Geochemistry, Geophysics, Geosystems®

AGU ADVANCING EARTH AND SPACE SCIENCES

METHOD
10.1029/2026GC013105

Yvonne Fröhlich is currently unaffiliated.

Key Points:

- Pythonic interface to the Generic Mapping Tools with intuitive syntax and auto-completion support
- Seamless integration with the scientific Python ecosystem and rich visualization support in Jupyter notebooks
- Free and open-source project maintained by an open, welcoming, and collaborative community

Correspondence to:
D. Tian,
dtian@cug.edu.cn

Citation:
Tian, D., Fröhlich, Y., Leong, W. J., Grund, M., Schlitzer, W., Jones, M., et al. (2026). PyGMT: Bridging Python and the generic mapping tools for geospatial visualization and analysis. *Geochemistry, Geophysics, Geosystems*, *27*, e2026GC013105. https://doi.org/10.1029/2026GC013105

Received 14 APR 2026
Accepted 30 JUN 2026

Author Contributions:
Conceptualization: Dongdong Tian, Yvonne Fröhlich, Wei Ji Leong, Leonardo Uieda
Funding acquisition: Dongdong Tian, Leonardo Uieda, Joaquim Manuel Freire Luis
Project administration: Dongdong Tian, Wei Ji Leong
Software: Dongdong Tian, Yvonne Fröhlich, Wei Ji Leong, Michael Grund, William Schlitzer, Max Jones, Leonardo Uieda, Joaquim Manuel Freire Luis
Supervision: Dongdong Tian
Visualization: Dongdong Tian, Yvonne Fröhlich

PyGMT: Bridging Python and the Generic Mapping Tools for Geospatial Visualization and Analysis

Dongdong Tian[1], Yvonne Fröhlich[2], Wei Ji Leong[3], Michael Grund[4], William Schlitzer[5], Max Jones[6], Leonardo Uieda[7], and Joaquim Manuel Freire Luis[8]

[1]Hubei Subsurface Multi-scale Imaging Key Laboratory, School of Geophysics and Geomatics, China University of Geosciences, Wuhan, China, [2]Karlsruhe Institute of Technology, Karlsruhe, Germany, [3]Development Seed, Wellington, New Zealand, [4]SNP Innovation Lab GmbH, Heidelberg, Germany, [5]Independent Researcher, Cherry Hill, NJ, USA, [6]Development Seed, Washington, DC, USA, [7]Universidade de São Paulo, São Paulo, Brazil, [8]IDL, University of Algarve, Faro, Portugal

Abstract PyGMT is a free and open-source Python library that provides a high-level interface to the Generic Mapping Tools (GMT), a widely used command-line toolset for geospatial data processing, analysis, and visualization in the Earth, ocean, and planetary sciences. By bridging GMT with the Python ecosystem, PyGMT enables users to access GMT's powerful visualization and analysis capabilities directly within Python-based workflows, making them more accessible to a broader audience. PyGMT supports diverse data sources, including local and remote files, as well as familiar data structures such as numpy.ndarray, pandas.DataFrame, and xarray.DataArray objects, allowing users to process data efficiently and create a wide variety of publication-quality figures. Compared to GMT, PyGMT provides a Pythonic interface with intuitive parameters and arguments, lowering the barrier to entry for both new and experienced users, and extends GMT's functionality by integrating seamlessly with the broader scientific Python ecosystem. The project is actively maintained as a community-driven effort under the BSD 3-Clause License, with source code and issue tracking hosted on GitHub and user support provided through a Discourse forum. It emphasizes reliability, usability, and sustainability while fostering an open and welcoming community that encourages contributions of all kinds.

Plain Language Summary PyGMT is a free, open-source Python library that simplifies the creation of high-quality maps and visualizations for various data types, including Cartesian and geographic data, time series, and geospatial data sets. It provides a user-friendly interface to the Generic Mapping Tools (GMT), a widely used toolset in the Earth, ocean, and planetary sciences. While GMT is powerful, its command-line interface can be challenging to learn, read, and interpret. PyGMT connects Python with GMT by allowing users to access GMT's functionality directly in Python and enhancing them with the broader Python ecosystem. With PyGMT, users can work with familiar Python data structures and generate high-quality figures efficiently. Maintained by a global community, PyGMT is available under a permissive open-source license. Anyone is welcome to contribute code, report issues, or improve documentation. The project follows modern software development practices, aiming to build a reliable and sustainable tool while maintaining an open and welcoming community.

1. Introduction

As scientific research increasingly relies on large and heterogeneous data sets, reproducible and programmable visualization tools have become essential for data exploration, analysis, and communication (Baker, 2016; Crameri, 2018; Hunter, 2007; Perez & Granger, 2007; Wilkinson et al., 2016). The Generic Mapping Tools (GMT) is an open-source software package for data processing, analysis, and visualization that has been widely used in the Earth, ocean, and planetary sciences since its initial public release in 1991 (Wessel & Smith, 1991, 1995, 1998; Wessel et al., 2013, 2019). GMT supports a broad range of data types, including Cartesian and geographic data, time series, and geospatial data sets, as well as numerous map projections. It uses the PostScript language (Adobe Systems Inc, 1990) as its backend to generate high-quality, publication-ready vector graphics that can be converted to static formats such as PDF, PNG, and JPEG, or to animations in GIF, MPEG4, and WebM (Wessel et al., 2024). Written in the C programming language and released under the GNU Lesser General Public License (LGPL) version 3.0, GMT is cross-platform and runs on all major operating systems, including Linux, macOS, Windows, and BSD. Inspired by the modular design philosophy of UNIX, GMT provides a command-

Writing – original draft: Dongdong Tian, Yvonne Fröhlich, Wei Ji Leong
Writing – review & editing: Dongdong Tian, Yvonne Fröhlich, Wei Ji Leong, Michael Grund, William Schlitzer, Max Jones, Leonardo Uieda, Joaquim Manuel Freire Luis

line interface (CLI) comprising more than 150 modules, each dedicated to a specific task such as plotting coastlines or gridding data. These modules can be combined in shell scripts to create reproducible workflows ranging from simple map generation to advanced data analysis.

Despite its powerful capabilities, GMT presents certain challenges for users, particularly beginners. Its CLI syntax, with numerous short-option flags and modifiers, can make commands difficult to read, write, and interpret, especially in advanced use cases. In addition, the reliance on CLI-based workflows introduces a steep learning curve for users unfamiliar with CLI and shell scripting. Another limitation is GMT's dependency on file-based input and output (I/O). While computing environments such as MATLAB, Python, and Julia can invoke GMT's functionality through system calls, this approach typically relies on temporary disk files for data exchange, which can become a performance bottleneck when working with large data sets. This approach also complicates integration with scripting environments because users have to manage intermediate disk files manually.

To address these limitations, GMT 5 introduced an Application Programming Interface (API) along with a virtual file mechanism (Wessel et al., 2013). The API provides direct access to GMT's core functionality, enabling the development of higher-level interfaces and wrappers in other programming languages. The virtual file mechanism enables GMT modules to read from and write to in-memory data structures, eliminating the need for intermediate disk files and streamlining data exchange. Successful applications of this architecture include the GMT/MATLAB Toolbox (Wessel & Luis, 2017) and GMT.jl (Luis & Wessel, 2018), which integrate GMT's geospatial visualization and analysis capabilities into MATLAB and Julia workflows, respectively. The release of GMT 6 further improved its usability through a new "modern mode" syntax, which simplifies GMT workflows and enhances the overall user experience (Wessel et al., 2019). Together, these developments demonstrate the flexibility of GMT's API and its potential to enhance GMT's accessibility through deeper integration with diverse computing environments.

The Python programming language (van Rossum & de Boer, 1991) has emerged as one of the leading languages in scientific computing due to its simplicity and rich ecosystem. The scientific Python ecosystem includes general-purpose libraries for numerical computing, data analysis, and visualization, such as NumPy (Harris et al., 2020), SciPy (Virtanen et al., 2020), pandas (McKinney, 2010), xarray (Hoyer & Hamman, 2017), and Matplotlib (Hunter, 2007), as well as specialized geospatial libraries such as GeoPandas (Van den Bossche et al., 2026), rioxarray (Snow et al., 2026), Verde (Uieda, 2018), and Cartopy (Met Office, 2010). These libraries have become essential components of modern scientific workflows. However, research in the Earth, ocean, and planetary sciences often requires specialized geospatial processing capabilities because geographic data are defined on a spherical or spheroidal surface rather than a flat plane. Consequently, geographic data present challenges that are not encountered in Cartesian coordinate systems. For example, longitude wraps at the dateline, coordinate systems become singular at the poles, and equal spacing in longitude does not correspond to equal physical spacing at different latitudes.

GMT provides a specialized collection of tools for geospatial visualization and analysis. Originally developed for Earth science applications, GMT was designed from the beginning to support both Cartesian and geographic data, ranging from simple x-y data and time series to regional and global geographic data sets. On the visualization side, GMT offers support for a wide range of map projections, publication-quality cartographic plots in either 2-D or 3-D perspective view, and robust handling of geographic data, including those that cross the dateline. On the analysis side, GMT's data processing modules explicitly account for the spherical or spheroidal geometry in many calculations, such as gridding, filtering, and sampling. GMT also implements specialized algorithms such as surface, which uses continuous-curvature splines in tension for gridding scattered data (W. H. F. Smith & Wessel, 1990), and greenspline, which interpolates data using Green's functions for splines in 1-D to 3-D (Wessel, 2009; Wessel & Becker, 2008). While similar geospatial analyses and visualizations can often be achieved using combinations of existing Python libraries, GMT provides an integrated toolkit that simplifies many common geospatial workflows and reduces the amount of code required. This convenience may in some cases come with rendering overhead because GMT generates publication-quality PostScript output that can subsequently be converted to other graphics formats. GMT's primary strengths nevertheless lie in its support for geographic data, publication-quality visualization, and robust geospatial processing capabilities.

Recognizing Python's potential to broaden GMT's accessibility, the development of PyGMT was initiated in 2017 as a high-level Python interface to GMT (Uieda & Wessel, 2017a). PyGMT integrates GMT's powerful functionality for data processing and visualization with a Pythonic interface, enabling users to create publication-

ready figures and perform advanced geospatial processing directly within Python workflows. Since its inception in 2017, PyGMT has evolved into a mature and feature-rich library that lowers the barrier for Python users to access GMT's functionality while providing existing GMT users with seamless integration into the broader scientific Python ecosystem.

In this paper, we document the design principles of PyGMT and describe how it integrates with the scientific Python ecosystem to enable reproducible geospatial workflows. Through illustrative examples, we demonstrate PyGMT's capabilities and its role in bridging the gap between GMT's powerful backend and Python's more user-friendly interface. We also highlight the project's community-driven and open development model, which welcomes contributors from diverse backgrounds and experiences, as well as the adoption of best practices in modern open-source software development, ensuring PyGMT's long-term reliability, usability, and sustainability.

2. Project Goals and Design

Since its inception, the primary goal of PyGMT has been to make GMT more accessible to a broader user base by providing a Pythonic interface that simplifies interaction with its powerful data processing and visualization capabilities (Uieda & Wessel, 2017b). A secondary goal is to seamlessly integrate with the scientific Python ecosystem by supporting standard data structures such as numpy.ndarray, pandas.DataFrame, and xarray.DataArray objects, while enabling rich outputs in interactive environments such as Jupyter notebooks. In this section, we outline the underlying design principles and technical details that allow PyGMT to accomplish its goals efficiently and reliably.

2.1. GMT API Access via ctypes

The most straightforward way to access GMT from Python is through system calls in which Python scripts invoke GMT as subprocesses. However, this approach introduces significant overhead due to the cost of process creation and the requirement to use temporary files for data exchange.

PyGMT adopts a more efficient strategy by leveraging ctypes, Python's built-in foreign function interface (FFI) library, to interact directly with the GMT shared library. ctypes can automatically locate the GMT shared library (e.g., *libgmt.so* on Linux, *libgmt.dylib* on macOS, and *gmt.dll* on Windows) from standard system paths, ensuring PyGMT works across different configurations, whether GMT is installed by package managers or compiled from source. If automatic detection fails, users can manually specify the library path by setting the *GMT_LIBRARY_PATH* environment variable.

Once the GMT shared library is found, ctypes loads it into memory, providing Python with direct access to all GMT API functions. It also allows NumPy arrays, the fundamental data structure in Python for numerical computing (Harris et al., 2020), to be referenced in GMT API functions via pointers to their underlying data buffers. This enables efficient data exchange by sharing memory between Python and the underlying C library (i.e., between PyGMT and GMT), minimizing overhead and avoiding the need for custom C code or additional compilation steps within PyGMT.

2.2. Efficient Data Exchange With Virtual Files

Building on the efficient API access provided by ctypes, PyGMT further facilitates data exchange through GMT's virtual file mechanism. This mechanism provides a unified interface, as GMT modules can read from and write to either physical files or in-memory data buffers (Wessel et al., 2013). Under the hood, virtual files are tied to GMT's core data containers, which are implemented as C structures for storing different types of data. GMT defines six primary data containers: GMT_DATASET for tabular data organized into tables and segments; GMT_GRID for regularly gridded data; GMT_IMAGE for raster images; GMT_CUBE for multi-layered grids; GMT_PALETTE for color palette tables; and GMT_POSTSCRIPT for chunks of PostScript code. For more details, see the GMT/MATLAB paper (Wessel & Luis, 2017). Two auxiliary containers, GMT_MATRIX and GMT_VECTOR, support user-defined data: GMT_MATRIX stores homogeneous 2-D arrays (e.g., numeric arrays with a single data type), while GMT_VECTOR represents a collection of 1-D column vectors, each of which can have different data types.

PyGMT uses the virtual file mechanism by converting native Python data structures into their corresponding GMT data containers and vice versa. Specifically, numeric arrays in tabular Python objects, such as numpy.ndarray and pandas.DataFrame, are mapped to GMT_VECTOR, GMT_MATRIX, or GMT_DATASET,

(a) GMT CLI

```
gmt begin GMT_map png
    gmt basemap -JR7c -R0/360/-90/90 -B
    gmt coast -Gtan -Slightblue
    echo PyGMT | gmt text -F+f40p,AvantGarde-Book,red@75+cMC
gmt end show
```

(b) PyGMT

```
import pygmt
fig = pygmt.Figure()
fig.basemap(projection="R7c", region=[0, 360, -90, 90], frame=True)
fig.coast(land="tan", water="lightblue")
fig.text(position="MC", text="PyGMT", font="40p,AvantGarde-Book,red@75")
fig.show()
fig.savefig("PyGMT_map.png")
```

(c)

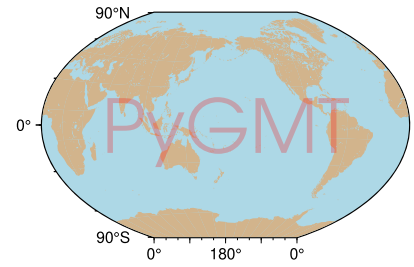

Figure 1. Example comparison of GMT CLI and PyGMT scripts that generate the same plot. (a) GMT CLI script written in Bash. (b) Equivalent PyGMT script. (c) Resulting map generated by both scripts.

depending on the structure and heterogeneity of the data. Similarly, xarray.DataArray objects are mapped to GMT_GRID, GMT_IMAGE, or GMT_CUBE, depending on their dimensions and metadata. This mapping allows Python users to work with familiar data structures, while PyGMT handles the conversion to GMT's internal data structures behind the scenes.

When a user passes a supported Python object to PyGMT, the object is first converted into the appropriate GMT data container. A virtual file is then registered in memory to reference this container, and its name can be passed to GMT modules as if it were a regular file on disk. For output, the process is reversed: PyGMT creates an empty data container, registers a virtual file, and the GMT module writes its result into the virtual file (i.e., the in-memory data container). PyGMT then retrieves the data from the container and converts it back into a native Python object. During these conversions, PyGMT passes data directly between Python and the GMT library, avoiding unnecessary duplication whenever possible to reduce memory consumption. While GMT itself can access input arrays directly and may choose to duplicate them to prevent unexpected modifications of the original data, PyGMT explicitly sets API flags to guarantee that users' input arrays remain unchanged, although rare exceptions may exist. This mechanism enables efficient bidirectional in-memory data exchange between PyGMT and GMT, avoiding temporary disk files and reducing the overhead and complexity associated with file-based I/O.

2.3. Pythonic Syntax and Auto-Completion

While powerful and flexible, the GMT CLI syntax can be challenging to learn and interpret, even for experienced users. In the GMT CLI, each GMT command begins with **gmt**, followed by the name of a GMT module (such as coast or text) and a long, whitespace-separated string of arguments. These arguments typically consist of single-letter option flags (e.g., -R for setting the region of interest and -J for setting the map projection parameters) followed by corresponding values and optional modifiers (e.g., `-R0/360/-90/90` and `-F+f40p,Avant-Garde-Book,red@75+cMC`). Figures are constructed by executing a series of plotting commands between `gmt` begin and `gmt` end. The `gmt` begin command starts a modern-mode GMT session and specifies the desired figure name and formats. Subsequent plotting commands incrementally add layers to a hidden PostScript file, which is finally converted to the specified graphic format by `gmt` end. This syntax, inherited from traditional UNIX-style command-line tools, favors brevity over readability. An example GMT CLI script is shown in Figure 1.

PyGMT improves usability and readability through three key strategies: introducing a central Figure class for plotting, adopting descriptive parameter names, and supporting readable Python-native argument types. Figure 1 also shows an equivalent PyGMT script that produces the same plot as the GMT CLI script.

Central Figure class: PyGMT provides a central Figure class to emulate GMT's figure creation workflow by wrapping all GMT plotting modules as methods. A figure is created by instantiating a Figure object, followed by sequential calls to plotting methods such as Figure.coast() for drawing coastlines or Figure.text() for typesetting text strings. Layers are added incrementally, and the figure can be displayed inline in environments such as Jupyter notebooks using Figure.show() or saved to supported graphics formats via Figure.savefig().

Descriptive parameter names: Rather than using GMT's single-letter option flags as parameter names, PyGMT adopts more descriptive and intuitive parameter names. For example, the GMT options -R and -J are replaced by region and projection. Internally, an alias system translates these readable parameter names into the corresponding single-letter GMT option flags. The alias system also simplifies complex options by splitting them into

multiple, logically grouped parameters. For instance, the text module option `-F+f40p,AvantGarde-Book,red@75+cMC`, which sets text size, font, color, and position, is expressed in PyGMT as `font="40p,AvantGarde-Book,red@75"` and `position="MC"` (where "MC" stands for "Middle Center"), making the command much easier to interpret. For commonly shared options across modules, PyGMT provides auxiliary classes to encapsulate related parameters and avoid parameter conflicts. Examples include the Position and Box classes, which control the placement and appearance of boxes for GMT embellishments, such as a map scale or a legend, as well as the Frame and Axis classes, which provide a Pythonic way to specify plot frame settings. Examples of these classes are demonstrated in Section 3.

Python-native arguments: Beyond descriptive parameter names, PyGMT also supports Python-native data types to improve the readability of arguments. Because the GMT C API requires arguments to be passed as strings, the alias system is also responsible for translating Python-native values into appropriate strings. General translation rules include the following: (a) None or False means an option or a modifier is omitted, typically resulting in the default behavior; (b) True enables an option or a modifier without further arguments (e.g., `frame=True` is translated to -B); (c) sequence-like values are joined as slash- (or comma-) separated strings if a parameter expects such a format (e.g., `region=[0, 360, −90, 90]` becomes -R0/360/-90/90); (d) certain long-form descriptive strings are converted to GMT's short-form, single-letter codes (e.g., `resolution="high"` becomes -Dh in the coast module); (e) other data types (such as datetime objects) are automatically converted to strings if possible.

While descriptive parameters and arguments greatly improve readability (see Figure 1 for a comparison), they also result in longer parameter names and argument strings, which may increase typing effort and the likelihood of typographical errors. Auto-completion, available in most modern text editors, integrated development environments (IDEs), and interactive computing environments such as Jupyter notebooks, helps mitigate this trade-off by suggesting available parameters and valid argument values as users type. To fully leverage this feature, PyGMT provides extensive static type hints throughout its codebase, enabling more accurate suggestions and enhancing the user experience.

2.4. Integration With the Scientific Python Ecosystem

One of PyGMT's key advantages is its seamless integration with other Python libraries, particularly those in the scientific Python ecosystem (Figure 2). The ecosystem includes foundational packages such as NumPy (Harris et al., 2020), pandas (McKinney, 2010), and xarray (Hoyer & Hamman, 2017), with NumPy serving as the basis for array-based computations, pandas facilitating tabular data processing, and xarray handling multi-dimensional arrays with labeled dimensions. Building on these foundational packages, PyGMT further extends its functionality through integration with geospatial tools built on this ecosystem, including rioxarray for reading and writing GeoTIFF and other raster formats with support for coordinate reference systems (Snow et al., 2026), contextily for fetching map tiles such as *OpenStreetMap*, and GeoPandas for handling vector geospatial data (Van den Bossche et al., 2026). PyGMT also supports PyArrow, which can handle large data sets in a more memory-efficient way. PyGMT has been adopted by several Python packages and projects for visualization and analysis workflows. Examples include SHTOOLS for spherical harmonic analysis (Wieczorek & Meschede, 2018), RT-EQcorrscan for real-time earthquake detection and analysis (Chamberlain et al., 2020), GPlately for tectonic plate reconstruction workflows (Mather et al., 2024), and PolarToolkit for polar geospatial visualization and processing (Tankersley, 2024).

While the GMT CLI typically operates on data by reading from and writing to disk files, PyGMT retains this capability and extends it with more flexible I/O options, which are enabled by the virtual file mechanism described in Section 2.2. For functions expecting tabular data, users can provide either a file path or a tabular data structure, such as a dictionary, a 2-D numpy.ndarray object, or a pandas.DataFrame object, via the data parameter, or supply individual columns (scalars or 1-D array-like objects) to parameters such as x, y, and z. Similarly, functions that expect raster data accept either a file path or an xarray.DataArray object. Regarding output, tabular data can be stored in a disk file or returned as a pandas.DataFrame or numpy.ndarray object, while raster data can be written to formats such as netCDF or GeoTIFF or returned as an xarray.DataArray object.

Internally, GMT supports signed and unsigned integers, floating-point numbers, strings, and datetimes. On top of this, PyGMT supports a wide range of NumPy data types (i.e., *dtypes*), which are mapped to the corresponding C data types used internally by GMT. These include signed and unsigned integers (e.g., numpy.int8, numpy.uint64), floating-point numbers (numpy.float32 and numpy.float64), datetimes (numpy.datetime64 with different

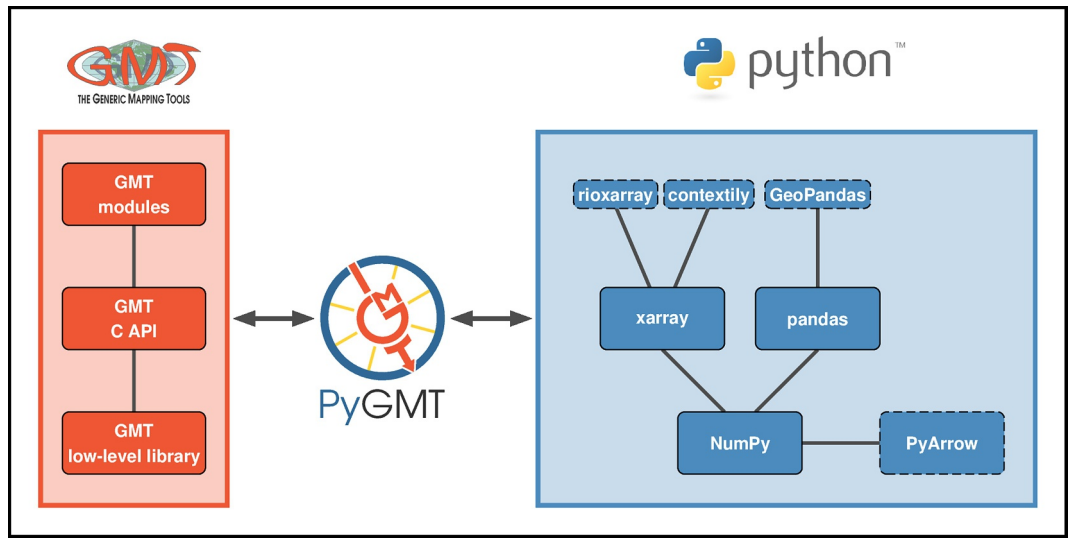

Figure 2. The PyGMT ecosystem. PyGMT serves as a bridge between GMT and the scientific Python ecosystem, enabling bidirectional data flow. Libraries in the Python ecosystem are organized from fundamental array libraries (NumPy and PyArrow) to general-purpose high-level packages (xarray and pandas) and geospatial tools (rioxarray, contextily, and GeoPandas). Libraries enclosed in dashed boxes are optional dependencies that can extend PyGMT's functionality when installed. The GMT and PyGMT logos can be generated via the Figure.logo() and Figure.pygmtlogo() methods, respectively. The Python logo is a registered trademark of the Python Software Foundation.

resolutions), time intervals (numpy.timedelta64 with different resolutions), and strings (numpy.str_). In addition, PyGMT supports various dtypes provided by pandas and PyArrow as long as they can be converted to NumPy arrays. Examples include pandas's nullable integer dtype Int64, which can handle missing values, and pandas's datetime64 dtype, which supports time zones. This flexibility allows PyGMT to work with data from diverse sources.

PyGMT also offers deep integration with xarray by providing the gmt accessor for xarray.DataArray objects. The gmt accessor stores essential grid metadata, such as registration (pixel or gridline) and grid type (Cartesian or geographic), which are critical for grid processing and plotting but are not natively supported by xarray. The grid metadata can be accessed or modified via attributes such as *grid*.gmt.registration and *grid*.gmt.gtype. The accessor also enables direct application of GMT's grid operations on xarray.DataArray objects. For example, a grid can be filtered using either the functional approach pygmt.grdfilter(*grid*) or the accessor approach *grid*.gmt.filter(). Additionally, PyGMT implements an xarray backend entry point that enables reading of grids and images using GMT as the underlying engine. This allows users to read raster data via xarray.open_dataarray() or xarray.open_dataset(). With the GMT engine, xarray can directly access GMT's remote data sets, specified by file names starting with @ (e.g., @earth_relief_01d), and automatically initialize the gmt accessor for easy access to GMT-specific grid metadata and functionality.

2.5. Python Wrappers for GMT Modules

One of the major development efforts of PyGMT has been to wrap GMT modules with a Pythonic interface. As of GMT v6.6.0, GMT provides 98 general-purpose core modules and 58 domain-specific supplementary modules (Wessel et al., 2025). Among these, 33 modules (25 core and 8 supplementary) focus on plotting, while the remainder support non-plotting operations such as processing tabular or gridded data. In PyGMT, all GMT plotting modules are wrapped as methods of the central Figure class, whereas non-plotting modules are exposed as functions or classes in the pygmt namespace (e.g., pygmt.grdfilter() for filtering a grid). In most cases, the names of these functions, methods, and classes match the corresponding GMT module names, making the interface familiar and intuitive for users transitioning from the GMT CLI. Some naming exceptions exist for clarity. For example, pygmt.config() wraps the gmtset module, which is used to configure GMT's default settings.

Although PyGMT primarily wraps GMT modules, it does not strictly adhere to a one-to-one mapping. GMT modules are sometimes designed to perform multiple tasks within a single, complex command, reflecting the

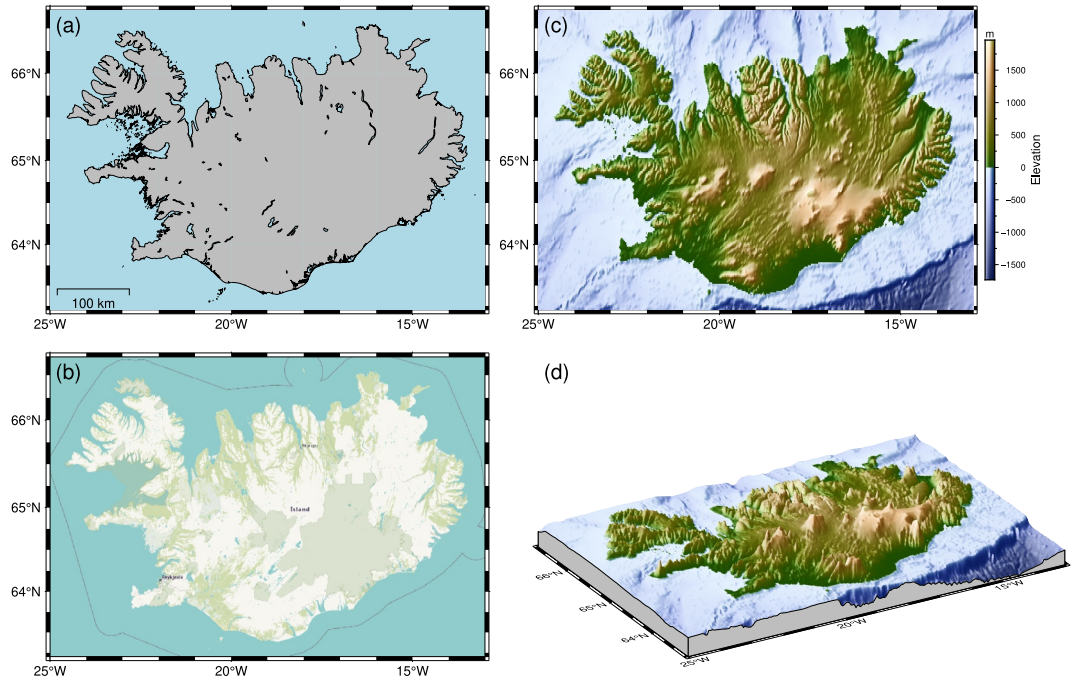

Figure 3. Four different types of geographic background base maps of Iceland. (a) Map showing coastlines and land/water masses; (b) web tile map from *OpenStreetMap*, with geographic features and place names; (c) 2-D relief map of topography and bathymetry; (d) 3-D perspective map of topography and bathymetry.

higher development cost of creating a new module compared to extending an existing one. This design can result in feature-rich modules and duplicated functionality across modules. For example, both the basemap and coast modules provide the same functionality for adding map scale bars, directional roses, and magnetic roses. Another example is the plot module, which supports plotting lines, polygons, and a wide variety of symbol markers, making it one of the most complex GMT modules.

To improve usability and simplify the interface, PyGMT splits certain complex GMT modules into multiple smaller, single-purpose methods. For example, Figure.hlines() and Figure.vlines() wrap the plot module to focus on plotting horizontal and vertical lines. Figure.scalebar(), Figure.directional_rose(), and Figure.magnetic_rose() wrap the basemap module and focus on adding map scale bars, directional roses, and magnetic roses, respectively, consolidating duplicated functionality in the basemap and coast modules. Beyond the functionality offered by GMT modules, PyGMT also implements new functionality such as Figure.tilemap(). This method integrates GMT's grdimage module with the Python library contextily, allowing the creation of basemaps from XYZ tile services such as *OpenStreetMap*.

3. Examples

PyGMT brings GMT's geospatial analysis and visualization capabilities into Python, enabling users to create the high-quality maps and figures for which GMT is known, while benefiting from the flexibility of Python workflows. By supporting Python-native data structures and seamlessly integrating with the scientific Python ecosystem, PyGMT fits naturally into modern scientific workflows. In this section, we present several illustrative examples showcasing how PyGMT can be used to process, analyze, and visualize geospatial data and produce a wide variety of scientific figures. These examples are available in Jupyter notebooks (See Availability Statement).

3.1. Base Maps of Iceland

Our first example illustrates several types of geographic base maps that PyGMT can generate for the same region, using Iceland as an example. These include a simple map showing coastlines and land/water masses (Figure 3a), a map using online XYZ tiles (Figure 3b), and 2-D and 3-D relief maps of topography and bathymetry (Figures 3c and 3d).

```python
import pygmt
from pygmt.params import Position

region = [-25, -13, 63.2, 66.7]

fig = pygmt.Figure()
with fig.subplot(
    nrows=2,
    ncols=2,
    subsize=("12c", "8.5c"),
    margins=("0.4c", "0.2c"),
    frame="WSen",
    tag="(a)",
    tag_position=Position("TL", offset=(0.15, 0.3)),
    tag_orientation="vertical",
):
    # Top left
    fig.basemap(region=region, projection="M?", panel=0)
    fig.coast(shorelines=True, land="gray", water="lightblue", resolution="high")
    fig.scalebar(length="100k")

    # Bottom left
    fig.tilemap(region=region, projection="M?", zoom=7, panel=1)

    # Top right
    grd_relief = pygmt.datasets.load_earth_relief(resolution="01m", region=region)
    fig.basemap(region=region, projection="M?", panel=2)
    fig.grdimage(grid=grd_relief, cmap="SCM/oleron", shading=True)
    fig.colorbar(
        annot=500, tick=250, label="Elevation", unit="m", shading=True,
        position=Position("MR", cstype="outside"),
    )

    # Bottom right
    fig.basemap(region=region, projection="M?", perspective=(-150, 25), panel=3)
    fig.grdview(
        grid=grd_relief,
        cmap="SCM/oleron",
        surftype="image",
        shading=True,
        zsize="1.5c",
        facade_fill="gray",
        perspective=True,
    )
fig.show()
```

We begin by importing the pygmt package, creating a Figure object, and setting up a 2 × 2 subplot layout using Figure.subplot(). Each subplot is 12 cm wide and 8.5 cm high, with margins of 0.4 and 0.2 cm in the x- and y-directions, respectively. Subplots are configured with `frame="WSen"`, which draws annotated axes on the west (**W**) and south (**S**) sides and unannotated axes on the east (**e**) and north (**n**) sides. Uppercase letters denote annotated axes, whereas lowercase letters indicate axes drawn without annotations. Subplots are tagged in the form of "(a)" using the tag parameter and placed at the top-left ("TL") corner of each subplot, with configurable offsets in the x- and y-directions. By default, subplots are arranged and tagged horizontally, but this can be changed to vertical using the tag_orientation parameter.

In the top-left panel, we use Figure.basemap() to define the study region `(region=[−25, −13, 63.2, 66.7])`, corresponding to longitudes 25°–13°W and latitudes 63.2°–66.7°N, and select a Mercator projection

(`projection="M?"`). The question mark instructs PyGMT to automatically determine the map dimension based on the subplot size and margins. The `panel=0` argument activates the first panel (top-left; panel indices start from zero), and all subsequent plotting commands apply to this panel until a different panel is selected. We then use Figure.coast() to plot a simple map with coastlines drawn with the default pen, and land and water masses filled in gray and light blue, respectively. The `resolution="high"` argument instructs PyGMT to use the high-resolution coastline data set provided with GMT (Wessel & Smith, 1996). A scale bar representing 100 km is added at the lower-left corner using Figure.scalebar().

Next, we activate the second panel (bottom-left, because the subplots are arranged vertically in this example) and use the Figure.tilemap() method to plot a map using online XYZ tiles. The `zoom=7` argument sets the tile resolution level, with higher values for higher-resolution tiles. By default, the tile provider is *OpenStreetMap Humanitarian*, although several other tile providers are also supported. This functionality is not available in GMT itself but is provided in PyGMT by integrating the Python package contextily with GMT's grdimage module.

We then use the pygmt.datasets.load_earth_relief() function to access Earth relief data at 1 arc-minute resolution. The underlying data sets are provided by GMT and stored on the GMT remote data server or mirrors, while PyGMT provides utility functions to access them and load them as xarray.DataArray objects. In addition to Earth relief data sets, GMT also provides other Earth and planetary data sets, such as Earth geoid and Mars relief grids. The data set can be visualized in multiple ways. In the top-right panel, we use the Figure.grdimage() method to visualize the data set as a shaded relief map by applying the *oleron* color palette table (CPT) from the *Scientific colour maps* package (Crameri, 2023). The corresponding colorbar is added to the middle-right ("MR") outside the plot using Figure.colorbar() with customized annotations and labels.

In the bottom-right panel, the same data set is visualized in a 3-D perspective view. The base map is set up with a perspective specified by an azimuth of −150° and an elevation of 25° (viewed from the southeast and slightly above). The grid is then rendered using Figure.grdview() in "image" style (`surftype="image"`), with the *oleron* CPT and automatic shading enabled. The vertical scale of the relief is set to 1.5 cm (`zsize="1.5c"`), and `perspective=True` ensures that the surface is rendered using the previously specified viewing geometry. By default, the minimum z value of the grid is used as the reference plane, although this can be customized. The frontal facade between this plane and the relief perimeter is filled in gray.

3.2. Seismicity Along the Andaman-Sumatra-Java Subduction Zone

Our second example presents a typical seismological map showing the seismicity of intermediate-depth and deep-focus earthquakes (hypocentral depth ≥ 70 km) with magnitude $M \geq 5$ along the Andaman-Sumatra-Java subduction zone from 2000 to 2025 (Figure 4). Earthquake catalogs are available in various formats. For simplicity, the global catalog is obtained from the United States Geological Survey (USGS, https://www.usgs.gov) in CSV format using the Python requests package and loaded into a pandas.DataFrame object using pandas.read_csv(). The pandas.DataFrame object enables flexible data manipulation, such as removing unneeded columns, adding new ones, and selecting a subset of events based on specific criteria. In this example, we use pandas to extract only the records within the study region. Although the catalog contains many columns, our analysis focuses on the *latitude*, *longitude*, *depth*, and *mag* columns. Epicenters are typically plotted as circles, with marker size scaled exponentially by magnitude and color representing hypocentral depth. We also sort the records in descending order of magnitude so that larger earthquakes (therefore larger circles) are drawn first, preventing them from obscuring smaller ones.

As before, Figure.basemap() sets the study region and applies the Mercator projection, while Figure.coast() draws coastlines and land masses as the base map. Seismicity is then overlaid using Figure.plot(). Each earthquake is plotted at its epicenter by passing event longitudes and latitudes to the x and y parameters, respectively. The argument `style="c"` specifies circles as the plotting symbol, and size scales the circle diameter according to the earthquake magnitude. To color the circles by hypocentral depth, a CPT is created with pygmt.makecpt() using the reversed *navia* CPT for depths ranging from 0 to 700 km. A transparency of 30% is applied to the CPT to prevent overplotting of underlying events. The hypocentral depths are supplied to the fill parameter to apply this color mapping, and each circle is outlined with a gray pen. A colorbar corresponding to the CPT is added using Figure.colorbar() with customized labels.

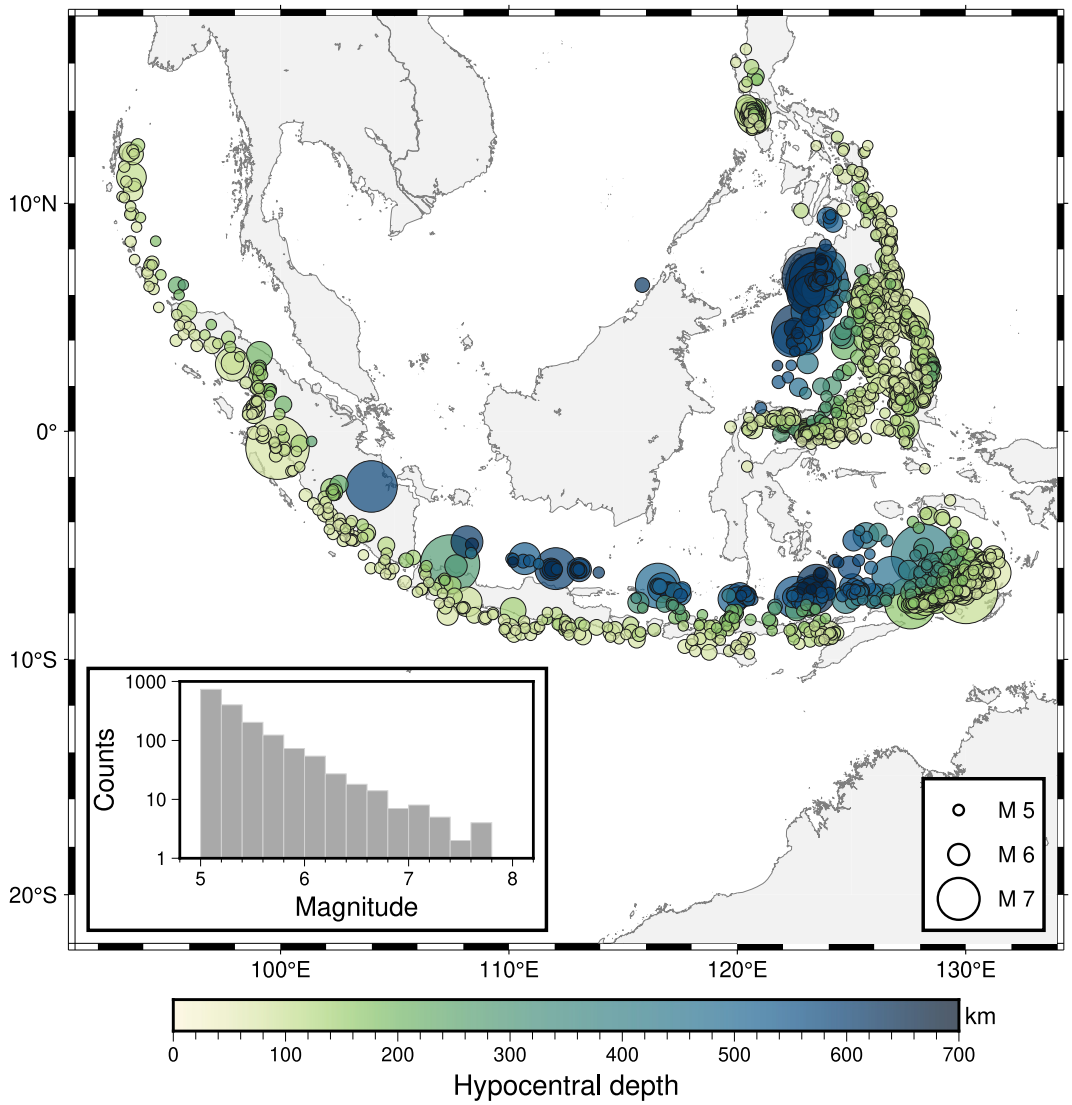

Figure 4. Seismicity along the Andaman-Sumatra-Java subduction zone from 2000 to 2025 with hypocentral depth ≥70 km and magnitude $M \geq 5$. Epicenters are shown as circles, with marker size exponentially scaled to magnitude and color indicating the hypocentral depth. The inset in the bottom-left corner shows the earthquake magnitude distribution as a Cartesian histogram with a logarithmic y-axis.

A legend illustrating the size-magnitude relationship is added using the Figure.legend() method at the bottom-right corner, with its placement controlled by the Position class. The legend is enclosed in a decorative box specified using the Box class, with a white fill and a black outline. The legend specification is created using Python's io.StringIO object, listing legend entries for magnitudes 5, 6, and 7. Each legend entry begins with S to indicate a symbol, followed by the offset from the left margin, the symbol type (c for circle), the symbol size in centimeters (scaled by magnitude), the fill color (a dash means no fill), and the outline pen, then the optional text label with its offset from the left margin. The line_spacing parameter adjusts the spacing between legend entries to prevent overlap. For simpler use cases, such as multiple lines or symbols with different styles, legends can be generated automatically (see the example in Section 3.5), while more complex and fully customized legends are also supported.

Finally, a Cartesian histogram showing the magnitude distribution is added. The Figure.inset() method defines an inset region in the bottom-left corner of the plot, restricting subsequent plotting to that area. The inset region is set with a width of 7 cm and a height of 4 cm with a box drawn around it. Additional margins are reserved around the inset to provide sufficient space for the histogram's axis annotations and labels. Within this inset, the histogram is

plotted using the Figure.histogram() method with a bin size of 0.2. The histogram bars are filled in dark gray and outlined in light gray. The region parameter is set to `[4.8, 8.2, 0, 0]`, where specifying both y-limits as zero allows GMT to automatically determine appropriate y-axis limits based on the histogram results. The projection is

```
import io

import pandas as pd
import pygmt
import requests
from pygmt.params import Axis, Box, Frame, Position

params = {
    "format": "csv",
    "starttime": "2000-01-01",
    "endtime": "2025-12-31",
    "mindepth": 70,
    "minmagnitude": 5,
}
r = requests.get("https://earthquake.usgs.gov/fdsnws/event/1/query", params=params)
df_eqs = pd.read_csv(io.StringIO(r.text))
df_eqs = df_eqs[
    df_eqs["longitude"].between(91, 134) & df_eqs["latitude"].between(-22, 18)
].sort_values(by="mag", ascending=False)

fig = pygmt.Figure()
fig.basemap(region=[91, 134, -22, 18], projection="M15c", frame=True)
fig.coast(land="gray95", shorelines="gray50")

# Plot epicenters with color (hypocentral depth) or size (magnitude)
pygmt.makecpt(cmap="SCM/navia", series=[0, 700], reverse=True, transparency=30)
fig.plot(
    x=df_eqs.longitude,
    y=df_eqs.latitude,
    style="c",
    size=0.005 * 2**df_eqs.mag,
    fill=df_eqs.depth,
    cmap=True,
    pen="gray10",
)
fig.colorbar(annot=True, tick=True, label="Hypocentral depth", unit="km")
# Add legend for size-coding
legend = io.StringIO(
    "\n".join(f"S 0.4 c {0.005 * 2**mag:.2f} - 1p 1 M {mag}" for mag in [5, 6, 7])
)
fig.legend(
    spec=legend,
    line_spacing=2,
    position=Position("BR", offset=0.2),
    box=Box(fill="white", pen="black"),
)

# Add histogram for magnitude distribution
with fig.inset(
    position=Position("BL", offset=0.2),
    box=True,
    width=7, height=4,
    clearance=(1.4, 0.2, 1.1, 0.2),
):
    fig.histogram(
        region=[4.8, 8.2, 0, 0],
        projection="X?/?l",
        frame=Frame(
            axes="WSrt",
            xaxis=Axis(annot=1, tick=0.2, label="Magnitude"),
            yaxis=Axis(annot=True, tick=True, label="Counts"),
        ),
        data=df_eqs.mag,
        series=0.2,
        fill="darkgray",
        pen="lightgray",
    )

fig.show()
```

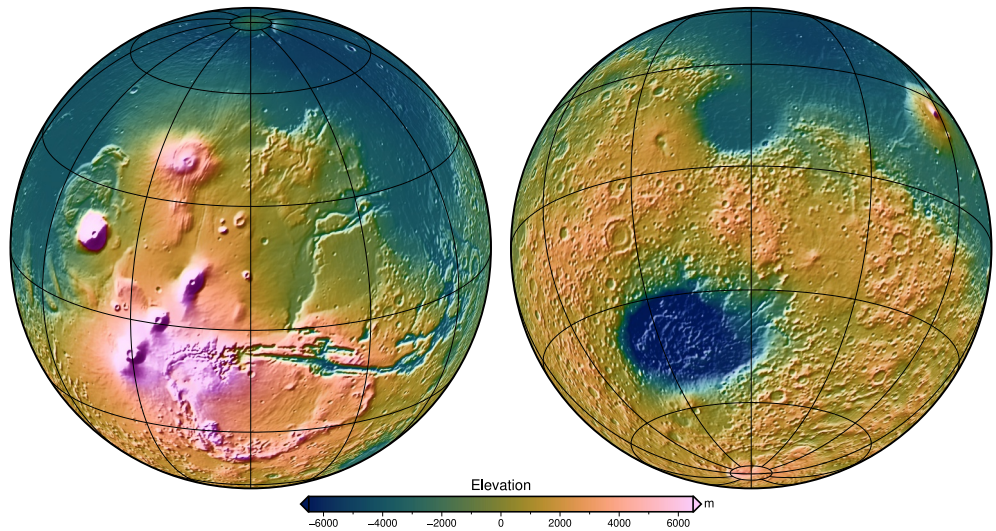

Figure 5. Hemispherical views of long-wavelength Mars topography derived from the 10 arc-min MOLA data. For details on the data processing, see the text.

set to "`X?/?l`," which selects a Cartesian coordinate system with a logarithmic y-axis. The two question marks instruct GMT to determine the plot width and height automatically from the inset dimensions and margins. By default, a count-based histogram is plotted although other histogram types (e.g., frequency) are also supported.

3.3. Mars Relief

Our third example presents a global map of Mars topography from the Mars Orbiter Laser Altimeter (MOLA; D. E. Smith et al., 2001). We begin by loading the 10 arc-min MOLA relief data set using pygmt.datasets.load_-mars_relief(), which retrieves the data set from the GMT data server. To emphasize long-wavelength topography associated with large-scale planetary features, we apply a Gaussian low-pass filter with a width of 50 km using the GMT accessor method *grid*.gmt.filter(). We then compute illumination gradients using *grid*.gmt.gradient() with an azimuth of −45° and normalize them using a cumulative Cauchy distribution. Because these grid operations must use the correct Martian reference ellipsoid, we set GMT's configuration parameter PROJ_ELLIPSOID to "`mars`" and perform all grid processing within a pygmt.config() context.

Figure 5 shows two hemispherical views of the long-wavelength MOLA topography. Both panels are plotted using Figure.grdimage() with the *batlow* CPT and shaded using the gradient grid computed above. Each map is 12 cm wide and uses an orthographic projection (projection code "G"), but with maps centered at (90°W, 20°N) and (90°E, 20°S), respectively. The right map is positioned by shifting the plotting origin, using Figure.shift_origin(), in the x-direction by "`w+0.5c`," where *w* refers to the width of the previously plotted object (the left map in this case). A colorbar corresponding to the CPT is added at the bottom center of the figure using Figure.colorbar(), with background and foreground triangles at both ends indicating values outside the range of the CPT.

This example illustrates how PyGMT combines geospatial processing and visualization within a single workflow. The filtering and gradient calculations are performed directly on a Martian topographic grid using the appropriate planetary reference ellipsoid, and the processed data set is then visualized, all within a unified PyGMT workflow. Similar workflows can also be implemented in different ways using combinations of existing Python libraries such as rioxarray, SciPy, and Cartopy, but typically require multiple processing steps across different libraries. For example, one possible workflow would involve reading the Martian topographic grid from a netCDF or GeoTIFF file, defining an appropriate Mars coordinate reference system, and reprojecting the grid into a suitable Cartesian coordinate system. A Gaussian filter and illumination gradients would then be computed on the projected grid, after which the processed grid could be reprojected if needed and finally visualized. PyGMT provides an integrated interface for these operations and enables the same workflow to be implemented with substantially less code.

```python
import pygmt
from pygmt.params import Axis, Position

mars = pygmt.datasets.load_mars_relief(resolution="10m")
with pygmt.config(PROJ_ELLIPSOID="mars"):
    mars_filter = mars.gmt.filter(
        filter="gaussian", width=50, distance="geo_spherical"
    )
    mars_grad = mars_filter.gmt.gradient(azimuth=-45, normalize="cauchy")

fig = pygmt.Figure()
pygmt.makecpt(cmap="SCM/batlow", series=[-6500, 6500])
fig.grdimage(
    mars_filter, shading=mars_grad, projection="G-90/20/12c", frame=Axis(grid=30)
)
fig.shift_origin(xshift="w+0.5c")
fig.grdimage(
    mars_filter, shading=mars_grad, projection="G90/-20/12c", frame=Axis(grid=30)
)
fig.colorbar(
    annot=2000, tick=1000, label="Elevation", unit="m",
    position=Position("BC", cstype="outside", offset=(-6.25, 0.2)),
    bg_triangle=True, fg_triangle=True,
    move_text="label",
)
fig.show()
```

3.4. States, Major Rivers, and Cities of the USA

This example demonstrates how PyGMT can be combined with GeoPandas to create a thematic map of the United States of America using multiple vector data sets with different geometry types (Figure 6). Three data sets available from Natural Earth (http://naturalearthdata.com), state boundaries ("*states*"), major rivers ("*rivers*"), and populated places ("*cities*"), illustrate the range of geometry types commonly found in GeoJSON and other vector formats. The *states* data set contains *Polygon* and *MultiPolygon* geometries, *rivers* contains *LineString* and *MultiLineString* geometries, and *cities* contains *Point* geometries.

Each file is read directly from its online source using geopandas.read_file(), which returns a geopandas.GeoDataFrame object suitable for attribute filtering, column manipulation, and spatial operations. The data sets are then filtered to include only U.S. features. State areas are computed in units of 1,000 km^2 by projecting the data set into an equal-area projection (EPSG:6933) and stored in the area column.

The map uses a Lambert conformal conic projection (projection code "L") for the continental U.S. State polygons are drawn with a 0.2-point-thick gray outline and colored according to the area using Figure.choropleth(). A CPT is generated using the *hawaii* CPT, with the corresponding color bar added to the lower-right corner of the map. Major rivers are overlaid as blue lines, while cities are plotted as dark orange squares. City names are annotated via Figure.text(), positioned above each city marker, with a semi-transparent white background and dark orange outline to ensure readability against the map.

Because Alaska and Hawaii are geographically distant from the continental U.S., they are plotted separately in the lower-left corner. For these two states, a Mercator projection is used, and the map size is set to 2 cm, so the scale differs from that of the continental states. The map extent for each state is determined directly from the data set using PyGMT's pygmt.info() function. Hawaii's horizontal extent is relatively large due to several small islands in its western part. To simplify the visualization, islands with an area smaller than 100 km^2 are filtered out. This is accomplished in GeoPandas by exploding *MultiPolygon* geometries into individual polygons with a *Polygon* geometry, removing polygons smaller than 100 km^2, and then dissolving the remaining polygons back into a

MultiPolygon geometry. The two states are colored using the same CPT as the continental states, enabling direct comparison of state areas across all U.S. regions.

```
import geopandas
import pygmt
from pygmt.params import Position

provider = "https://naciscdn.org/naturalearth"
states = geopandas.read_file(
    f"{provider}/50m/cultural/ne_50m_admin_1_states_provinces.zip"
)
rivers = geopandas.read_file(
    f"{provider}/50m/physical/ne_50m_rivers_lake_centerlines.zip"
)
cities = geopandas.read_file(
    f"{provider}/110m/cultural/ne_110m_populated_places_simple.zip"
)

states = states[states["admin"] == "United States of America"]
rivers = rivers[rivers.intersects(states.union_all())]
cities = cities[cities["adm0name"] == "United States of America"]

states["area"] = states.to_crs(epsg=6933).area / 10**9

fig = pygmt.Figure()
fig.basemap(
    projection="L-96/35/33/41/12c", region=[-126, -66, 25, 49], frame="none"
)
pygmt.makecpt(cmap="SCM/hawaii", series=[0, states["area"].max()], reverse=True)
fig.choropleth(states, pen="0.2p,gray50", column="area")
fig.colorbar(
    annot=True, tick=True, label="Area (1000 km@+2@+)",
    position=Position("BR", offset=(1.9, 0.2)),
    length=3,
    width=0.15,
    move_text="label",
)
fig.plot(data=rivers, pen="0.5p,dodgerblue4")
fig.plot(data=cities, style="s0.17c", fill="darkorange", pen="0.5p")
fig.text(
    x=cities.geometry.x,
    y=cities.geometry.y,
    text=cities["name"],
    offset="0.35c/0.2c",
    justify="BC",
    font="4.5p,Helvetica-Bold",
    fill="white@30",
    pen="0.2p,darkorange",
    clearance="0.05c+tO",
)

# Add the states Alaska and Hawaii separately in the lower left corner
for name, xshift in zip(["Alaska", "Hawaii"], ["1.2c", "2.8c"], strict=False):
    substate = states[(states["name"] == name)]
    substate = substate.explode()
    substate = substate[substate.to_crs(epsg=6933).area > 1.0e8].dissolve()
    region = pygmt.info(substate, spacing=1)
    with fig.shift_origin(xshift=xshift):
        fig.choropleth(
            substate,
            region=region,
            projection="M2c",
            pen="0.2p,gray50",
            column="area",
        )

fig.show()
```

3.5. Star History of GMT, GMT/MEX, GMT.jl, and PyGMT

Our final example illustrates the GitHub star history of four projects: the core GMT project and its three main interfaces, GMT/MEX (Wessel & Luis, 2017), GMT.jl (Luis & Wessel, 2018), and PyGMT, since 2016. The number of stars provides a rough indication of a project's visibility and interest among GitHub users. The data were retrieved via the GitHub API and stored in separate CSV files, each containing two columns: the timestamp when a user starred the project and the user's login name (e.g., "2017-01-02T03:04:05Z, anonymous").

```
import datetime

import pandas as pd
import pygmt
from pygmt.params import Axis, Box, Frame, Position

fig = pygmt.Figure()
fig.basemap(
    projection="X12c/6c",
    region=[datetime.date(2016, 1, 1), datetime.datetime.now(), -50, 1000],
    frame=Frame(xaxis=True, yaxis=Axis(annot=200, tick=50, label="GitHub stars")),
)

for wrapper, file, color, symbol in zip(
    ["GMT", "PyGMT", "GMT.jl", "GMT/MEX"],
    ["gmt", "pygmt", "gmt.jl", "gmtmex"],
    ["238/86/52", "48/105/152", "149/88/178", "230/51/51"],
    ["C", "A", "T", "I"],
    strict=False,
):
    df = pd.read_csv(
        f"star_history_github_{file}.csv",
        parse_dates=["timestamp"],
        index_col="timestamp",
    )
    df = df.resample("6ME").count().cumsum().rename(columns={"user": "stars"})
    fig.plot(x=df.index, y=df["stars"], pen=color)
    fig.plot(
        x=df.index, y=df["stars"], style=f"{symbol}0.2c", fill=color, label=wrapper
    )
fig.legend(
    position=Position("TL", offset=0.1),
    width=2.3,
    box=Box(fill="gray95", pen="0.5p,gray50", radius="3p"),
)
fig.show()
```

In this example, we begin by setting up a 12 cm × 6 cm base map using Figure.basemap(). The x-axis spans from January 2016 to the current date, using Python's standard datetime library, while the y-axis is set to range from −50 to 1,000. We then load the star-history data for a single project from the CSV file into a pandas.DataFrame object. The timestamp column is parsed as a datetime dtype and set as the index, enabling pandas' time-series operations. The data are resampled into 6-month intervals, and the number of stars in each interval is counted and cumulatively summed to produce the project's star-history data, stored in the stars column. This time-series data is then visualized using Figure.plot(), with the timestamp (the pandas.DataFrame index) and cumulative star counts (the stars column) passed to the x and y parameters, respectively.

A for-loop iterates over the four data sets plotting each with different colors (specified in *R/G/B* format), symbol markers, and labels. Each data set is plotted twice: first as a line connecting the points and then as scatter markers

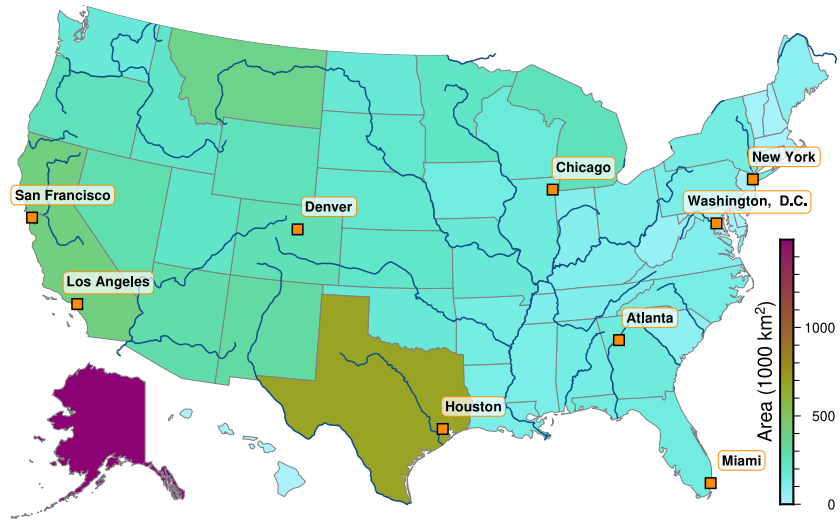

Figure 6. States, major rivers, and cities in the United States of America. GeoPandas layers with *Polygon*/*MultiPolygon* (state boundaries), *LineString*/*MultiLineString* (rivers), and *Point* (cities) geometries from the Natural Earth data set are visualized. State polygons are colored by area (in units of 1,000 km^2), while rivers and cities are shown as blue lines and black-outlined dark orange-filled squares, respectively. Alaska and Hawaii are displayed separately in the lower left corner using Mercator projections, whereas the main map uses a Lambert conformal conic projection.

(set via the style parameter), with the plotting order ensuring that the symbols appear above the line. The symbol marker codes C, A, T, and I represent circles, stars, triangles, and inverted triangles, respectively, with uppercase letters indicating that the symbols are scaled to have the same area as a circle with a 0.2 cm diameter. Finally, a legend is added to the top-left corner using the Figure.legend() method, with the legend entries controlled by the label parameter in Figure.plot(). A decorative box is added behind the legend with a 0.5-point-thick, *gray50* outline and rounded corners filled in *gray95*.

The resulting figure shows a steady increase in the number of stars for all four projects over time, reflecting their growing visibility and popularity on GitHub and the increasing user interest (Figure 7). This example highlights not only PyGMT's capability to handle and visualize time series data efficiently but also its usefulness for visualizing trends and comparisons across multiple data sets.

4. Project Governance and Community

PyGMT is developed and maintained as an open-source project with a strong emphasis on reliability, usability, and sustainability. This section outlines how the project is made available, the policies guiding its development and compatibility, and the efforts to maintain an open and welcoming community.

4.1. Project Maintenance

PyGMT is an open-source project released under the BSD 3-Clause license, a permissive license that allows users to freely use, modify, and redistribute the software with minimal restrictions, making it suitable for both academic and commercial use. The source code is publicly hosted on GitHub (https://github.com/GenericMappingTools/pygmt) and managed by the GMT organization.

To ensure software reliability and maintain high code quality, the PyGMT project uses the Git version control system for its development workflow, which is a best practice in modern software development. Contributors propose changes through pull requests, which are reviewed by other developers and contributors. Reviewers may suggest improvements or request revisions, and contributors update the code accordingly until the changes meet project standards. Once approved by one or more maintainers, depending on the complexity and impact of the changes, the pull request is merged into the main branch. The main branch reflects the current development version, which may be unstable but serves as the basis for the next release. This workflow ensures that all contributions are thoroughly reviewed, tested, and discussed before being integrated into the codebase.

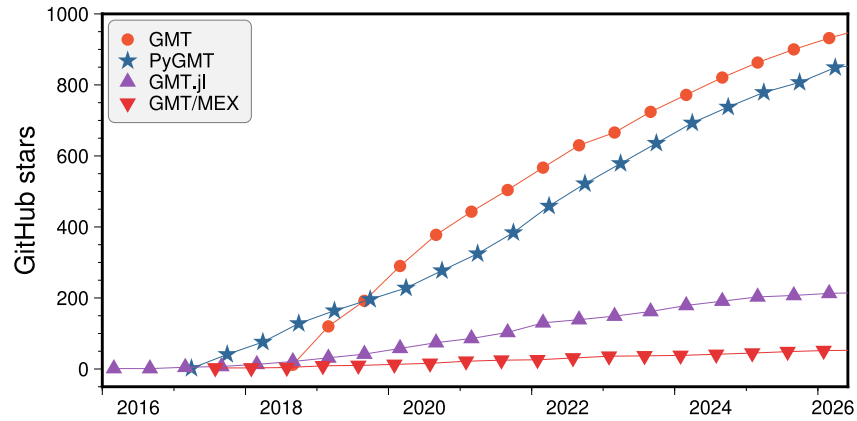

Figure 7. GitHub star history of the GMT (red circles), GMT/MEX (orange inverted triangles), GMT.jl (purple triangles), and PyGMT (blue stars) projects since 2016. GMT was originally developed outside GitHub and was migrated to GitHub in 2016. Therefore, its GitHub star history does not reflect the full history of the project.

All changes in PyGMT, including those in pull requests and the main branch, undergo automatic continuous integration (CI) workflows. The CI workflows run style checks, static type checks, benchmarks, and the full test suite across Windows, macOS, and Linux. Style and type checks are performed using automated linting and static type-checking tools, ensuring consistent coding style and maintaining high code quality throughout the codebase. Benchmarks help monitor performance changes over time and ensure that changes do not introduce efficiency regressions. The full test suite covers the minimum supported, latest, and development versions of Python, GMT, and required dependencies, including NumPy, pandas, and xarray. This ensures that changes do not break existing functionality and that plotting outputs remain consistent with baseline images. Code coverage is also tracked to monitor test coverage across the codebase, which is currently around 99%.

PyGMT also strives to provide high-quality, reproducible documentation to help users learn and use the software effectively. The documentation follows the DIVIO documentation system (https://docs.divio.com/documentation-system/) and is structured into four types. *Tutorials* are learning-oriented documents that guide users step by step through complete examples, demonstrating the core concepts and typical workflows in PyGMT. *How-to guides* are problem-oriented documents that show how to accomplish specific user tasks and provide working solutions. *Explanations* are understanding-oriented documents that describe technical details and underlying mechanisms of PyGMT. *Reference documentation* is information-oriented and provides detailed descriptions of the purpose, parameters, and arguments of each function, method, and class, accompanied by working examples. All examples in the documentation are fully reproducible, with figures generated automatically when the documentation is built and deployed through CI workflows. This ensures that figures displayed in the documentation are always consistent with the current codebase. Each example can also be copied or downloaded as a standalone Jupyter notebook or Python script, facilitating straightforward reproducibility. Documentation for the development version and all release versions is hosted online at https://www.pygmt.org, with ZIP archives and PDFs available for offline use. Users can easily switch between versions using the documentation version selector.

These practices ensure that PyGMT remains reliable, maintainable, and well-documented, supporting both developers and users while fostering a robust and sustainable open-source project.

4.2. Availability and Citation

PyGMT is a pure Python package with no compiled code, making it lightweight and easy to install on operating systems such as Windows, macOS, and Linux. It is available for installation through the Python Package Index (PyPI) (https://pypi.org/project/pygmt/) as well as via the conda or mamba package managers from the conda-forge (Conda-Forge Community, 2015) channel (https://anaconda.org/conda-forge/pygmt). Some dependencies, including GMT itself and tools such as Ghostscript, cannot be installed via PyPI and must be installed separately through the conda-forge channel, other platform-specific package managers, or by building them from source. As a result, installing PyGMT may require more dependencies than some pure-Python visualization and geospatial processing libraries. For most users, the easiest and recommended installation method is a single command:

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

This command creates an isolated virtual environment and installs all required dependencies. Full installation instructions, including how to install the latest stable or development version and guidance on common installation issues, are available at https://www.pygmt.org/latest/install.html.

To further lower the barrier to entry, PyGMT can also be used directly in web browsers without local installation through platforms such as Binder (https://mybinder.org) and Google Colab (https://colab.research.google.com). These platforms provide cloud-based environments where users can explore PyGMT, making it well-suited for teaching, workshops, and reproducible research.

PyGMT follows an organized release process and uses semantic versioning (https://semver.org/). Changes merged in the main branch are thoroughly tested through CI workflows. Releases are currently made approximately every 3 months although the schedule may evolve in the future. The release process is largely automated and requires minimal manual effort. It includes tagging a release on GitHub, updating the documentation, uploading packages to PyPI and conda-forge, archiving the source code on Zenodo, where a digital object identifier (DOI) is assigned for each release, and making announcements. Semantic versioning, in the format of *MAJOR.MINOR.PATCH* (e.g., 0.19.0), clearly indicates the nature of changes: *MAJOR* for incompatible API changes, *MINOR* for new features in a backward-compatible manner, and *PATCH* for bug fixes and minor improvements. For each release, important changes are summarized in the changelog page, which is organized into categories such as new features, bug fixes, enhancements, deprecations, and maintenance, allowing users and developers to quickly identify the changes introduced in each release.

To ensure proper attribution and reproducibility, users are encouraged to cite the main PyGMT publication (this paper) and the GMT paper (Wessel et al., 2019), along with the DOIs corresponding to the specific PyGMT and GMT versions used in their work. This practice helps recognize the contributions of both the PyGMT and GMT development teams and supports reproducibility in research.

4.3. Compatibility Policy

To maintain compatibility with the broader scientific Python ecosystem, PyGMT has adopted the SPEC 0 policy developed by the Scientific Python Ecosystem Coordination project. Following this policy, PyGMT will drop support for Python versions 3 years after their initial release and for core package dependencies, such as NumPy, pandas, and xarray, 2 years after their initial release. In addition, the PyGMT development team has decided to drop support for GMT versions 3 years after their initial release, while ensuring that at least the two latest minor GMT versions remain supported. Limiting support to two GMT versions helps reduce the maintenance burden while still providing users with a reasonable range of supported environments. For example, at the time of writing, PyGMT v0.19.0 supports Python 3.12–3.14, GMT 6.5–6.6, NumPy $\geq$ 2.1, pandas $\geq$ 2.3, xarray $\geq$ 2024.7. For optional dependencies that fall outside SPEC 0, PyGMT generally maintains support for at least 1 year after their initial release. The SPEC 0 policy is enforced on a best-effort basis, and the PyGMT team may decide to drop support for core or optional dependencies earlier than recommended when required for compatibility reasons. While backward compatibility is typically adhered to, users are encouraged to use the most recent versions of Python, GMT, and optional dependencies in order to benefit from the latest features and improvements. A dedicated page (https://www.pygmt.org/latest/minversions.html) is available listing the supported versions for all PyGMT releases.

Since PyGMT is still under active development, some changes may break backward compatibility. To minimize the inconvenience, deprecated functionality is retained for at least two minor releases, and a PythonFutureWarning is issued to notify users about upcoming incompatible changes and recommend alternative approaches. All deprecations are documented in the changelog, allowing users to track upcoming incompatible changes and plan updates accordingly.

4.4. Open and Welcoming Community

The PyGMT community aims to maintain an open and welcoming environment where contributors from different backgrounds, experience levels, and geographic regions are encouraged to participate. Contributions are defined broadly and include writing code, reviewing pull requests, improving documentation, creating tutorials and examples, reporting bugs, organizing workshops, and other community activities. Contributions are recognized in

several ways, including inclusion in the list of PyGMT developers, acknowledgment in release changelogs, and authorship of Zenodo records and scientific publications when contributions meet the relevant criteria. Contributors who participate frequently may choose to become maintainers although there is no expectation of long-term commitment. Clear authorship, contributing, maintenance, and code-of-conduct guidelines are publicly available to help new contributors get started and foster a welcoming and collaborative community environment. At the time of writing, PyGMT has 51 contributors from a variety of backgrounds, genders, and countries.

PyGMT development primarily takes place on GitHub, where issues and pull requests are the main ways for collaboration. GitHub Issues use standard issue templates to help users provide the necessary information when reporting bugs, proposing improvements, or requesting new features. Users are encouraged to comment on issues if they are interested in working on them. Issues labeled "good first issue" are tasks with a low knowledge barrier and are particularly suitable for new contributors. Pull requests are the main method for contributing code, documentation, and examples. Pull requests are reviewed collaboratively, with feedback intended to help contributors improve their submissions and become familiar with project practices.

The GMT Discourse forum (https://forum.generic-mapping-tools.org) provides an additional platform for community discussions, including a dedicated section for PyGMT-related questions and discussions. Users can ask questions, share experiences, and report problems, while developers and other participants can provide guidance and support. This open communication channel facilitates collaboration, encourages contributions from the community, and ensures that user feedback is incorporated into the ongoing development.

Beyond software development, the PyGMT community has been actively engaging in outreach activities to promote the project and expand its user base. These activities take various forms, including community coding sprints (e.g., the SciPy 2022 (Jones et al., 2022)), educational training sessions (e.g., Remote Online Sessions for Emerging Seismologists (ROSES) 2020 (Dannemann Dugick et al., 2021)), conference workshops (e.g., EGU22 PyGMT short course (Leong et al., 2022), AGU24 pre-conference workshop (Leong et al., 2024)), and conference talks (Fröhlich et al., 2024). These activities have played an important role in introducing PyGMT to both GMT and non-GMT users, raising awareness of the project, and fostering community engagement.

5. Current Status and Future Development

At the time of writing, PyGMT wraps about 20 plotting modules and 30 non-plotting modules among the 98 GMT core modules, along with four modules among the 58 GMT supplementary modules. For modules that have not yet been wrapped, experienced users can quickly create a basic wrapper by following the existing PyGMT source code, though making the code Pythonic may require additional effort and discussions. Even among the wrapped modules, not all functionality is fully implemented. As a temporary workaround, users can still pass the GMT-style argument strings to the corresponding single-letter parameters directly. For example, if the Figure.legend () method does not provide the box parameter and the Box class is not implemented, users can still pass `F = "+p1p+glightblue"` to the method. These single-letter parameters are currently supported as a temporary workaround, but their use is discouraged in favor of the more Pythonic parameter names when available. Whether they will be phased out in the future will depend on user feedback. The PyGMT team will continue wrapping more modules and adding new functionality to expand the package's capabilities. The prioritization of module wrapping is primarily driven by community needs, feature requests, contributor interest, and the importance of the functionality to common PyGMT workflows. Because PyGMT is a wrapper around the GMT C API, most new functionality in PyGMT builds upon capabilities already implemented in GMT. In some cases, higher-level functionality can be implemented directly in PyGMT without requiring changes to GMT itself (e.g., the Figure.tilemap() method demonstrated in Figure 3). However, when new capabilities require functionality that is not available in GMT, they are typically first implemented in GMT and then exposed through PyGMT.

GMT's plot module is one of the most complex modules and can plot lines and polygons as well as a variety of symbols. Currently, PyGMT provides four high-level methods to simplify the usage of this module, including Figure.hlines(), Figure.vlines(), Figure.choropleth(), and Figure.fill_between(). Potential future additions include methods such as Figure.scatter(), Figure.errorbars(), Figure.stem(), and more, as well as extensions to 3-D plotting. These additions will offer simpler and more flexible ways for users to create both 2-D and 3-D plots.

Parallel and out-of-core computing frameworks such as Dask (Rocklin, 2015) have become important components of the scientific Python ecosystem for efficiently processing large data sets. PyGMT can accept Dask-backed xarray and pandas objects through its integration with these libraries, but PyGMT is not currently Dask-aware. Consequently, Dask-backed data are loaded into memory before being passed to GMT modules. Nevertheless, some computationally intensive GMT modules, such as grdimage and grdfilter, already support multi-threaded execution through OpenMP (Dagum & Menon, 1998). Improving integration with Dask and similar frameworks remains a possible direction for future development.

The PyGMT team is also exploring the possibility of running PyGMT locally in a web browser using Pyodide (https://pyodide.org), which would be especially useful for workshops and for beginners who want to quickly try PyGMT without installing anything. However, there are still technical challenges to overcome, particularly the need to port the Ghostscript package into Pyodide, as it is a required dependency for GMT.

Future development and feature requests are tracked through GitHub issues. The PyGMT team welcomes contributions, feedback, and ideas from the community to help shape the future evolution of the project.

6. Summary

PyGMT leverages GMT's API to provide a high-level interface that integrates seamlessly with the scientific Python ecosystem, including libraries such as NumPy, pandas, and xarray. By enabling efficient in-memory data exchange with GMT and offering an intuitive and Pythonic syntax, PyGMT lowers the barrier to entry for new users while enabling experienced programmers to build complex, reproducible geospatial workflows. Combining GMT's geospatial processing, analysis, and visualization capabilities with Python's flexible programming environment, PyGMT is a valuable tool for modern geospatial research. As an open-source and community-driven project, PyGMT continues to evolve through active development, user feedback, and community contributions, supporting its long-term reliability, usability, and sustainability.

Conflict of Interest

The authors declare no conflicts of interest relevant to this study.

Availability Statement

The PyGMT source code is archived on Zenodo (Tian, Leong, et al., 2026) with the development repository hosted on GitHub (https://github.com/GenericMappingTools/pygmt). The documentation is available at https://www.pygmt.org. All figures in this paper were generated using PyGMT v0.19.0 (Tian, Leong, et al., 2026) and GMT version 6.6.0 (Wessel et al., 2019, 2025). The Jupyter notebooks and accompanying data files used to generate the figures are available at https://github.com/GenericMappingTools/pygmt-paper-figures and archived on Zenodo (Tian, Fröhlich, et al., 2026).

Acknowledgments
We thank Fabio Crameri and an anonymous reviewer for their constructive comments and suggestions, which helped improve this manuscript. We acknowledge all contributors to PyGMT and thank the GMT developers for developing and maintaining the core GMT software. In particular, we honor the memory of Pål Wessel (1959–2024), the founder of GMT, whose vision and contributions laid the foundation for this project. We also acknowledge Sfurti (https://github.com/sfrooti) for contributing the initial version of the PyGMT logo (https://github.com/sfrooti/PyGMT-Logo). PyGMT relies on several freely available platforms and tools, which collectively contribute to the project's long-term sustainability and maintainability. At the time of writing, these include GitHub (https://github.com/) for source code hosting and collaboration; GitHub Actions (https://github.com/features/actions) for continuous integration; CodeCov (https://about.codecov.io/) for measuring code coverage; CodSpeed (https://codspeed.io/) for performance benchmarking; DagsHub (https://dagshub.com/) for storing baseline images, with data versioning managed using DVC (Data Version Control); ReadTheDocs (https://about.readthedocs.com/) for documentation previews in pull requests; Ruff (https://astral.sh/ruff) for code formatting and linting; mypy (https://mypy-lang.org) for static type checking; codespell (https://github.com/codespell-project/codespell) and typos (https://github.com/crate-ci/typos) for documentation and source code spell checking; PyPI (https://pypi.org) and conda-forge (https://conda-forge.org) for package distribution; and Zenodo (https://zenodo.org) for long-term archiving and citation through digital object identifiers (DOIs). The development of PyGMT has been supported by NSF Grants OCE-1558403 and EAR-1948602. D.T. is supported by the National Natural Science Foundation of China (No. 42274122) and the "CUG Scholar" Scientific Research Funds at China University of Geosciences (Wuhan) (Project No. 2022012). J.M.F.L. is supported by FCT, I.P./MCTES through national funds (PIDDAC): LA/P/0068/2020, UID/50019/2025, UID/PRR/50019/2025, and UID/PRR2/50019/2025.

References

Adobe Systems Inc. (1990). *PostScript language reference manual*. Addison-Wesley Longman Publishing Co., Inc.

Baker, M. (2016). 1,500 scientists lift the lid on reproducibility. *Nature*, *533*(7604), 452–454. https://doi.org/10.1038/533452a

Chamberlain, C. J., Townend, J., & Gerstenberger, M. C. (2020). RT-EQcorrscan: Near-real-time matched-filtering for rapid development of dense earthquake catalogs. *Seismological Research Letters*, *91*(6), 3574–3584. https://doi.org/10.1785/0220200171

Conda-Forge Community. (2015). The conda-forge Project: Community-based Software Distribution Built on the conda Package Format and Ecosystem. *Zenodo*. https://doi.org/10.5281/zenodo.4774217

Crameri, F. (2018). Geodynamic diagnostics, scientific visualisation and StagLab 3.0. *Geoscientific Model Development*, *11*(6), 2541–2562. https://doi.org/10.5194/gmd-11-2541-2018

Crameri, F. (2023). Scientific colour maps (Version 8.0.1). *Zenodo*. https://doi.org/10.5281/zenodo.1243862

Dagum, L., & Menon, R. (1998). OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, *5*(1), 46–55. https://doi.org/10.1109/99.660313

Dannemann Dugick, F. K., van der Lee, S., Prieto, G. A., Dybing, S. N., Toney, L., & Cole, H. M. (2021). ROSES: Remote online sessions for emerging seismologists. *Seismological Research Letters*, *92*(4), 2657–2667. https://doi.org/10.1785/0220200421

Fröhlich, Y., Tian, D., Leong, W. J., Jones, M., & Grund, M. (2024). PyGMT—Accessing and integrating GMT with Python and the scientific Python ecosystem. In *Presented at the Annual Meeting of the American Geophysical Union*. https://doi.org/10.6084/m9.figshare.28049495.v1

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., et al. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Hoyer, S., & Hamman, J. (2017). xarray: N-D labeled arrays and datasets in Python. *Journal of Open Research Software*, *5*(1), 10. https://doi.org/10.5334/jors.148

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, *9*(3), 90–95. https://doi.org/10.1109/MCSE.2007.55

Jones, M., Uieda, L., & Leong, W. J. (2022). Geospatial analysis and visualization with PyGMT. In *Presented at the SciPy 2022*. https://doi.org/10.6084/m9.figshare.20483793.v1

Leong, W. J., Fröhlich, Y., Tong, J.-H., Esteban, F., Jones, M., & Belem, A. L. (2024). Mastering geospatial visualizations with GMT/PyGMT. In *Presented at the Annual Meeting of the American Geophysical Union*. https://doi.org/10.5281/zenodo.15809717

Leong, W. J., Uieda, L., Jones, M., & Belém, A. (2022). Crafting beautiful maps with PyGMT. In *Presented at the General Assembly of the European Geoscience Union*. https://doi.org/10.5281/zenodo.12631465

Luis, J. M. F., & Wessel, P. (2018). A Julia wrapper for the Generic Mapping Tools. In *Presented at the Annual Meeting of the American Geophysical Union*. Retrieved from https://ui.adsabs.harvard.edu/abs/2018AGUFMNS53A0563L

Mather, B. R., Müller, R. D., Zahirovic, S., Cannon, J., Chin, M., Ilano, L., et al. (2024). Deep time spatio-temporal data analysis using pyGPlates with PlateTectonicTools and GPlately. *Geoscience Data Journal*, *11*(1), 3–10. https://doi.org/10.1002/gdj3.185

McKinney, W. (2010). Data structures for statistical computing in Python. In *Presented at the SciPy 2010*. https://doi.org/10.25080/Majora-92bf1922-00a

Met Office. (2010). Cartopy: A cartographic Python library with a Matplotlib interface. Manual, Exeter, Devon. Retrieved from https://cartopy.readthedocs.io

Perez, F., & Granger, B. E. (2007). IPython: A system for interactive scientific computing. *Computing in Science & Engineering*, *9*(3), 21–29. https://doi.org/10.1109/MCSE.2007.53

Rocklin, M. (2015). Dask: Parallel computation with blocked algorithms and task scheduling. In *Presented at the SciPy 2015*. https://doi.org/10.25080/Majora-7b98e3ed-013

Smith, D. E., Zuber, M. T., Frey, H. V., Garvin, J. B., Head, J. W., Muhleman, D. O., et al. (2001). Mars Orbiter Laser Altimeter: Experiment summary after the first year of global mapping of Mars. *Journal of Geophysical Research*, *106*(E10), 23689–23722. https://doi.org/10.1029/2000JE001364

Smith, W. H. F., & Wessel, P. (1990). Gridding with continuous curvature splines in tension. *Geophysics*, *55*(3), 293–305. https://doi.org/10.1190/1.1442837

Snow, A. D., Braun, R., RichardScottOZ, Raspaud, M., Brochart, D., Kouzoubov, K., et al. (2026). corteva/rioxarray: 0.22.0 (Version 0.22.0). *Zenodo*. https://doi.org/10.5281/zenodo.18892731

Tankersley, M. D. (2024). PolarToolkit: Python tools for convenient, reproducible, and open polar science. *Journal of Open Source Software*, *9*(100), 6502. https://doi.org/10.21105/joss.06502

Tian, D., Fröhlich, Y., Grund, M., Schlitzer, W., & Leong, W. J. (2026). Reproducible materials for "PyGMT: Bridging Python and the Generic Mapping Tools for Geospatial Visualization and Analysis" [Dataset]. *Zenodo*. https://doi.org/10.5281/zenodo.19412361

Tian, D., Leong, W. J., Fröhlich, Y., Grund, M., Schlitzer, W., Jones, M., et al. (2026). PyGMT: A Python interface for the Generic Mapping Tools (Version 0.19.0). *Zenodo*. https://doi.org/10.5281/zenodo.19398871

Uieda, L. (2018). Verde: Processing and gridding spatial data using Green's functions. *Journal of Open Source Software*, *3*(30), 957. https://doi.org/10.21105/joss.00957

Uieda, L., & Wessel, P. (2017a). A modern Python interface for the Generic Mapping Tools. In *Presented at the Annual Meeting of the American Geophysical Union*. https://doi.org/10.6084/m9.figshare.5662411

Uieda, L., & Wessel, P. (2017b). Bringing the Generic Mapping Tools to Python. In *Presented at the SciPy 2017*. https://doi.org/10.6084/m9.figshare.7635833

Van den Bossche, J., Jordahl, K., Fleischmann, M., Richards, M., McBride, J., Jacob, W., et al. (2026). geopandas/geopandas: Version 1.1.3 (Version v1.1.3). *Zenodo*. https://doi.org/10.5281/zenodo.18934477

van Rossum, G., & de Boer, J. (1991). Interactively testing remote servers using the Python programming language. *CWI Quarterly*, *4*(4), 283–304.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., et al. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, *17*(3), 261–272. https://doi.org/10.1038/s41592-019-0686-2

Wessel, P. (2009). A general-purpose Green's function-based interpolator. *Computers & Geosciences*, *35*(6), 1247–1254. https://doi.org/10.1016/j.cageo.2008.08.012

Wessel, P., & Becker, J. M. (2008). Interpolation using a generalized Green's function for a spherical surface spline in tension. *Geophysical Journal International*, *174*(1), 21–28. https://doi.org/10.1111/j.1365-246X.2008.03829.x

Wessel, P., Esteban, F., & Delaviel-Anger, G. (2024). The Generic Mapping Tools and animations for the masses. *Geochemistry, Geophysics, Geosystems*, *25*(6), e2024GC011545. https://doi.org/10.1029/2024GC011545

Wessel, P., & Luis, J. F. (2017). The GMT/MATLAB toolbox. *Geochemistry, Geophysics, Geosystems*, *18*(2), 811–823. https://doi.org/10.1002/2016GC006723

Wessel, P., Luis, J. F., Uieda, L., Scharroo, R., Wobbe, F., Smith, W. H. F., et al. (2025). The Generic Mapping Tools version 6.6.0 (Version 6.6.0). *Zenodo*. https://doi.org/10.5281/zenodo.16448627

Wessel, P., Luis, J. F., Uieda, L., Scharroo, R., Wobbe, F., Smith, W. H. F., & Tian, D. (2019). The Generic Mapping Tools version 6. *Geochemistry, Geophysics, Geosystems*, *20*(11), 2019GC008515. https://doi.org/10.1029/2019GC008515

Wessel, P., & Smith, W. H. F. (1991). Free software helps map and display data. *EOS Transactions of the American Geophysical Union*, *72*(41), 441–446. https://doi.org/10.1029/90EO00319

Wessel, P., & Smith, W. H. F. (1995). New version of the Generic Mapping Tools. *EOS Transactions of the American Geophysical Union*, *76*(33), 329. https://doi.org/10.1029/95EO00198

Wessel, P., & Smith, W. H. F. (1996). A global, self-consistent, hierarchical, high-resolution shoreline database. *Journal of Geophysical Research*, *101*(B4), 8741–8743. https://doi.org/10.1029/96JB00104

Wessel, P., & Smith, W. H. F. (1998). New, improved version of Generic Mapping Tools released. *EOS Transactions of the American Geophysical Union*, *79*(47), 579. https://doi.org/10.1029/98EO00426

Wessel, P., Smith, W. H. F., Scharroo, R., Luis, J., & Wobbe, F. (2013). Generic Mapping Tools: Improved version released. *EOS Transactions of the American Geophysical Union*, *94*(45), 409–410. https://doi.org/10.1002/2013EO450001

Wieczorek, M. A., & Meschede, M. (2018). SHTools: Tools for working with spherical harmonics. *Geochemistry, Geophysics, Geosystems*, *19*(8), 2574–2592. https://doi.org/10.1029/2018GC007529

Wilkinson, M. D., Dumontier, M., Aalbersberg, I., Appleton, G., Axton, M., Baak, A., et al. (2016). The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data*, *3*(1), 160018. https://doi.org/10.1038/sdata.2016.18