MetPy

A Meteorological Python Library for Data Analysis and Visualization

Ryan M. May, Kevin H. Goebbert, Jonathan E. Thielen, John R. Leeman, M. Drew Camron, Zachary Bruick, Eric C. Bruning, Russell P. Manser, Sean C. Arms, and Patrick T. Marsh

ABSTRACT: MetPy is an open-source, Python-based package for meteorology, providing domain-specific functionality built extensively on top of the robust scientific Python software stack, which includes libraries like NumPy, SciPy, Matplotlib, and xarray. The goal of the project is to bring the weather analysis capabilities of GEMPAK (and similar software tools) into a modern computing paradigm. MetPy strives to employ best practices in its development, including software tests, continuous integration, and automated publishing of web-based documentation. As such, MetPy represents a sustainable, long-term project that fills a need for the meteorological community. MetPy's development is substantially driven by its user community, both through feedback on a variety of open, public forums like Stack Overflow, and through code contributions facilitated by the GitHub collaborative software development platform. MetPy has recently seen the release of version 1.0, with robust functionality for analyzing and visualizing meteorological datasets. While previous versions of MetPy have already seen extensive use, the 1.0 release represents a significant milestone in terms of completeness and a commitment to long-term support for the programming interfaces. This article provides an overview of MetPy's suite of capabilities, including its use of labeled arrays and physical unit information as its core data model, unit-aware calculations, cross sections, skew *T* and GEMPAK-like plotting, station model plots, and support for parsing a variety of meteorological data formats. The general road map for future planned development for MetPy is also discussed.

KEYWORDS: Atmosphere; Algorithms; Data processing/distribution; Software

https://doi.org/10.1175/BAMS-D-21-0125.1
Corresponding author: Ryan M. May, rmay@ucar.edu
In final form 6 July 2022

AFFILIATIONS: May and Camron—Unidata, University Corporation for Atmospheric Research, Boulder, Colorado; **Goebbert**—Valparaiso University, Valparaiso, Indiana; **Thielen**—Colorado State University, Fort Collins, Colorado; **Leeman***—Leeman Geophysical LLC, Siloam Springs, Arkansas; **Bruick***—McKinsey and Company, Denver, Colorado; **Bruning and Manser**—Texas Tech University, Lubbock, Texas; **Arms***—Longmont, Colorado; **Marsh**—NOAA/Storm Prediction Center, Norman, Oklahoma
***FORMER AFFILIATION: Leeman, Bruick, and Arms**—Unidata, University Corporation for Atmospheric Research, Boulder, Colorado

Meteorology and atmospheric science have long had a strong reliance on data visualization and especially map analysis. From the mid 1980s through the early 2000s, this meant widespread use of software tools like Read Interpolate Plot version 4 (RIP4; Stoelinga 2018), NCAR Command Language (NCL; NCAR 2019), General Meteorology Package (GEMPAK; Unidata 2019), and Grid Analysis and Display System (GrADS; George Mason University 2018). Tools like these achieved favor because they promoted a scripting workflow, where users make small data analysis programs that also display and save images of the results. In these "scripting" workflows, the tight loop between making changes to the program and seeing results promotes interrogation of the data. On the other hand, these tools were implemented in low-level languages such as C and Java, making them harder for scientists without significant software development experience to extend beyond their original scope. Furthermore, these packages seldom sought to leverage similarities across scientific disciplines and instead took a singular and problem-specific approach to workflow and data formats.

In parallel, general-purpose array-based scientific computing packages such as IDL and MATLAB also became widely utilized through the mid-2000s, though these packages lacked an open-source library of meteorology-specific calculations and visualization functionality. Around this time, the Python language grew in prominence in scientific programming contexts, becoming a viable replacement for general-purpose scientific computing and visualization, while also having the advantage of being a modern, full-featured programming language with wide use in system scripting and web server development. Python is now, as of December 2021, the most popular programming language according to the TIOBE Index (TIOBE 2021) for programming languages, and has been adopted within a variety of scientific disciplines—including meteorology and atmospheric science. Among the features that have led to this adoption are Python's generally readable syntax and its "batteries included" nature, wherein many useful tools are included in the language's own standard library. MetPy fills gaps in the Python ecosystem in the areas of map analysis and meteorological calculations, while aligning with ongoing, active development in the wider Python ecosystem. The MetPy package has already seen use in atmospheric science journal articles to create publication quality graphics (e.g., Wade and Parker 2021; Schueth et al. 2021; McDonald and Weiss 2021). MetPy joins other recent efforts to build Python tools for atmospheric science, and geoscience applications in general [e.g., Pytroll (Raspaud et al. 2018), SHARPpy (Blumberg et al. 2017)].

Software design philosophy

One of the goals of the MetPy project is to help the atmospheric science community modernize its software toolset by bringing the best features of GEMPAK and tools like it to the scientific

Python ecosystem. This allows MetPy to take advantage of scientific Python's extensive community-driven, cross-discipline development.

MetPy builds on a core of scientific Python libraries that have grown and reached maturity over the previous decade, including NumPy (numeric arrays for fast computation; Harris et al. 2020), SciPy (scientific algorithms; Virtanen et al. 2020), Matplotlib (publication-quality plotting; Hunter 2007), pandas (data structures and algorithms for tabular data; Pandas Development Team 2022; McKinney 2010), and xarray (multidimensional numeric arrays with named coordinate dimensions and metadata; Hoyer and Hamman 2017). These robust and widely used libraries have helped fuel the explosive growth of Python in the sciences. This software "stack" has continued to grow with more recent additions, such as Dask (Dask Development Team 2022) for distributed computing and Cartopy (Met Office 2021) for map-based plotting. Together, these libraries form a shared foundation that can be leveraged to build more domain-specific tools without the need to build core numerical and algorithmic functionality from the ground up.

Since MetPy is built on the rich interdisciplinary scientific computing ecosystem that has developed around Python, we have purposely tried to restrict the scope of development to meteorology-specific pieces not readily available elsewhere. As appropriate, we have contributed general-purpose functionality to MetPy's upstream dependencies, helping to benefit the broader ecosystem—an approach that pays dividends in an Earth system science framework (Leemans et al. 2009; Reid et al. 2010) that routinely crosses the disciplinary boundaries of the twentieth century.

Another foundational development principle includes building a collection of pieces that can be joined together to suit the problem at hand. Users may bring data from their existing Python workflows in one of the standard Python numerical data formats and hand those off to MetPy's calculation functions. These functions are intentionally small such that they are composable, i.e., easily combined to create more complex functions. For plotting, MetPy's utilities directly extend Matplotlib so that users can leverage their prior expertise. This approach also means that MetPy can serve as an introduction to skills with broader application across the Python ecosystem. This philosophy of creating small and composable functions has enabled smooth development of new functionality, such as the declarative plotting syntax described below.

At the core of the interoperability between these pieces is MetPy's data model, which is chiefly built upon the xarray package. Xarray is based on the successful netCDF data model, consisting of regular arrays of data with shared named dimensions and attached metadata, stored in a self-describing fashion. The netCDF data model has been enormously successful as a portable binary data format, with use across many disciplines (Unidata 2021). The xarray package improves on prior Python interfaces to the netCDF data model by automating access to dimensionally consistent coordinate information and metadata. In this way, xarray more fully realizes the promise of self-describing data by automating what the computer can automate, giving the data more human immediacy. The rapid adoption of xarray across the Python data science ecosystem points to the success of meteorology's netCDF data model in addressing a fundamental data science challenge. Another benefit of using xarray is its support for a variety of data formats beyond netCDF, including remote access protocols like OPeNDAP, pluggable custom storage backends such as the cloud-native Zarr (Zarr Developers 2021), and flexible array types.

By building on these broadly accepted xarray and numpy data models, MetPy functions can operate on any dataset that can be transformed into xarray data structures or NumPy arrays. This allows compatibility with file formats that can be read using other Python libraries in the scientific computing ecosystem. For instance, the domain specific file format GRIB can be read with cfgrib to obtain xarray data structures or pygrib to obtain numpy arrays. More general file formats like HDF5 can be read using h5py and the data can be subsequently converted to numpy arrays. The existence of these community-developed packages lessens, or even completely eliminates, the need to implement support for these formats directly within MetPy.

Along with xarray, MetPy leverages the Pint (Grecco et al. 2021) library to track and automatically reconcile unit information on data through various operations and calculations. We chose to make unit awareness a key part of the data model to simplify documentation and eliminate a common source of (silent) user mistakes. Automatically checking unit correctness can help users identify potential problems with their use of a calculation function, since the library can provide error messages if the dimensionality of provided data are not as expected. By explicitly including unit information as part of the data model, we also enable more automated conversions to take place, allowing users to readily switch data sources in their code without the need to manually audit the physical units of the new data—provided that the data have proper unit metadata stored in a machine-readable format. In cases where the data do not have such metadata, users are still able to add the necessary unit information.

MetPy and the core stack of Python libraries are released under permissive open source licenses, usually either Berkeley Software Distribution (BSD) 3-Clause or MIT license (Open Source Initiative 2022). These licenses impose minimal requirements for the use of the tools and redistribution of code using them (so-called derived works). This means that, from an intellectual property standpoint, MetPy and the scientific Python stack are suitable for a wide range of uses across the education, research, and commercial application sectors. The permissive licensing also encourages code reuse and helps facilitate downstream development.

MetPy's overall development workflow strives to build a robust and sustainable project. Everything from documentation and code style to testing and deployment is handled via modern, automated processes. By automating many development tasks using the GitHub Actions service, and taking advantage of other externally provided services, the MetPy pipeline of development, documentation, testing, and deployment has been designed to be robust, resilient, and sustainable. A detailed contributor's guide equips prospective contributors to propose changes to the codebase (known as a "pull request"). The full suite of documentation, built using the popular Sphinx Python documentation framework (Sphinx Team 2022), guides users with a mix of narrative text, a thorough gallery of examples and tutorials, helpful theming, and automatic generation of function reference documentation across all of MetPy's functionality. User documentation is automatically published online with every release, and development documentation snapshots are published with every accepted contribution to the MetPy repository.

Beyond the sustainability of the project, MetPy has a focus on being a trustworthy part of its users' computing stack and providing a stable user experience. This starts with the requirement that every calculation cites primary sources for its implementation. By joining scientific citation practices to software development, the software becomes a part of the scientific record and provides the user ample resources to understand how and why a calculation was implemented. Undertaking the often challenging journey to track down the source material also reflects a commitment to having scientific software development follow scholarly norms, instead of treating software as an implementation detail. Beyond the source material, every part of the MetPy code base contains software tests that ensure code continues to work as expected. Tests are automatically run with every proposed and accepted contribution; currently these tests exercise 95% of the lines of code in MetPy. Not only does the coverage of automated tests free developers to make modifications without fear of introducing unknown breakage, it helps the project manage the constant change in the ecosystem around it. The MetPy project pledges not to change user-level interfaces in a way that would cause existing programs to fail during the lifespan of a major release—this means that any code written using MetPy version 1.x should work for any later version 1.y.

Installation of MetPy is as straightforward as the installation of any other Python package. MetPy is available on the Python Package Index (PyPI) and can be installed using the standard "pip" tool as "pip install metpy." MetPy is also available for the Conda package manager using the community conda-forge channel with "conda install -c conda-forge metpy."

Capabilities

MetPy encapsulates a wide breadth of tools across functionality like calculations, file reading, plotting, and interpolation. (Writing data files is done using other libraries like xarray.) Through the modularity and interoperability of the Python ecosystem, users can choose to pull out individual pieces and mold MetPy around their existing workflows, and can integrate different parts of the MetPy toolbox with each other to create brand new workflows altogether.

An early feature of MetPy was implementation of a skew *T*–log*p* diagram that has long been used in the field to display the vertical profile of the atmosphere from radiosonde balloon observations and model simulations thereof (Fig. 1). In addition to being able to plot observed or gridded profiles, MetPy's calculation suite includes the ability to calculate many sounding parameters, a capability which exceeds those of GEMPAK. MetPy also features the ability to plot a hodograph of wind components, which frequently is used alongside such plots.

A limiting factor in moving away from GEMPAK to Python for many years was the inability to plot surface and upper-air observational data using a station model. MetPy has the ability to produce station plots and includes a domain-specific font to plot current weather symbols—a

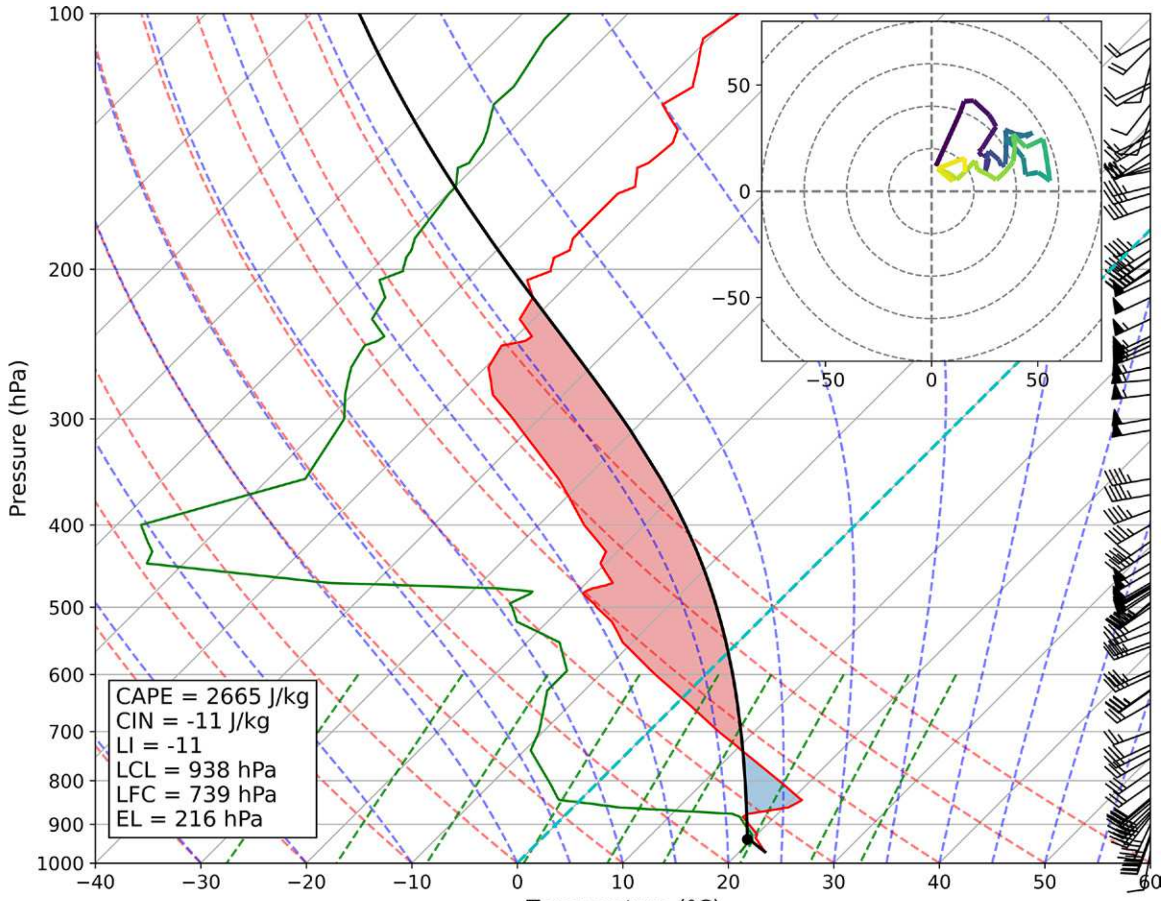


Fig. 1. Vertical profile of the atmosphere, valid 1200 UTC 22 May 2011 from Topeka (TOP), presented on a skew *T*–log*p* diagram. Shown are observed temperature (red line), dewpoint temperature (green line), calculated parcel profile trace (black line), 0°C isotherm (cyan line, dashed), and wind barbs (right axis; kt; 1 kt ≈ 0.51 m s^{-1}), with shaded areas for CIN (blue shaded) and CAPE (red shaded). Calculated indices are inset at the bottom left, and a hodograph is presented in the top right, colored by altitude.

must for any surface map. With the modular implementation of the different plotting elements, it is easy to layer different elements together to create more complex figures that help reveal atmospheric phenomena. For example, a set of surface observations can be overlaid on visible satellite imagery with contours of analyzed equivalent potential temperature (Fig. 2).

MetPy's unit-aware calculation suite covers a wide range of the most commonly used meteorological calculations. The large majority of calculations that are a part of GEMPAK's scalar functions have been implemented in MetPy. Many of the calculations work on data ranging from single values, to one-dimensional arrays, to multidimensional arrays—especially for the dynamic and kinematic functions. Currently, the implementation of a number of key sounding calculations, such as convective available potential energy (CAPE), can only be used with one-dimensional data due to their iterative nature, which precludes fast calculations on a gridded dataset until they are optimized using compiled routines.

MetPy also has support for interpolation of data, both for producing arbitrary cross sections (Fig. 3) from three-dimensional datasets and for creating regular grids of data from irregularly spaced observations, like those seen in surface and upper air observations. In the case of gridding irregular observations, this includes wrapping interpolation functions from the SciPy library, as well as support for natural neighbor interpolation and the Cressman- and Barnes-style interpolation traditionally used in meteorology.

While the netCDF data model and data stored in netCDF format are core to the design of MetPy and its interoperability with other tools, meteorological and atmospheric data are often stored in domain-specific formats. This is especially true when it comes to historical archives, which are crucial for many education and research applications. To facilitate the use of these data in Python, MetPy has support for many important domain-specific formats for data, including NEXRAD level 2, NEXRAD level 3, GINI, METAR, and GEMPAK. Reading data in these formats produces a pandas DataFrame or an xarray Dataset as appropriate;

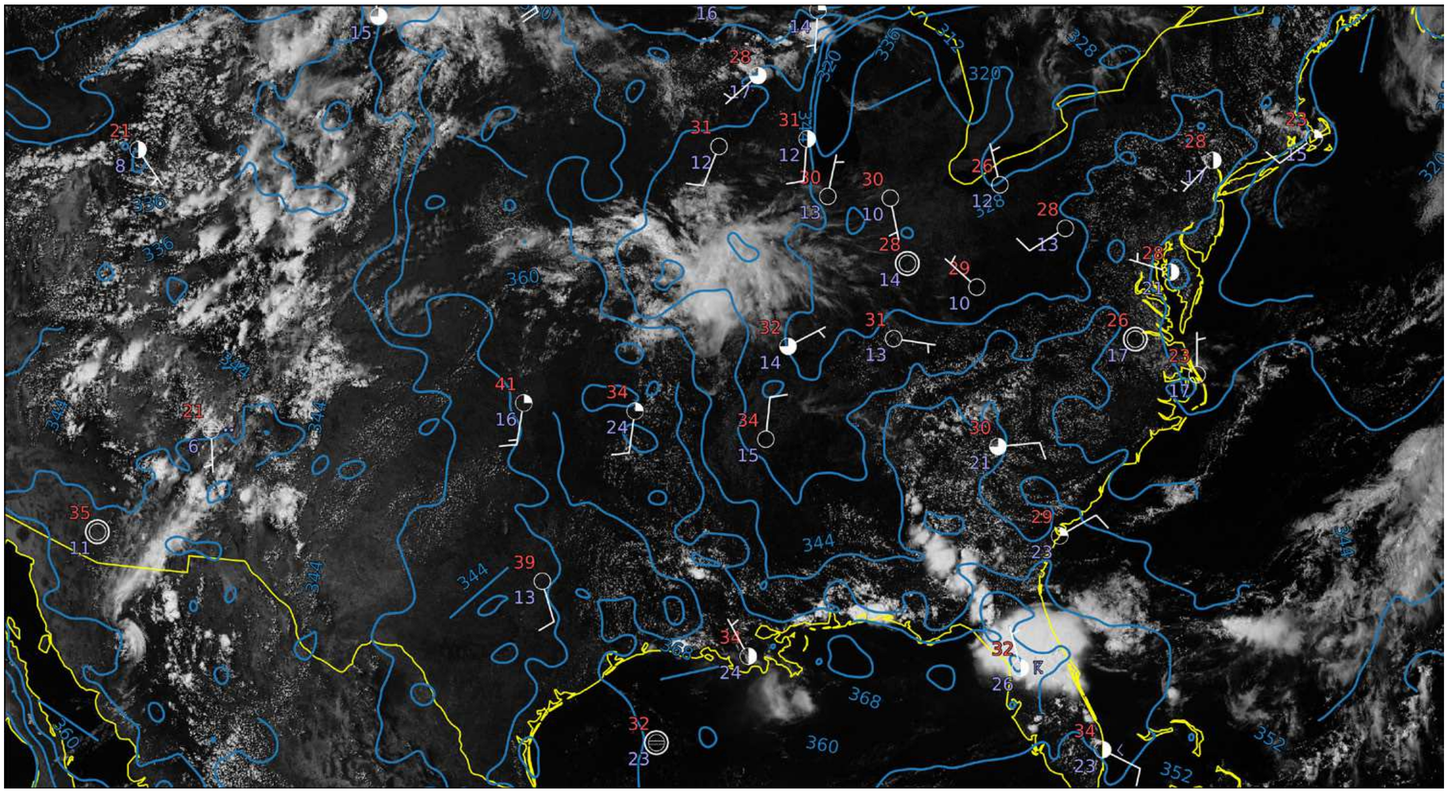


Fig. 2. Map of the continental United States with background *GOES-16* channel 02 imagery valid 1926 UTC 27 Jun 2022, overlayed with contours of equivalent potential temperature calculated from Real-Time Mesoscale Analysis (RTMA) output and station models of surface observations from a collection of surface observation METARs.

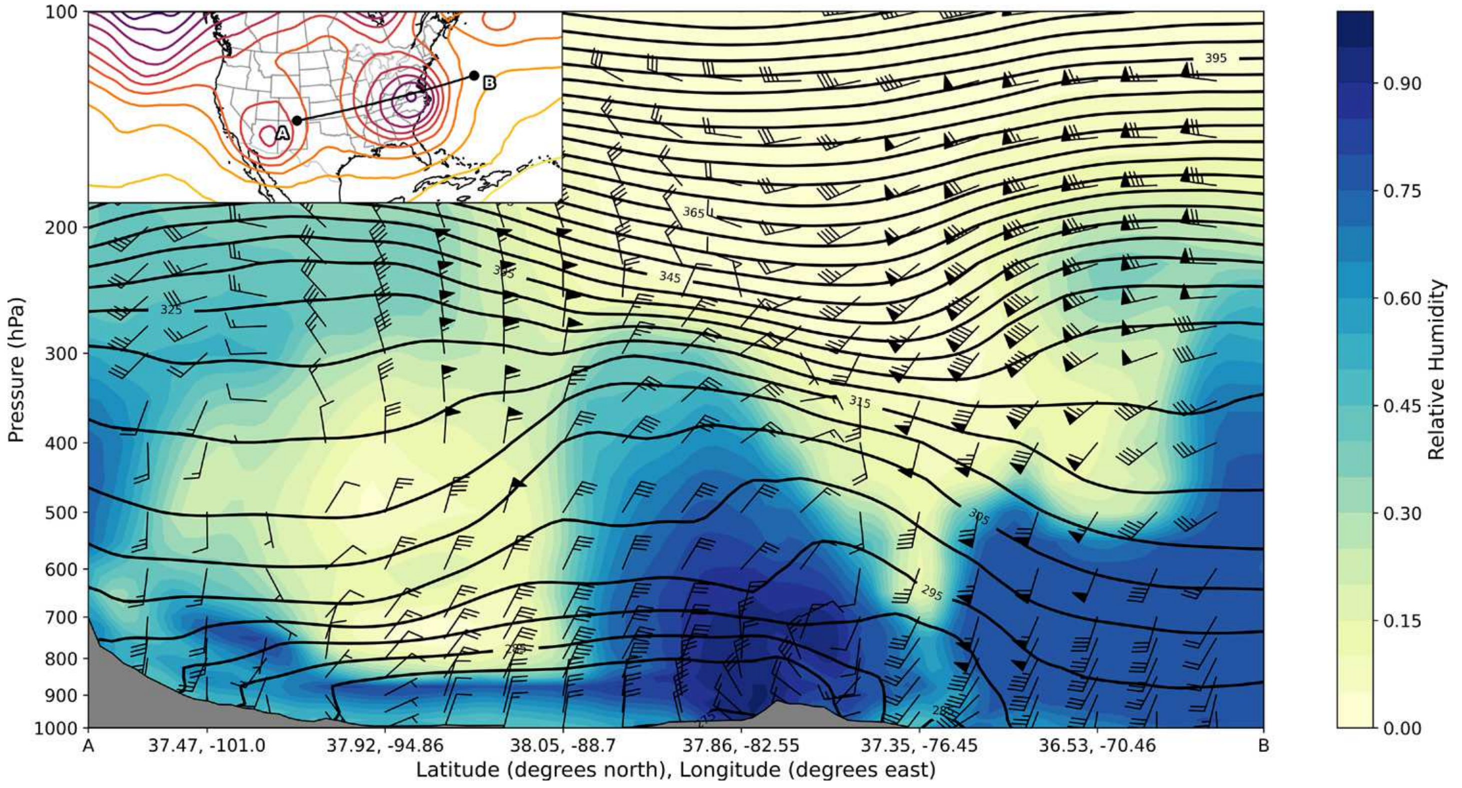


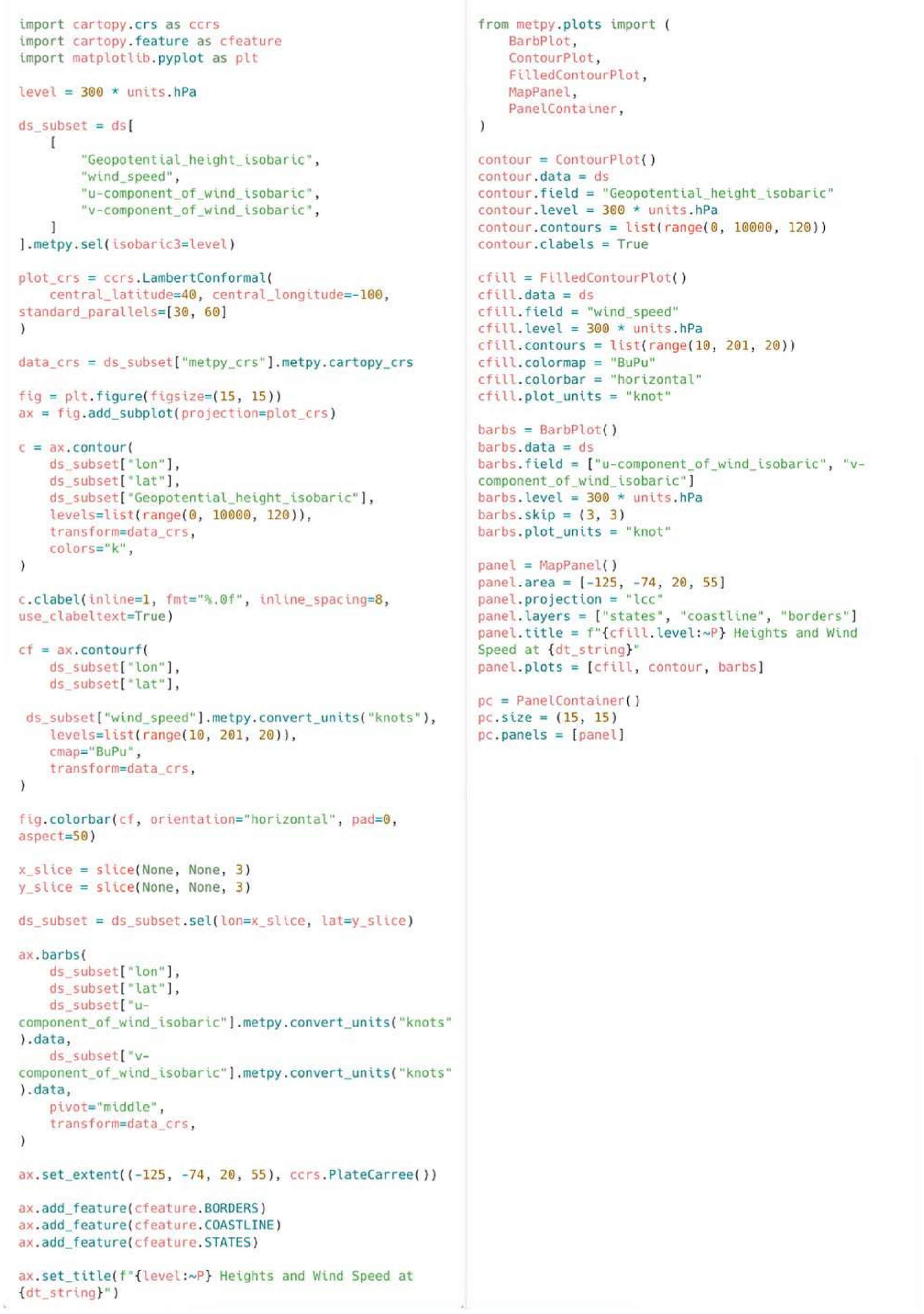
Fig. 3. Vertical cross section of relative humidity (shaded; dimensionless), potential temperature (contours; K), and wind components (barbs; kt) tangential and normal to the plane of the cross section. Latitude, longitude coordinates along the cross-section path are provided along the *x* axis; vertical pressure levels are provided along the *y* axis. The top-left corner inset is a map of the trace of the cross section and contours of 500-hPa geopotential height. Data from North American Regional Reanalysis (NARR) valid 1800 UTC 4 Apr 1987.

an exception is MetPy's NEXRAD readers, which return custom data structures. Other file formats, such as GRIB and HDF5, can be read by existing libraries in the Python ecosystem (see "Software design philosophy" section).

One key element of MetPy is a simplified interface for creating useful data visualizations without the need for deep knowledge of the Python stack. This interface is known within the package as the *declarative syntax* and allows the user to create a plot by specifying a small number of plot and data attributes, which MetPy will interpret into a plot with a "sensible" default presentation. The sensible defaults were chosen to create a low barrier to being able to produce a quality visible representation of a set of data for the CONUS. Producing a similar analysis using a traditional procedural syntax may require nearly twice as many lines of code (Fig. 4) to produce an identical plot (Fig. 5). However, the declarative interface is optimized for common use cases and relies heavily on proper metadata attached to the data, a trade-off for some users and workflows.

As of Metpy version 1.1, the declarative syntax has the capability to produce map-based plots using observations from either surface or upper-air data in the form of station models or wind barbs. Additionally, scalar gridded output (e.g., GFS, NAM, *GOES-16*/*17*) can be contoured, color filled, or image plotted, and vector gridded data can be plotted as wind barbs (Fig. 5). The declarative syntax can also plot shapefile geometries; this allows for easy plotting of data sources like Storm Prediction Center Convective Outlooks (Fig. 6) and National Hurricane Center storm forecasts.

By offering these user-friendly Python tools for atmospheric scientists, like the declarative syntax, MetPy is uniquely equipped to support educators in the field. Since Python has been increasingly adopted by individuals across all sectors of the atmospheric science community, educators need to prepare graduates for success in any of these sectors through integration



```python
import cartopy.crs as ccrs
import cartopy.feature as cfeature
import matplotlib.pyplot as plt

level = 300 * units.hPa

ds_subset = ds[
    [
        "Geopotential_height_isobaric",
        "wind_speed",
        "u-component_of_wind_isobaric",
        "v-component_of_wind_isobaric",
    ]
].metpy.sel(isobaric3=level)

plot_crs = ccrs.LambertConformal(
    central_latitude=40, central_longitude=-100,
standard_parallels=[30, 60]
)

data_crs = ds_subset["metpy_crs"].metpy.cartopy_crs

fig = plt.figure(figsize=(15, 15))
ax = fig.add_subplot(projection=plot_crs)

c = ax.contour(
    ds_subset["lon"],
    ds_subset["lat"],
    ds_subset["Geopotential_height_isobaric"],
    levels=list(range(0, 10000, 120)),
    transform=data_crs,
    colors="k",
)

c.clabel(inline=1, fmt="%.0f", inline_spacing=8,
use_clabeltext=True)

cf = ax.contourf(
    ds_subset["lon"],
    ds_subset["lat"],

 ds_subset["wind_speed"].metpy.convert_units("knots"),
    levels=list(range(10, 201, 20)),
    cmap="BuPu",
    transform=data_crs,
)

fig.colorbar(cf, orientation="horizontal", pad=0,
aspect=50)

x_slice = slice(None, None, 3)
y_slice = slice(None, None, 3)

ds_subset = ds_subset.sel(lon=x_slice, lat=y_slice)

ax.barbs(
    ds_subset["lon"],
    ds_subset["lat"],
    ds_subset["u-
component_of_wind_isobaric"].metpy.convert_units("knots"
).data,
    ds_subset["v-
component_of_wind_isobaric"].metpy.convert_units("knots"
).data,
    pivot="middle",
    transform=data_crs,
)

ax.set_extent((-125, -74, 20, 55), ccrs.PlateCarree())

ax.add_feature(cfeature.BORDERS)
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.STATES)

ax.set_title(f"{level:~P} Heights and Wind Speed at
{dt_string}")
```

```python
from metpy.plots import (
    BarbPlot,
    ContourPlot,
    FilledContourPlot,
    MapPanel,
    PanelContainer,
)

contour = ContourPlot()
contour.data = ds
contour.field = "Geopotential_height_isobaric"
contour.level = 300 * units.hPa
contour.contours = list(range(0, 10000, 120))
contour.clabels = True

cfill = FilledContourPlot()
cfill.data = ds
cfill.field = "wind_speed"
cfill.level = 300 * units.hPa
cfill.contours = list(range(10, 201, 20))
cfill.colormap = "BuPu"
cfill.colorbar = "horizontal"
cfill.plot_units = "knot"

barbs = BarbPlot()
barbs.data = ds
barbs.field = ["u-component_of_wind_isobaric", "v-
component_of_wind_isobaric"]
barbs.level = 300 * units.hPa
barbs.skip = (3, 3)
barbs.plot_units = "knot"

panel = MapPanel()
panel.area = [-125, -74, 20, 55]
panel.projection = "lcc"
panel.layers = ["states", "coastline", "borders"]
panel.title = f"{cfill.level:~P} Heights and Wind
Speed at {dt_string}"
panel.plots = [cfill, contour, barbs]

pc = PanelContainer()
pc.size = (15, 15)
pc.panels = [panel]
```

Fig. 4. Example Python code to produce Fig. 5. (left) Manual creation of figure and map through Matplotlib and Cartopy. (right) Simplified code using MetPy's declarative plotting syntax.

of Python in coursework. Historically, there were substantial barriers to the adoption of scientific Python due to the complex package management and the sheer number of packages needed to accomplish many tasks. With tools like the declarative syntax, MetPy and its suite of functionality lowers the barrier to entry for using Python to analyze and visualize a wide variety of weather data and model output. This means that it can be used very early in an educational journey (e.g., in a freshman undergraduate course), and then be built upon in subsequent educational and research-based experiences by delving deeper into the scientific Python stack. The use of MetPy and Python within the education community creates a smoother transition to computing in research, operations, and other private sector work, where these tools are extensively used.

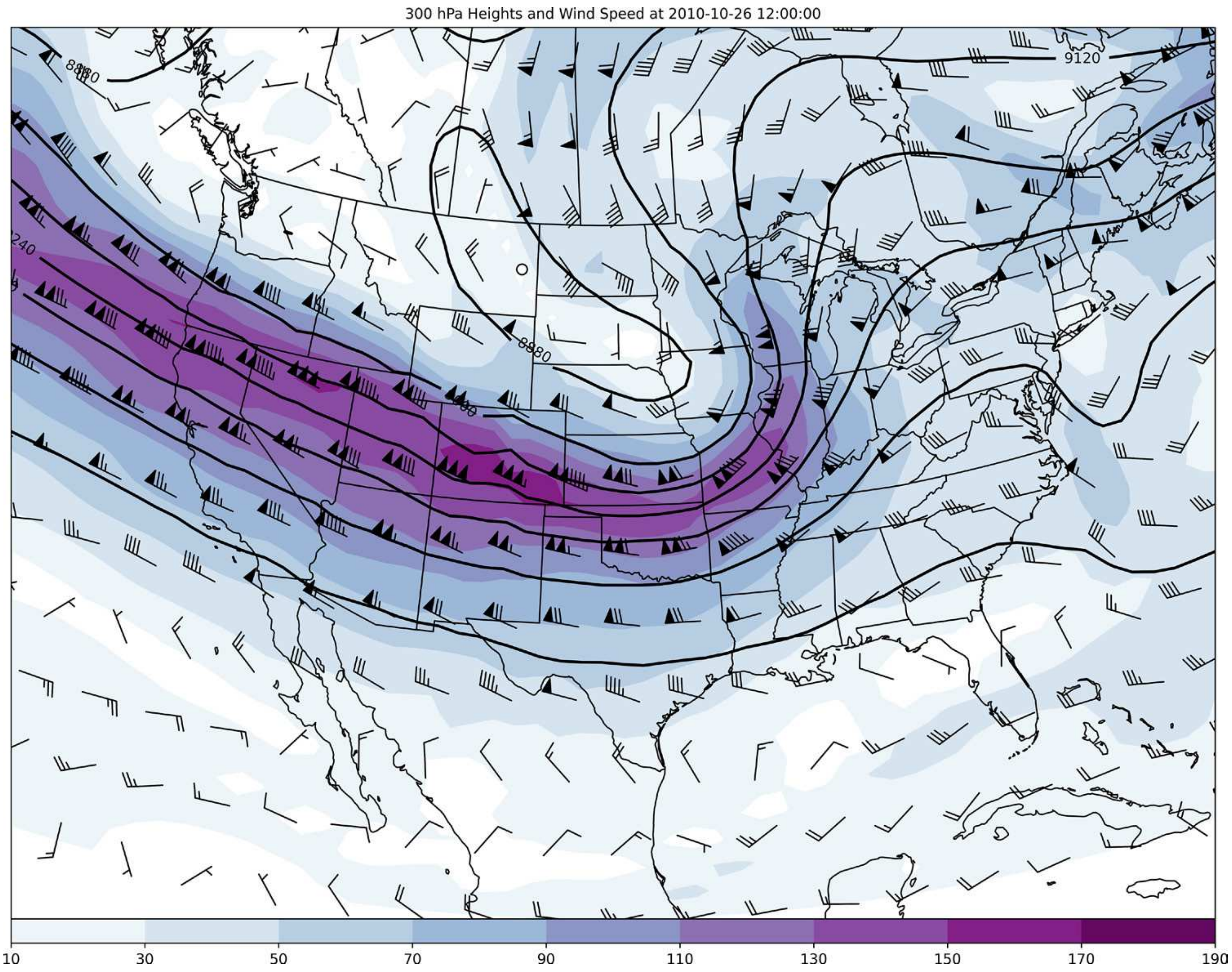


Fig. 5. Geopotential heights (contoured; m), wind speed (filled contours; kt), and wind barbs (kt) from Global Forecast System output valid at 1200 UTC 31 Oct 2010. Example analysis produced using MetPy's declarative plotting syntax demonstrated in Fig. 4.

Community

MetPy is built first and foremost to serve the scientific needs of the meteorological and atmospheric science communities. As such, we place an emphasis on interacting with our community to improve the software, build trust, and help people learn. MetPy's code is hosted on the popular GitHub code collaboration platform. As mentioned previously, MetPy is a permissively licensed open source project, so users are free to browse the code to understand how various portions of the library work and see how calculations are implemented. With the support of Unidata, workshops are held regularly to engage with the community and teach users about MetPy and the broader Python scientific ecosystem.

Through the use of GitHub, MetPy users have a variety of channels for interacting with developers and other community members. Users are encouraged to ask questions using GitHub discussions and to report any problems, suggestions for improvement, and feature requests using GitHub's issue tracker. We supplement these channels with Stack Overflow (a public question and answer site) and Gitter (synchronous web-based chat). We believe this set of channels is critical so that users are able to get the help and guidance they need for using MetPy. The focus on public channels encourages participation in this process by community members, and allows users to see answers to questions that have already been asked.

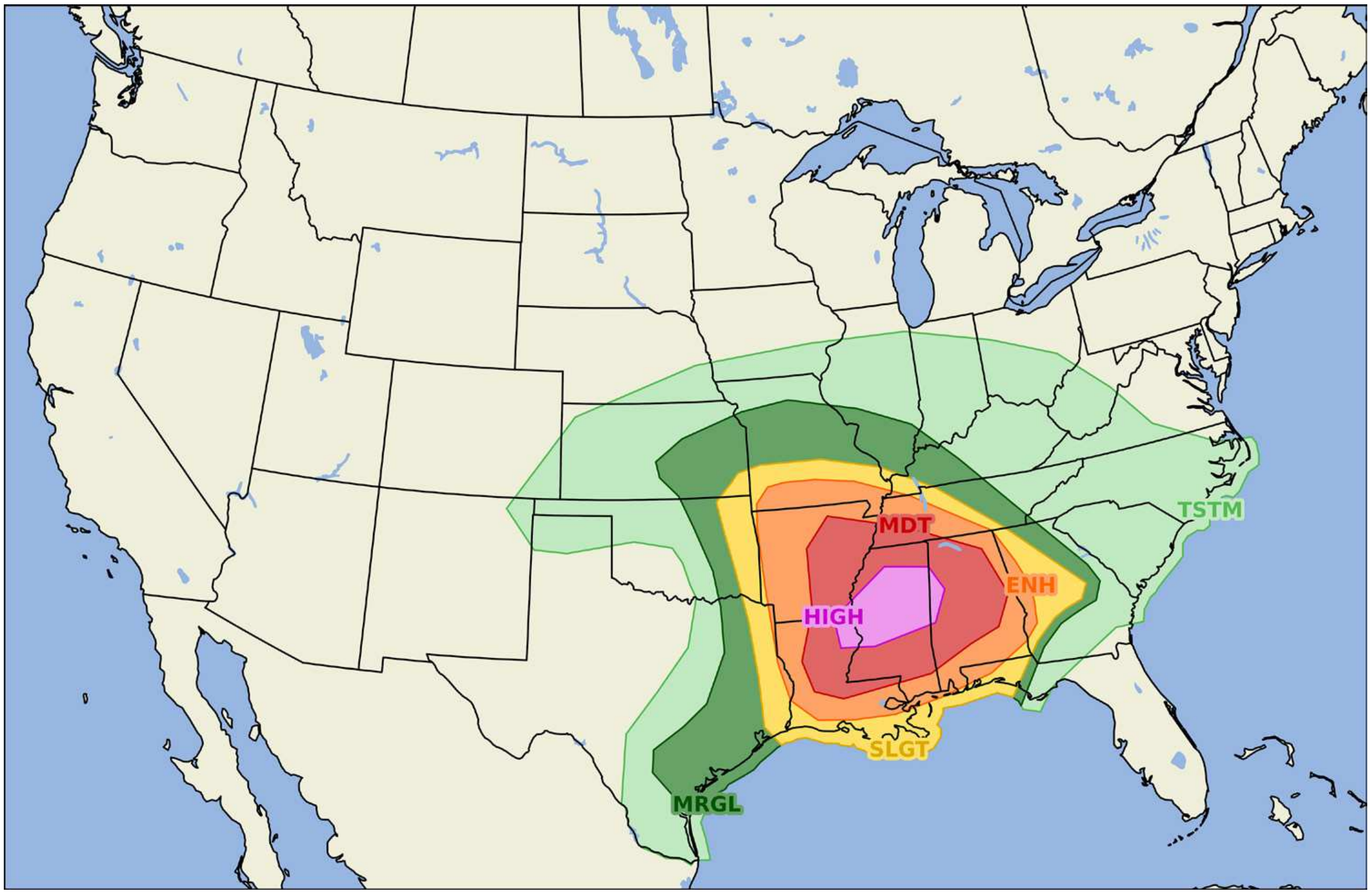


Fig. 6. **NOAA/NWS/Storm Prediction Center (SPC) 1200 UTC 17 Mar 2021 Day 1 Convective Outlook recreated using MetPy's declarative plotting interface. Data from SPC GeoJSON archive.**

As an open source, community-driven project, MetPy actively encourages and solicits contributions from the community. This is one of the chief benefits of being part of the scientific Python ecosystem: users of the library generally possess the baseline programming knowledge needed to modify the library. Contributions to MetPy make use of GitHub's "pull request" feature, whereby any user can request project maintainers "pull in" a suggested set of changes. During this process developers review the submission, provide feedback, and request changes, and ensure that automated tests and style checks pass. This process is iterated until the contributor and maintainers are satisfied with the changes, at which point the contribution is merged into the project source code repository. This open contribution process empowers users to add desired features to MetPy and fix any issues that they have discovered in the code.

The contribution process outlined above applies to all contributions and modifications to the project, even those from the core development team. This is done to help encourage participation as well as producing a historical record of discussions and feedback surrounding changes. MetPy has a public development road map hosted in the documentation, and we have regular virtual developer meetings that are open to the community.

At this point, MetPy represents the collective code contributions of 60 different authors, as well as numerous other users who have contributed to the improvement of the library through their bug reports, feature requests, and support questions asked through a variety of venues. On GitHub alone, 277 users have interacted with the project since 2015. In terms of more general usage, MetPy was downloaded over 208,000 times in 2021 from the conda-forge and Python Package Index repositories; during this time MetPy's documentation averaged just under 35,000 page views per month.

The MetPy development effort receives significant support from the Unidata Program. The Unidata Program Center (UPC) is a community data and software facility sponsored primarily by the National Science Foundation's Division of Atmospheric and Geospace Sciences (AGS), with a mission to help transform the conduct of research and education in the atmospheric and related sciences by providing well integrated data services and tools that address the entire scientific data life cycle. Several of the primary MetPy developers are now or have previously been based at the UPC; other developers are now or have been involved with Unidata's community governance mechanisms. In addition to providing direct support for UPC software engineers' MetPy work, Unidata supports the MetPy community's open development model and community contribution process by providing infrastructure (the MetPy repository takes advantage of Unidata's organizational GitHub account) and training resources.

Future plans

The needs and feedback of the community have set the course for MetPy's continued development. One of the foremost user requests has been performant calculation of parcel-related calculations like CAPE on gridded datasets, which is inefficient in current versions of MetPy. To move past the present inefficiencies in these iteration-based calculations, we will be addressing bottlenecks in the current implementations through the use of Python tools like Numba and Cython for creating optimized, compiled routines.

Another performance-related area for improvement in MetPy involves integration with the Dask Python library. Dask enables larger-than-memory and distributed-memory calculations across large datasets (Dask Development Team 2022). Native support for data provided in Dask arrays will allow MetPy users to much more readily analyze large datasets, such as ensemble and climate model output. This support will necessarily entail comprehensive testing of the calculation library across the wide array of supported data types. Supporting Dask arrays will allow MetPy users to optimize their calculations on large datasets as they see fit. We plan to implement support by testing against this data type, but deliberately making sure that Dask is not a required dependency. Other plans for the future include continuing to add supported data formats (e.g., BUFR, McIDAS area files) and adding to the collection of calculations as requests from the community come in. We also plan to continue to investigate ways of integrating with other new related Python projects, such as PyART for radar data and xgcm for climate models.

Learn more

To learn more about MetPy, visit the web documentation at https://unidata.github.io/MetPy/. The GitHub repository with issue tracker, pull requests, and discussions can be found at https://github.com/Unidata/MetPy.

Acknowledgments. MetPy represents the collective code and documentation contributions of numerous individuals and has been greatly improved by the feedback from its broader user community. MetPy development has been supported by the National Science Foundation through Awards OAC-1740315, OCE-1740633, and OAC-2103682, as well as through its support of the Unidata program (AGS-1344155 and AGS-1901712).

Data availability statement. All MetPy software is publicly available from the official GitHub repository at https://github.com/Unidata/MetPy. Copies of the Python code used to generate figures from publicly available data can be found at https://github.com/Unidata/metpy-bams-2022.

References

Blumberg, W. G., K. T. Halbert, T. A. Supinie, P. T. Marsh, R. L. Thompson, and J. A. Hart, 2017: SHARPpy: An open-source sounding analysis toolkit for the atmospheric sciences. *Bull. Amer. Meteor. Soc.*, **98**, 1625–1636, https://doi.org/10.1175/BAMS-D-15-00309.1.

Dask Development Team, 2022: Dask: Library for dynamic task scheduling, version 2022.2.0. Dask, https://dask.org.

George Mason University, 2018: The Grid Analysis and Display System, version 2.2.1. George Mason University COLA, http://cola.gmu.edu/grads/gadoc/gadoc.php.

Grecco, H. E., and Coauthors, 2021: Pint, version 0.18. Pint, https://pint.readthedocs.io.

Harris, C. R., and Coauthors, 2020: Array programming with NumPy. *Nature*, **585**, 357–362, https://doi.org/10.1038/s41586-020-2649-2.

Hoyer, S., and J. Hamman, 2017: xarray: N-D labeled arrays and datasets in Python. *J. Open Res. Software*, **5**, 10, https://doi.org/10.5334/jors.148.

Hunter, J. H., 2007: Matplotlib: A 2D graphics environment. *Comput. Sci. Eng.*, **9**, 90–95, https://doi.org/10.1109/MCSE.2007.55.

Leemans, R., and Coauthors, 2009: Developing a common strategy for integrative global environmental change research and outreach: The Earth System Science Partnership (ESSP). *Curr. Opin. Environ. Sustainability*, **1**, 4–13, https://doi.org/10.1016/j.cosust.2009.07.013.

McDonald, J. M., and C. C. Weiss, 2021: Cold pool characteristics of tornadic quasi-linear convective systems and other convective modes observed during VORTEX-SE. *Mon. Wea. Rev.*, **149**, 821–840, https://doi.org/10.1175/MWR-D-20-0226.1.

McKinney, W., 2010: Data structures for statistical computing in PYTHON. *Proc. Ninth Python in Science Conf.*, Austin, TX, SciPy, 56–61, https://doi.org/10.25080/Majora-92bf1922-00a.

Met Office, 2021: Cartopy version 0.20.2. SciTools, https://scitools.org.uk/cartopy/.

NCAR, 2019: The NCAR Command Language, version 6.6.2. UCAR/NCAR/CISL/TDD, https://doi.org/10.5065/D6WD3XH5.

Open Source Initiative, 2022: Licenses & standards. Accessed February 2022, https://opensource.org/licenses.

Pandas Development Team, 2022: pandas version 1.4.1. PyData, https://pandas.pydata.org.

Raspaud, M., and Coauthors, 2018: PyTroll: An open-source, community-driven Python framework to process Earth observation satellite data. *Bull. Amer. Meteor. Soc.*, **99**, 1329–1336, https://doi.org/10.1175/BAMS-D-17-0277.1.

Reid, W. V., and Coauthors, 2010: Earth system science for global sustainability: Grand challenges. *Nature*, **330**, 916–917, https://doi.org/10.1126/science.119626.

Schueth, A., C. Weiss, and J. M. L. Dahl, 2021: Comparing observations and simulations of the streamwise vorticity current and the forward flank convergence boundary in a supercell storm. *Mon. Wea. Rev.*, **149**, 1651–1671, https://doi.org/10.1175/MWR-D-20-0251.1.

Sphinx Team, 2022: Sphinx documentation. Accessed February 2022, www.sphinx-doc.org.

Stoelinga, M. T., 2018: Users' guide to RIP version 4.7: A program for visualizing mesoscale model output. UCAR, accessed February 2022, www2.mmm.ucar.edu/wrf/users/docs/ripug.htm.

TIOBE, 2021: TIOBE Index. Accessed December 2021, www.tiobe.com/tiobe-index/.

Unidata, 2019: General Meteorology Package (GEMPAK), version 7.5.1. Unidata, https://doi.org/10.5065/D6H70CW6.

——, 2021: NetCDF, version 4.8.1. Unidata, https://doi.org/10.5065/D6H70CW6.

Virtanen, P., and Coauthors, 2020: SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nat. Methods*, **17**, 261–272, https://doi.org/10.1038/s41592-019-0686-2.

Wade, A. R., and M. D. Parker, 2021: Dynamics of simulated high-shear low-CAPE supercells. *J. Atmos. Sci.*, **78**, 1389–1410, https://doi.org/10.1175/JAS-D-20-0117.1.

Zarr Developers, 2021: zarr-python version 2.10.3. Zenodo, https://doi.org/10.5281/zenodo.5712786.