to process and move data fast. It specifies a standardized language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware. The Arrow Python bindings (also named "PyArrow") have first-class integration with NumPy, pandas, and built-in Python objects. They are based on the C++ implementation of Arrow.

Note

If you have PyArrow installed, PyGMT does have some initial support for `pandas.Series` and `pandas.DataFrame` objects with Apache Arrow-backed arrays. Specifically, only uint/int/float, date32/date64 and string types are supported for now. Support for Duration types and GeoArrow geometry types is still a work in progress. For more details, see issue #2800.

12.2 PyGMT ecosystem

Various packages rely on PyGMT for geospatial data processing, analysis, and visualization. Below is an incomplete list (in no particular order) of tools which form the PyGMT-related ecosystem.

Note

If your package relies on PyGMT, please let us know or *add it by yourself*.

CHAPTER

THIRTEEN

PYGMT TEAM

We are an international team dedicated to building a Pythonic API for the Generic Mapping Tools (GMT).

All are welcome to become involved with the PyGMT project! For more information about how to get involved, see the *Contributors Guide*. A more complete list of contributors is available in the `AUTHORS.md` file in the source repository.

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. Maintainers are recognized for their responsibilities in maintaining the project, as detailed in the *Maintainers Guide*.

New Distinguished Contributors and Active Maintainers are selected and voted by current Active Maintainers before each release. Maintainers that are inactive for more than one year will be moved to Distinguished Contributors.

13.1 Founders

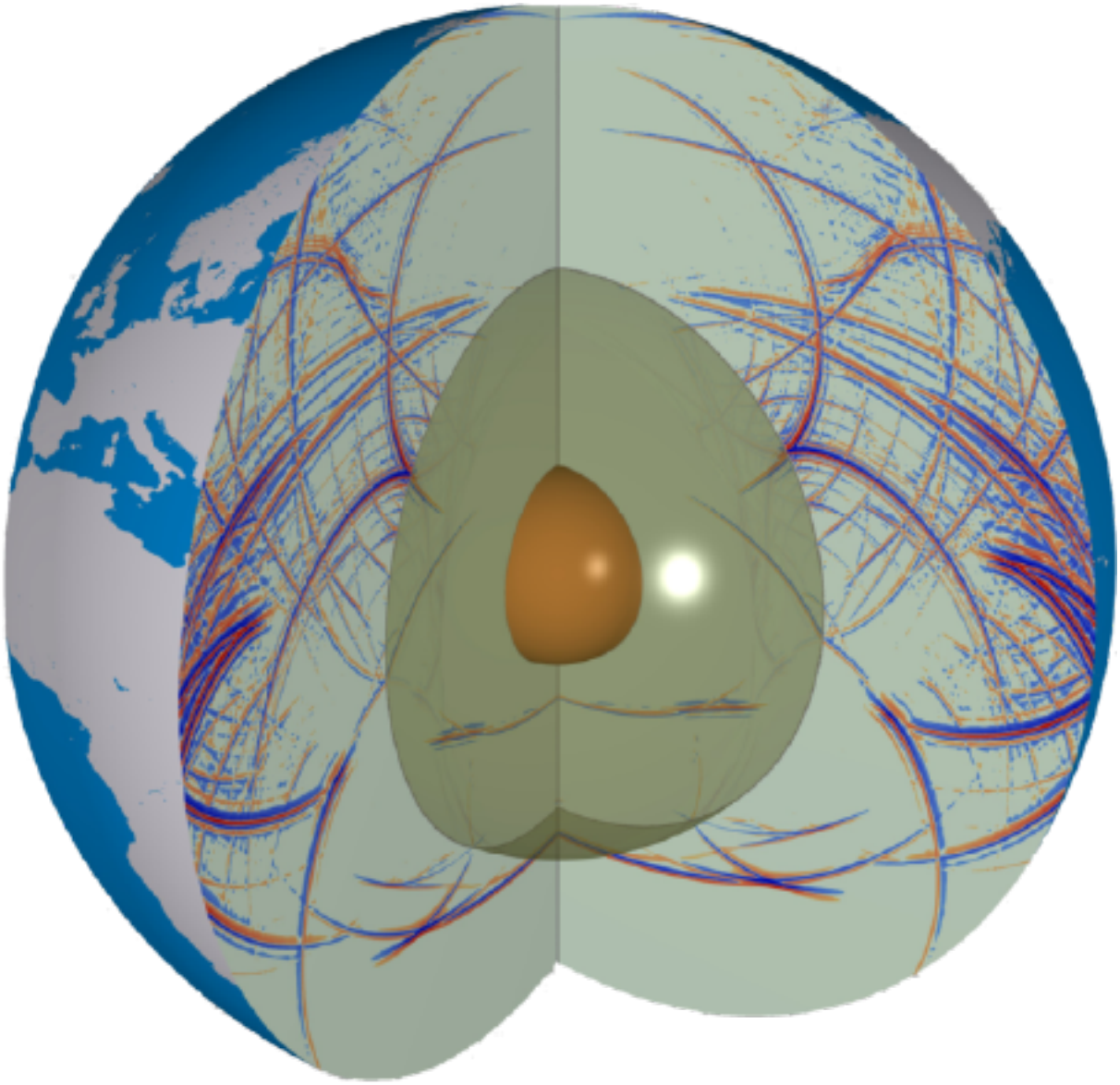


Leonardo Uieda @leouieda



Paul Wessel @PaulWessel

13.2 Active Maintainers



Dongdong Tian @seisman



Wei Ji Leong @weiji14



Michael Grund @michaelgrund

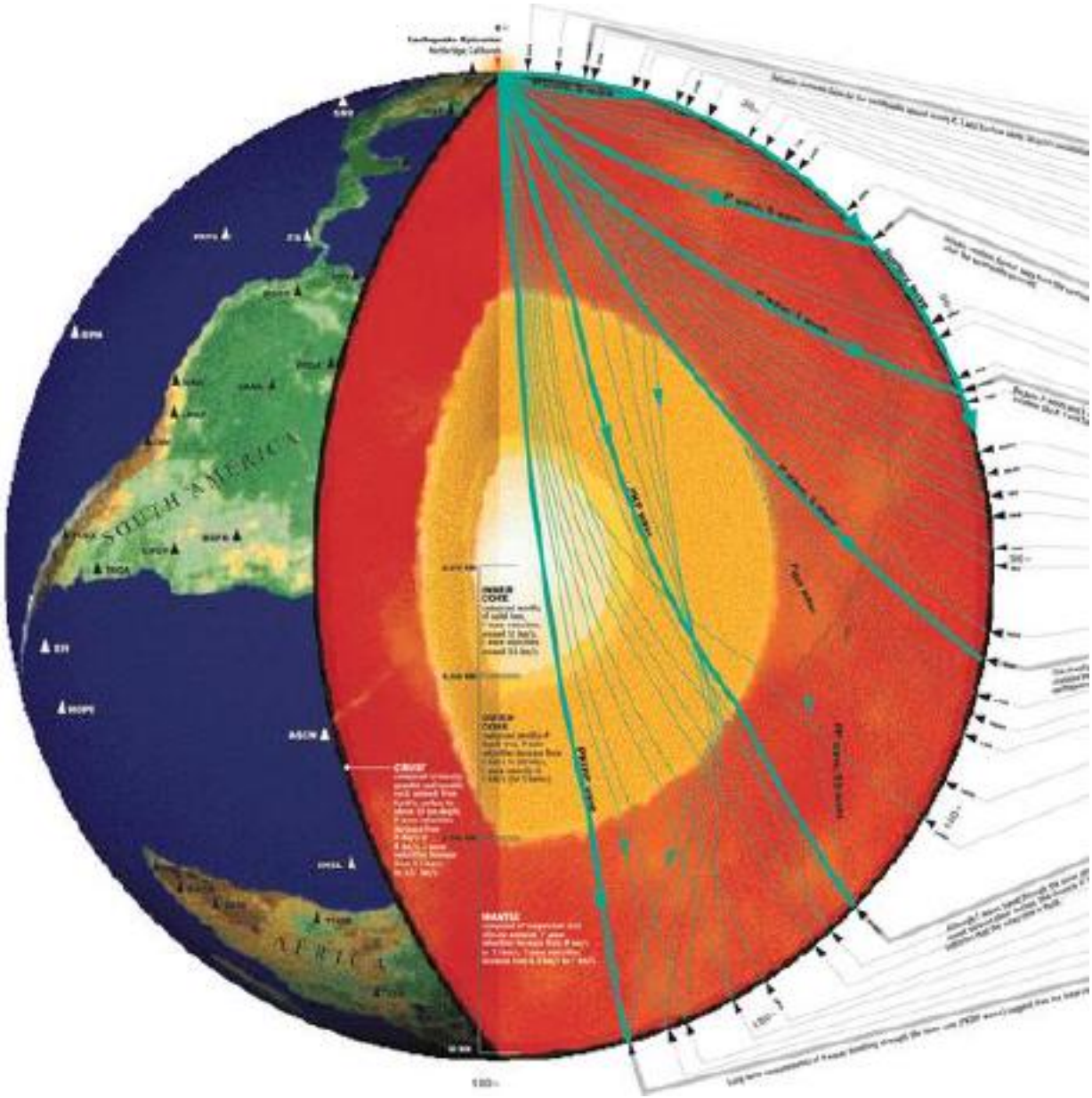


Yvonne Fröhlich @yvonnefroehlich

13.3 Distinguished Contributors



Max Jones @maxrjones



Jiayuan Yao @core-man



Liam Toney @liamtoney



Will Schlitzer @willschlitzer

CHAPTER

FOURTEEN

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.

14.1 Ways to Contribute

14.1.1 Ways to Contribute Documentation and/or Code

- Tackle any issue that you wish! Some issues are labeled as **"good first issue"** 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.

14.1.2 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.

14.1.3 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!

14.2 Providing Feedback

14.2.1 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.

Reporting Upstream Bugs

If you are aware that a bug is caused by an upstream GMT issue rather than a PyGMT-specific issue, you can optionally take the following steps to help resolve the problem:

- Add the line `pygmt.config(GMT_VERBOSE="d")` after your import statements, which will report the equivalent GMT commands as one of the debug messages.
- Either append all messages from running your script to your GitHub issue, or filter the messages to include only the GMT-equivalent commands using a command such as:

```
python <test>.py 2>&1 | awk -F': ' '$2=="GMT_Call_Command string" {print $3}'
```

  where `<test>` is the name of your test script.
- If the bug is produced when passing an in-memory data object (e.g., a pandas.DataFrame or xarray.DataArray) to a PyGMT function, try writing the data to a file (e.g., a netCDF or ASCII txt file) and passing the data file to the PyGMT function instead. In the GitHub issue, please share the results for both cases along with your code.

14.2.2 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.

14.2.3 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.

14.3 General Guidelines

14.3.1 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.

14.3.2 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.

14.3.3 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*.

14.4 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 Miniforge and the `mamba` package manager to install and manage your Python packages. It will make your life a lot easier!

The repository includes a virtual environment file `environment.yml` with the specification for all development requirements to build and test the project. In particular, these are some of the key development dependencies you will need to install to build the documentation and run the unit tests locally:

- git (for cloning the repo and tracking changes in code)
- dvc (for downloading baseline images used in tests)
- pytest-mpl (for checking that generated plots match the baseline)
- sphinx-gallery (for building the gallery example page)

See the `environment.yml` file for the full 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:

```
mamba env create --file environment.yml
```

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):

```
mamba 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
python -m 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).

14.5 Contributing Documentation

14.5.1 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 is 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*.

14.5.2 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 "Details" button of the "docs/readthedocs.org:pygmt-dev" status check, after the building has finished (usually 10-15 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.

14.5.3 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.

14.5.4 Adding example code

Many of the PyGMT functions have example code in their documentation. To contribute an example, add an "Example" header and put the example code below it. Have all lines begin with `>>>`. To keep this example code from being run during testing, add the code `__doctest_skip__ = ["function_name"]` to the top of the module.

Inline code example

Below the import statements at the top of the file:

```
__doctest_skip__ = ["function_name"]
```

At the end of the function's docstring:

```
Example
-------
>>> import pygmt
>>> # Comment describing what is happening
>>> Code example
```

14.5.5 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 that 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.

14.5.6 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:

- Create a `.py` file in the `examples/tutorials/advanced` folder.
- Write the tutorial in "notebook" style with code mixed with paragraphs explaining what is being done. See the other tutorials for the format.

- Choose the most representative figure as the thumbnail figure by adding the comment line `# sphinx_gallery_thumbnail_number = <fig_number>` at the end of 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 pygmt.Figure.show method needs to be called for a plot to be inserted into the documentation.

14.5.7 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 ruff, docstrings must be formatted manually. To play nicely with Jupyter and IPython, **keep docstrings limited to 88 characters** per line.

14.5.8 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 brackets) with the prefix "Default is". Example: [Default is **p**].

14.5.9 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/method listed there.

You can reference functions, classes, modules, and methods 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/6.5/makecpt.html. For GMT configuration parameters, an example is :gmt-term:`COLOR_FOREGROUND` to link to COLOR_FOREGROUND.

Sphinx will create a link to the automatically generated page for that function/class/module/method.

14.6 Contributing Code

14.6.1 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.

14.6.2 Code Style

We use the ruff tool to format the code, so we don't have to think about it. It loosely follow 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 `pre-commit` hooks to enforce 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.

When working on a tutorial or a gallery plot, it is good practice to use code block separators to split a long script into multiple blocks. The separators also make it possible to run the script like a Jupyter notebook in some modern text editors or IDEs. We consistently use `# %%` as code block separators (please refer to issue #2660 for the discussions) and require at least one separator in all example files.

We also use ruff to check the quality of the code and quickly catch common errors.

The `Makefile` contains rules for running the linter checks:

```
make check   # Runs ruff in check mode
```

14.6.3 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 methods, such as pygmt.Figure.plot and pygmt.Figure.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
```

14.6.4 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:

```
@pytest.mark.mpl_image_compare
def test_my_plotting_case():
    """
    Test that my plotting method works.
    """
    fig = Figure()
```

(continues on next page)

(continued from previous page)

```
    fig.basemap(region=[0, 360, -90, 90], projection="W15c", 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
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"
dvc status --remote upstream  # Report which files will be pushed to the dvc remote
dvc push  # Run before git push to enable automated testing with the new images
git 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 method 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
```

CHAPTER

FIFTEEN

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 *Contributors Guide* instead.

15.1 Onboarding/Offboarding Access Checklist

Note that anyone can contribute to PyGMT, even without being added to the GenericMappingTools team. The onboarding items below are for people who would like to make regular contributions, and could benefit from extra permissions to the developer and communication tools we use.

15.1.1 As a Contributor

- Add to the pygmt-contributors team (gives 'write' permission to the repository)
- Add as a collaborator on DAGsHub (gives 'write' permission to dvc remote storage)
- Add as a member on HackMD (for draft announcements) [optional]

15.1.2 As a Maintainer

- Add to the pygmt-maintainers team (gives 'maintain' permission to the repository)
- Add to "Active Maintainers" on the *Team Gallery page*
- Add as a moderator on the GMT forum (to see mod-only discussions) [optional]
- Add as a maintainer on ReadtheDocs [optional]
- Add as a curator to the GMT community on Zenodo (for making releases) [optional]

15.1.3 As an Administrator

- Add to the pygmt-admin team (gives 'admin' permission to the repository)
- Add as an admin on DAGsHub
- Add as a maintainer on PyPI and Test PyPI [optional]

Note: When a maintainer is no longer active (no activity in one year), we will mirror the onboarding access checklist:

- Move from the pygmt-maintainers team to the pygmt-contributors team
- Move from "Active Maintainers" to "Distinguished Contributors" on the *Team Gallery page*
- Remove 'maintain' permission from GMT forum, ReadTheDocs, Zenodo

15.2 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.

15.3 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.

15.4 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 or (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.

15.5 Continuous Integration

We use GitHub Actions continuous integration (CI) services to build, test and manage the project on Linux, macOS and Windows. The GitHub Actions CI are controlled by workflow files located in `.github/workflows`. Here we briefly summarize the functions of the workflows. Please refer to the comments in the workflow files for more details.

- `benchmarks.yml`: Benchmarks the execution speed of tests to track performance of PyGMT functions
- `cache_data.yaml`: Cache GMT remote data files and uplodas as artifacts
- `check-links.yml`: Check links in the repository and documentation
- `ci_docs.yml`: Build documentation on Linux/macOS/Windows and deploy to GitHub
- `ci_doctest.yaml`: Run all doctests on Linux/macOS/Windows

- `ci_tests.yaml`: Run regular PyGMT tests on Linux/macOS/Windows
- `ci_tests_dev.yaml`: Run regular PyGMT tests with GMT dev version on Linux/macOS/Windows
- `ci_tests_legacy.yaml`: Run regular PyGMT tests with GMT legacy versions on Linux/macOS/Windows
- `dvc-diff.yml`: Report changes in test images
- `format-command.yml`: Format the codes using slash command
- `publish-to-pypi.yml`: Publish archives to PyPI and TestPyPI
- `release-baseline-images.yml`: Upload the ZIP archive of baseline images as a release asset
- `release-drafter.yml`: Draft the next release notes
- `slash-command-dispatch.yml`: Support slash commands in pull requests
- `style_checks.yaml`: Code lint and style checks
- `type_checks.yml`: Static type checks

15.6 Continuous Documentation

We use the ReadtheDocs service to preview changes made to our documentation website every time we make a commit in a pull request. The service has a configuration file `.readthedocs.yaml`, with a list of options to change the default behaviour at https://docs.readthedocs.io/en/stable/config-file/index.html.

15.7 Continuous Benchmarking

We use the CodSpeed service to continuously track PyGMT's performance. The `pytest-codspeed` plugin collects benchmark data and uploads it to the CodSpeed server, where results are available at https://codspeed.io/GenericMappingTools/pygmt.

Benchmarking is handled through the `benchmarks.yml` GitHub Actions workflow. It's automatically executed when a pull request is merged into the main branch. To trigger benchmarking in a pull request, add the `run/benchmark` label to the pull request.

To include a new test in the benchmark suite, apply the `@pytest.mark.benchmark` decorator to a test function.

15.8 Dependencies Policy

PyGMT has adopted SPEC 0 alongside the rest of the scientific Python ecosystem, and made a few extensions based on the needs of the project. Please see *Minimum Supported Versions* for the detailed policy and the minimum supported versions of GMT, Python and core package dependencies.

In `pyproject.toml`, the `requires-python` key should be set to the minimum supported version of Python. Minimum supported versions of GMT, Python and core package dependencies should be adjusted upward on every major and minor release, but never on a patch release.

15.9 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 (i.e., v*MAJOR*.*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 the 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.

15.9.1 Deprecating a function parameter

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 exist). A `TODO` comment should also be added to indicate the deprecation period (see below). Here is an example:

```
# TODO(PyGMT>=0.6.0): Remove the deprecated "columns" parameter.
@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`.

15.9.2 TODO comments

Occasionally, we need to implement temporary code that should be removed in the future. This can occur in situations such as:

- When a parameter, function, or method is deprecated and scheduled for removal.
- When workarounds are necessary to address issues in older or upcoming versions of GMT or other dependencies.

To track these temporary codes or workarounds, we use TODO comments. These comments should adhere to the following format:

```
# TODO(package>=X.Y.Z): A brief description of the TODO item.
# Additional details if necessary.
```

The TODO comment indicates that we should address the item when *package* version *X.Y.Z* or later is required.

It's important not to overuse TODO comments for tracking unimplemented features. Instead, open issues to monitor these features.

15.10 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.

15.10.1 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 *Changelog*.

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 and methods, 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 accounts. Sort their names by the number of contributions (including commits and PR reviews) made since the last release (e.g., use `git shortlog vX.Y.Z..HEAD -sne --group=author --group=trailer:co-authored-by`).
8. Update `doc/minversions.md` with new information on the new release version, including a vX.Y.Z documentation link, and minimum required versions of GMT, Python and core package dependencies (NumPy, pandas, Xarray). Follow SPEC 0 for updates.
9. Refresh citation information. Specifically, the BibTeX in `README.md` and `CITATION.cff` needs to be updated with any metadata changes, including the DOI, release date, and version information. Please also 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.

15.10.2 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.

15.10.3 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.

15.10.4 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 sure the minimum support versions of all dependencies are correctly pinned.
5. Make a new branch, commit, and push the changes **to your personal fork**.
6. Create a PR against the original feedstock main.
7. Once the CI tests pass, merge the PR or ask a maintainer to do so.