Pymplot: An open-source, lightweight plotting package based on Python and matplotlib

(November 24, 2020)

GEO-XXXXXX

Running head: **A convenient plotting package**

ABSTRACT

We develop a lightweight, easy-to-use plotting package based on Python and matplotlib for scalar data visualization. The package contains eight plotting functions that can efficiently and conveniently render 1D, 2D and 3D regular-grid scalar data into publication-quality figures of various formats. These plotting functions include plotting 1D scalar data as a curve or a set of colored scatter points, showing 2D regular-grid scalar data as an image, wiggles, or contours, and displaying 3D regular-grid scalar data as a volume or three orthogonal slices in the image or contour form. We develop this package to facilitate quick rendering of 1D, 2D and 3D scalar data into visually decent forms with simple commands and options. The package is also capable of rendering various fonts, subscripts, superscripts, and mathematical symbols on plots in a consistent manner. Example plots demonstrate the efficiency, convenience, and versatility of our plotting package in generating high-quality plots. We make our plotting package open-source at GitHub.

INTRODUCTION

Data visualization is gaining increasing importance in modern scientific research (e.g., Schroeder et al., 2000; Ahrens et al., 2005; Fogal et al., 2010; Ramachandran and Varoquaux, 2011). The reason behind such transition is that modern scientific research, particularly those associated with data analysis, image analysis, and experimental result analysis, essentially relies on descent rendering of data, images, or results, in various forms to convey important information of the research. Scientific disciplines such as geophysics, biology, astrophysics, experimental physics, computer vision, imaging science, among others, always require proper and effective data visualization for publication. In some fields, neat graphical representations of principles and results could explain ideas/findings much more clearly than complex equations and lengthy text descriptions.

Starting from simple dot, line and curve representations in early days, scientific data visualization nowadays enjoys a vast pool of plotting tools than ever. The most notable ones include matplotlib (Hunter, 2007), Mathematica (Wolfram Research, Inc., 2020), MATLAB (MathWorks, Inc., 2020), gnuplot (Williams et al., 2020), VTK (Schroeder et al., 2000), Mayavi (Ramachandran and Varoquaux, 2011), Paraview (Ahrens et al., 2005), VisIt (Childs et al., 2012), OpenGL (Woo et al., 1999), Surfer (Golden Software, LLC, 2020), to name a few without a specific order. For geophysical particularly earthquake and seismic studies, generic mapping tools (GMT) (Wessel et al., 2019), SeismicUnix (SU) (Stockwell Jr. and Cohen, 2008), and Madagascar (Madagascar Development Team, 2012) are probably the most popular packages. The package MinesJTK (Hale, 2015) based on Java programming language with Python interfaces also gains attention in the seismic research community recently.

The aforementioned plotting tools are based on drastically different fundamental-layer libraries, programming languages, orientations, and goals, and they are not necessarily compatible with one another. For example, Madagascar uses its own special data I/O format, which is not widely adopted in any other aforementioned software or libraries. More importantly, some of these tools are generic plotting tools that are applicable to a wide range of visualization tasks, but are not necessarily convenient for rendering geophysical data. For instance, both MATLAB and Mathematica are fancy scientific computational tools, but neither of them is designed primarily for data visualization. Some of these tools require heavy or at least an intermediate level of programming effort to render even a simple dataset. For example, OpenGL and VTK are undoubtedly very comprehensive and powerful, but are generally considered to be not user-friendly for usual users, and both of them require a very steep learning curve. GMT partially shares this drawback of a steep learning curve until probably the most recent version (Wessel et al., 2019). Another reason that motivates us to develop a plotting package for routine plotting tasks is that many of these aforementioned plotting tools are commercial software, and require high initial and annual subscription fees to use.

Though geophysical data may be in any modern data storage forms, such as regularly sampled, irregularly sampled, unstructured, multi-component (vector or tensor), the most frequently used data form is perhaps the regularly sampled scalar data in 2D and 3D shapes, e.g., seismic velocity, density, and subsurface images. Very frequently, researchers in the geophysics community only need to render these 2D and 3D data into simple forms such as 2D images, contours, 3D image volumes, or multiple slices. In such a case, some of these aforementioned tools may be overkill for these plotting tasks.

With consideration of convenience, simplicity, and lightweight in mind, we develop a

plotting package based on Python and matplotlib for visualizing 1D scalar data and 2D/3D regularly sampled scalar data.

We have no intention to develop our plotting package, `pymplot` as we name it for now, to surpass any existing plotting software. Our intention is to provide a convenient application layer to Python's native plotting library, matplotlib. We choose Python and matplotlib because both of them are free, open-source, de facto most widely used, and portable on different operating system platforms. In a sense our package is not an independent, generic, or fundamental layer or library. We design and implement the `pymplot` package mainly to render regularly-sampled scalar data to a limited number of visualization forms (images, contours, wiggles, volumes, etc.). It is therefore not comprehensive enough to rendering complex data forms other than regularly sampled scalar data, including unstructured (e.g., unstructured 2D or 3D mesh) or multi-component (e.g., vector or tensor). Properly rendering these types of complex data usually require significant advanced programming efforts, particularly for 3D unstructured or multi-component data. In short, our plotting package is a collection of *applications* with specific purposes, instead of a *library* with generic functions and interfaces.

Our paper is organized as follows. In the function description and examples section, we describe the eight plotting functions in our package. Out of succinct consideration, we describe the main features of these plotting functions and show one to two examples to demonstrate each function. For the 3D plotting functions, we also use two illustrations to describe the layout of the generated plots. We summarize our paper in the Conclusions section.

PLOTTING PACKAGE "PYMPLOT"

We develop a plotting package called `pymplot` based on Python and matplotlib for visualizing 1D scalar data and 2D/3D regularly sampled scalar data. Our package contains eight core plotting functions, including one 1D visualization function (`showgraph`), three 2D visualization functions (`showmatrix`, `showcontour`, and `showwiggle`), and four 3D visualization functions (`showslice`, `showslicon`, `showvolume`, and `showvolcon`).

The purposes of these eight plotting functions are:

1. `showgraph`: To show 1D scalar data as curves or scatter points with or without colors representing their attributes.
2. `showmatrix`: To show 2D regularly sampled scalar data as a colored image.
3. `showcontour`: To show 2D regularly sampled scalar data as contours.
4. `showwiggle`: To show 2D regularly sampled scalar data as wiggles. The 2D data should be regularly sampled along at least one of the two axes.
5. `showslice`: To show 3D regularly sampled scalar data using three orthogonal slices, each with in a colored image.
6. `showslicon`: To show 3D regularly sampled scalar data using three orthogonal slices, each in the form of contours.
7. `showvolume`: To shown 3D regularly sampled scalar data using a volume in the orthogonal projection, with a sub-volume cropped out. The volume and sub-volume surfaces are in 2D colored images.
8. `showvolcon`: To shown 3D regularly sampled scalar data using a volume in the orthogonal projection, with a sub-volume cropped out. The volume and sub-volume surfaces are 2D contours.

All these plotting functions use Python scripts. They take command line arguments with input data, and then output the desired figures in different formats, such as pdf, png, jpg, tiff, and some others. They can generate multiple formats of figures using the same command by specifying multiple output formats in the output option, eliminating the needs of writing multiple scripts to generate the same plot in different figure formats. All these plotting functions allow users to choose from a large set of well-tuned colormaps, providing visually descent images through simple commands and easy-to-tune options. These plotting functions enable users to generate publication-quality plots with simple command-line scripts in a consistent style and manner, alleviate users' burden to write and tune lengthy codes.

The options associated with each of these functionalities are explained in the user documentation of the package. Each functionality contains tens of options and these options are displayed when no option is given to the functionality command.

FUNCTION EXAMPLES

We demonstrate the aforementioned functionalities using several simple examples. We provide the scripts to generate these plots in the open-source package.

1D data visualization

One function of our plotting package is `showgraph` for rendering a 1D scalar data into a curve or scatter points with or without colors.

1D data are a set of data points specified by a pair or coordinates, say, (x, y), with or without one additional scalar value to represent the attribute of the points. Strictly speak-

ing, some of these data, particularly the ones with the third value, should be categorized as 2D data. We still call them 1D data to differentiate them from the regularly sampled, scalar data we use in 2D and 3D plotting.

Rendering 1D data is widely used in scientific visualization. For instance, visualizing the convergence of an inversion process over iterations, spatial locations of seismic sources and receivers, or a set of irregularly sampled gravity data in space, etc. Rendering 1D data is well supported in almost every existing plotting software, and the associated syntax is usually fairly straightforward and simple. The function is peripheral of our package.

Our package supports straightforward rendering of a set of data specified by (x, y) coordinates. When designing this function, we allow the input data being in either the ASCII format or the raw binary format, so that reading a large dataset into our code for plotting does not become a time-consuming task.

Figure 1 shows a simple example of rendering the qP- and qSV-wave group velocity curves using `showgraph`. The two group velocity curves are specified in their separate files, `qP.txt` and `qSV.txt`, each with a total of 360 scatter points in the format of $gx1$ $gz1$, $gx2$ $gz2$, $\cdots$, line by line. Our plotting program simply uses option `-in=qP.txt,qSV.txt` to read the contents of these two ASCII files, and to make the plot. Multiple datasets can be conveniently read in using `-in=data1,data2,data3,`$\cdots$. For each dataset, the plotting style (color, marker shape, marker size, curve width, closed or open, etc.) can be specified through adding additional values to the options provided by `showgraph`. For instance, given a total of five datasets, the line colors can be specified with `-linecolor=b,r,k,p,c`, representing setting blue, red, black, purple, and cyan for the five lines, respectively. Other plot styles can be set similarly.

Our package can also conveniently render 1D dataset with the value attributes, i.e., specified by (x, y, v) where v is the value attribute associated with the scatter data points. In our package, we use option `-color` to allow users to choose a suitable colormap to colorize these data points based on their values, and options `-markersizemin` and `-markersizemax` to specify the size range of the data points according to the magnitudes of their values. Users do not have to specify these options because all these options have their default values.

Figure 2 displays an example of rendering a dataset consisting of randomly distributed scatter points in space, with random attribute values, using `showgraph`. We specify a uniform colormap (`magma`) instead of the default colormap (`jet`) for these data points. We use varying sizes for these data points with a set of disks of different radii. In Figure 2, we also illustrate the convenience of our package to specify the base for each axis. In this case, we set `-norm2=log` so that Axis 2 (the vertical axis) is in the logarithmic scale, while Axis 1 is in the linear scale (`-norm1=linear` is default). In addition, it is easy to use our plotting program to add simple or complex mathematical texts (e.g., the annotation $\Delta\omega = 80\%$ in Figure 2), filled polygons (e.g., the yellow quadrilateral at the lower-left corner), curves (the blue dashed lines), and arrows (e.g., the red arc arrow with double arrow heads), etc, making our plotting program `showgraph` a convenient tool to achieve complex plotting with just simple options. The convenience is an advantageous feature of our plotting program compared with existing software, such as SeismicUnix and Madagascar.

[Figure 1 about here.]

[Figure 2 about here.]

2D data visualization

Color images, contours, and wiggles are probably the three most frequently used data visualization forms in the applied geophysics community to display 2D data. Our package contains three plotting functions, `showmatrix`, `showwiggle`, and `showcontour`, for rendering a 2D regularly sampled array data into an image, wiggles, and contours, respectively. These plotting functions read a single 2D regularly sampled dataset stored in the raw binary format as the input.

Figure 3 shows the rendering result using `showmatrix` for the BP-2004 P-wave velocity model (Billette and Brandsberg-Dahl, 2005). We use the default colormap, `jet`, to colorize the model. We also add several rectangles, arrows, and text labels on the plot to demonstrate the convenience of our package to use annotations for various kinds of highlighting. We can add all these annotations to the plot using command line options within a single command, rather than creating them separately and superimposing them to the plot like other packages such as GMT. Although this strategy limits the possibility of creating extremely complex plot using `showmatrix`, our experience is that, in most cases, it is sufficient to add only several additional annotations.

[Figure 3 about here.]

It is simple to choose from a large set of colormaps to render different types of data using our package. In addition to tens of well-tuned colormaps natively available through matplotlib, we also create several visually descent colormaps in the codes. Together they should suffice for most of plotting tasks. Colormap tuning is usually a fairly time-consuming task. Unlike in the plotting functions of SU, users never need to define and tune colormaps

by themselves using our plotting program. We also provide several font typeface options in the package. Users can choose from a set of carefully chosen fonts including Arial, Helvetica, Times New Roman, Consolas, Courier New, Courier Prime, etc., for their plotting tasks. These fonts belong differently to sans, sans-serif, and monospaced typefaces, and should be sufficient for a wide range of plotting tasks. Additional fonts are available by slightly modifying the codes. Our experience is that sans and monospaced fonts are good choices for plotting in most cases. We choose `arial` (Arial) font as the default font for our plotting functions to ensure clear visualization for publication and different types of presentations.

Figures 4a-f show the Marmousi-II P-wave velocity model (Martin et al., 2006) using the `jet`, `gist_ncar`, `bwr` (custom blue-white-red), `binary` (black-white), `viridis` (viridis), and `rainbow256` (custom blue-red rainbow) colormap, respectively. Figures 4a-f also display the use of six different fonts, including `arial` (Arial), `consolas` (Consolas), `times` (Times New Roman), `courier` (Courier Prime), `plex` (IBM Plex), and `helvetica` (Helvetica).

[Figure 4 about here.]

Contours are frequently used in temperature maps, traveltime fields, tomographic inversion results, gravity fields, etc. Contour plotting is one of the most widely used visualization formats in science. Figures 5a and b illustrate two examples of the 2D contour plot generated using `showcontour`. In Figure 5a, we display a traveltime field computed using the eikonal equation in a heterogeneous medium. The interval of major contours can be specified with option `-contourlevel=`, and the width and color of the contours can be specified using options `-contourwidth=` and `contourcolor=`, respectively. It is also convenient to set the number of minor contours between any two adjacent major contours using option `-mcontour=`. In this example, we use `-mcontour=1`, meaning there is one minor contour

between any two adjacent major contours. We can fill between contours based on the values of contours using `-contourfill=1`, and show the colorbar associated with this contour color filling using option `-legend=1`. As in `showmatrix`, users can choose a colormap `showcontour` from a large pool of colormaps to fill between contours based on their values.

In Figure 5b, we show an example of contour plotting using `showcontour` for a set of irregularly distributed contours using option `-contours=0.3,0.4,1,1.2,1.5,1.7,2`. We also set three minor contours between any two adjacent major contours using `-mcontour=3`. In this case, the minor contours are evenly distributed between any two adjacent major contours, but are not evenly distributed across the entire value range. The contour values in these two plots can be turned off by setting the font size of contour labels to zero, i.e., `-clabelsize=0`.

In the contour plots, one can either use a mask file through `-mask=` or by setting corresponding values in the data file to `NaN` (a.k.a. Not-A-Number), to mask out the region. For example, in Figure 5b, we mask out a region near the top of the plot using `NaN` in that region to create a visually equivalent surface topography. We also illustrate the possibility of assigning flexible tick labels in Figure 5a. For example, we assign a text tick label "`Start`" at the position of 0, and assigned mathematical symbols α and $G(\omega)$ at two other locations along the horizontal axis by setting `-ticks=loc1:text1,loc2:text2,...`, where `loc#` is the true tick location, and `text#` is the desired tick label placed at `loc#`.

[Figure 5 about here.]

The function `showcontour` can superimpose contours onto a background image. For instance, in Figure 6, we show a single-contour plot of a traveltime field in an anisotropic medium, with a background image of the wavefield snapshot at this time step. The feature

is convenient to render two different datasets, one contour plot and one image plot, within a single plot. This feature is also available in `showmatrix`. In Figure 6, we also illustrate the possibility of rendering complex axis labels, including Greek letters, subscripts, superscripts, and mathematical symbols, using our plotting package. Such feature is missing in widely used plotting packages such as SU, partially because they are incapable of using LaTeX.

[Figure 6 about here.]

The function `showwiggle` is specifically for rendering acoustic and elastic waveforms. Figure 7a is an example of a wiggle plot for a seismic common-shot gather. Figure 7a represents the data using plain wiggles, with filled positive amplitudes using option `-fill=1`. To fill the negative part of waveforms, one only needs to set `-fill=-1`. To show only the wiggles without filled amplitudes, one can set `-fill=0` (the default option). Figure 7b is an example of wiggle plot for the same dataset as in Figure 7a, superimposed onto a background image. The background image uses the same dataset, and is colored in `binary` (black-white colormap). The background image can use a different dataset other than the one for the wiggle plot, as long as the two datasets have the same dimensions, that is, the numbers of temporal samples and traces, and the sample spacing along each axis, are all the same for the two datasets.

[Figure 7 about here.]

In some circumstances, it is useful to plot two datasets within one single plot with different wiggle colors for waveform comparison. Figure 8 is an example of wiggle plot generated using `showwiggle` for two datasets, represented by blue and red wiggles, respectively, with option `-wigglecolor=b,r`. The two wiggle plots have different wiggle line widths, specified

through option `-wigglewidth=1,2`. We also add an green triangle annotation in this plot to demonstrate the versatility of graphical or textual annotations in our plotting package. An optional plot label legend is at the top right corner of the plot, with legends consistent with the color, width, and type of each plotted wiggle. Wiggles in Figures 8 are also possible to be displayed along the vertical direction as those in Figure 7. In addition, there are no limitations on the number of wiggles in one plot, although plotting multiple datasets in one single plot might result in messy rendering in some cases.

[Figure 8 about here.]

3D data visualization

Our plotting package contains four functions, `showslice`, `showslicon`, `showvolume`, and `showvolcon`, to render 3D regularly sampled scalar data as three slices of image, three slices of contours, a 3D volume in the parallel projection, and 3D volume contours in the parallel projection, respectively. We use parallel projection to visualize a 3D volume in `showvolume` and `showvolcon`, that is, any two opposite edges on the same surface are parallel to each other in our plot. The perspective projection, which can generates more realistic rendering views, is somehow difficult to achieve based on matplotlib, and therefore is our future work.

The function `showslice` displays three orthogonal slices selected from a regular-grid 3D scalar data volume. Our plotting program selects three orthogonal slices based on given slice positions defined using option `-slice1=zz -slice2=yy -slice3=xx`, where `xx`, `yy`, and `zz` represent slicing positions along three axes, and then displays these three slices in the layout shown in Figure 9. The three slices occupy the positions of the top left, bottom left, and bottom right corners of the plot. By default, `showslice` displays the three slices

at the centers of their respective axial ranges, i.e., $s_i = o_i + \frac{1}{2}(n_i - 1) \times d_i$, where o_i, n_i, and d_i are the origin, number of samples, and sample interval of the ith axis, respectively.

The top right area of the plot is blank by default, but it can be filled with an existing image. For example, we can generate a 3D view of the image volume using some external tools such as Paraview or Mathematica, and then place the 3D view at the top right area of the plot using option `-vol3d=xx` where `xx` is the name of the existing image file. The existing image is displayed "as it is," and fills the top right area with the area's maximum width or height depending on the ratio of the existing image. This function is useful when one needs to display, for instance, a 3D view of the image volume associated with the slicing position. Such 3D view rendering is difficult for matplotlib to generate, but is fairly easy for such as Paraview or Mathematica to produce. The plotting function `showslice` can place the colorbar at the bottom, left, or top of the plot. By default, it places the colorbar on the right side of the plot.

Figure 10 displays a slice view of the overthrust model (Aminzadeh et al., 1997) generated using `showslice`. The figure uses a large horizontal-vertical aspect ratio as it is in the model. The horizontal axes are "squeezed" to accommodate the rendering, which is specified using options `-size2=yy -size3=xx`, where `xx` and `yy` are some numerical values.

Figure 11 is an example of using our `showslice` to render a 3D subsurface imaging result, superimposed by the detected faults with colors representing the fault probability. This rendering superimposes the second image on the top of the first image, with tuned opacity of the top image. This feature enables us to visualize not only slices from a 3D image volume, but also some other features of the image volume that provide complementary information. We also create an additional 3D view of the image volume using VTK, and

incorporate this external image to our plot in Figure 11 using option `-vol3d=scene3d.png`. The colorbar on the right-hand side of the figure is associated with the fault probability shown on the three slices. Currently, our plotting program only supports plotting the colorbar for one image, the image on the top.

[Figure 9 about here.]

[Figure 10 about here.]

[Figure 11 about here.]

Figure 12 illustrates an example of using `showslicon` to display an image volume using contours on three slices. This function is an extended version of `showcontour`, where the options associated with contours for are identical for the three slices. Similar to `showslice`, `showslicon` also allows an optional image on the top right area of the plot. In Figure 12, we create a 3D view containing isosurfaces and three orthogonal slices of the data using VTK, and place the 3D view at the top right area of the plot.

[Figure 12 about here.]

The other two plotting functions for 3D data visualization are `showvolume` and `showvolcon`, for displaying 3D regular-grid data in the form of a volume with a sub-volume cropped out. The plotting function can add a colorbar to the plot as those aforementioned plotting functions. The colorbar can be placed on the left, right, top, or bottom of the rendered volume for flexibility. We use an illustration in Figure 13 to show the layout in `showvolume`.

Figure 14 depicts a 3D plot of the SEG/EAGE salt model (Aminzadeh et al., 1997) generated using `showvolume` at two different view angles and slicing positions. Using option

`-angle= angle1,angle2`, where `angle1` and `angle2` are two numeric values to specify the rotations of the rendered volume, the plotting functions `showvolume` and `showvolcon` can create slightly more realistic volume representation as shown in Figure 14b compared with that in Figure 14a.

It is also convenient to use our plotting functions to display the four upper octants of the volume using options `-octant=-++`, `-octant=-+-`, `-octant=--+`, or `-octant=---`, where "`-`" and "`+`" represent the lower or upper half of an axis, respectively. For instance, the octant shown in Figure 14a is `--+`, while the octant shown in Figure 14b is `-++`. Currently, our plotting program only supports `-` (i.e., the lower if measured in value, or shallower if measured in depth) octants of the first axis, and supports both `-` and `+` for the other two axes.

Different from those shown in all previous examples, all the plotting elements in `showvolume`, including the axes with ticks and labels, the colorbar with ticks and labels, are not native elements in matplotlib because matplotlib does not support a native tilt axis and associated ticks and labels. We write our own Python codes to render these elements.

[Figure 13 about here.]

[Figure 14 about here.]

Figure 15 displays two example of volume contour plots generated using `showvolcon`. Figure 15a shows a dataset in the linear scale, while Figure 15b depicts another dataset in the logarithmic scale, with different colormaps. Figure 15b shows that `showvolume` and `showvolcon` can use complex axial and colorbar labels. For example, we can assign ω_1, ω_2 and ω_3 to the three axes in Figure 15b, all are Greek letters with subscripts. The plotting

packages can also use other complex labels using LaTeX expressions.

[Figure 15 about here.]

CONCLUSIONS

We have developed a lightweight, simple, command-line-based plotting package called `pymplot` based on Python and matplotlib for rendering 1D scalar data, and 2D and 3D regularly sampled scalar data. The package consists of eight plotting functions for generating curves or scatter points for 1D data, images, contours, or wiggles for 2D data, and slices and volumes for 3D data. We have described the main features of these plotting functions, and demonstrated their simplicity and versatility using several examples. Benefited from using matplotlib, our plotting programs allow users to choose from a large set of well-tuned colormaps, and to generate visually descent plots using simple commands and easy-to-tune options. These plotting functions enable users to effectively generate publication-quality plots in a consistent style and manner, alleviate users' burden to write and tune lengthy plotting scripts or codes. Our codes are open-source at GitHub. Future work aims at incorporating perspective projection to volume plotting functions and developing other plotting functions to render different types of data.

REFERENCES

Ahrens, J., B. Geveci, and C. Law, 2005, Paraview: An end-user tool for large data visualization: Elsevier München, volume **717** *of* The Visualization Handbook.

Aminzadeh, F., J. Brac, and T. Kunz, 1997, SEG/EAGE 3-D Salt and Overthrust Models, *in* SEG/EAGE 3-D Modeling Series, No. 1: Distribution CD of Salt and Overthrust models.

Billette, F., and S. Brandsberg-Dahl, 2005, The 2004 BP velocity benchmark, *in* 67th Annual International Meeting, EAGE, Expanded Abstracts: EAGE, B035.

Childs, H., E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil, 2012, VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data, *in* High Performance Visualization–Enabling Extreme-Scale Scientific Insight: 357–372.

Fogal, T., H. Childs, S. Shankar, J. H. Krüger, R. D. Bergeron, and P. J. Hatcher, 2010, Large data visualization on distributed memory multi-GPU clusters: High Performance Graphics, **3**, doi: 10.2312/EGGH/HPG10/057-066.

Golden Software, LLC, 2020, Surfer®: Golden Software, LLC.

Hale, D., 2015, Mines Java Toolkit (JTK): Open-source Java packages for science and engineering: inside.mines.edu/~dhale/jtk.

Hunter, J. D., 2007, Matplotlib: A 2D graphics environment: Computing in Science & Engineering, **9**, no. 3, 90–95, doi: 10.1109/MCSE.2007.55.

Madagascar Development Team, 2012, Madagascar software, version 1.4: www.ahay.org.

Martin, G. S., R. Wiley, and K. J. Marfurt, 2006, Marmousi2: An elastic upgrade for Marmousi: The Leading Edge, **25**, no. 2, 156–166, doi: 10.1190/1.2172306.

MathWorks, Inc., 2020, MATLAB, Version R2020a: The MathWorks, Inc.

Ramachandran, P., and G. Varoquaux, 2011, Mayavi: 3D visualization of scientific data: Computing in Science Engineering, **13**, no. 2, 40–51, doi: 10.1109/MCSE.2011.35.

Schroeder, W. J., L. S. Avila, and W. Hoffman, 2000, Visualizing with VTK: A tutorial: IEEE Computer Graphics and Applications, **20**, no. 5, 20–27, doi: 10.1109/38.865875.

Stockwell Jr., J. W., and J. K. Cohen, 2008, The New SU User's Manual.

Wessel, P., J. F. Luis, L. Uieda, R. Scharroo, F. Wobbe, W. H. F. Smith, and D. Tian, 2019, The Generic Mapping Tools Version 6: Geochemistry, Geophysics, Geosystems, **20**, no. 11, 5556–5564, doi: 10.1029/2019GC008515.

Williams, T., C. Kelley, and many others, 2020, Gnuplot 5.2: An interactive plotting program: www.gnuplot.info.

Wolfram Research, Inc., 2020, Mathematica, Version 12.1: Wolfram Research, Inc.

Woo, M., J. Neider, T. Davis, and D. Shreiner, 1999, OpenGL programming guide: The official guide to learning OpenGL, version 1.2: Addison-Wesley Longman Publishing Co., Inc.

LIST OF FIGURES

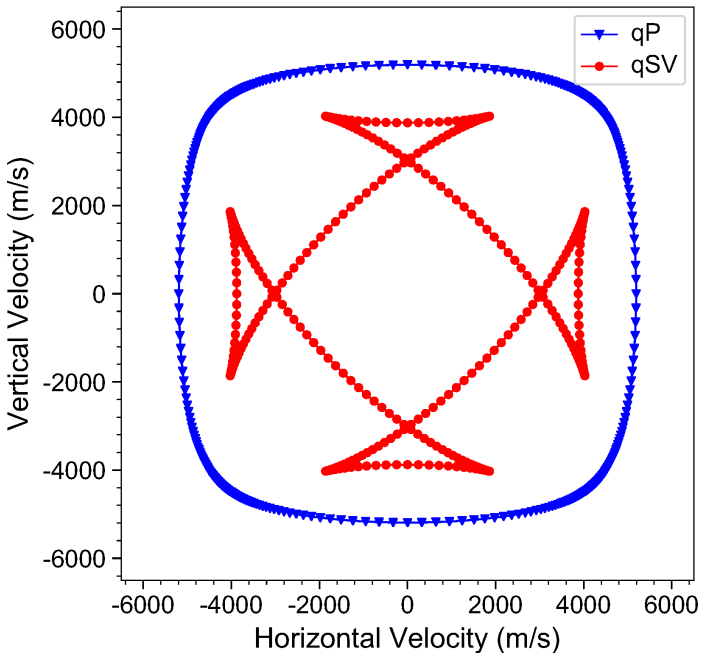


Figure 1: An example of rendering qP- and qSV-wave group velocity curves in an anisotropic medium using `showgraph`.

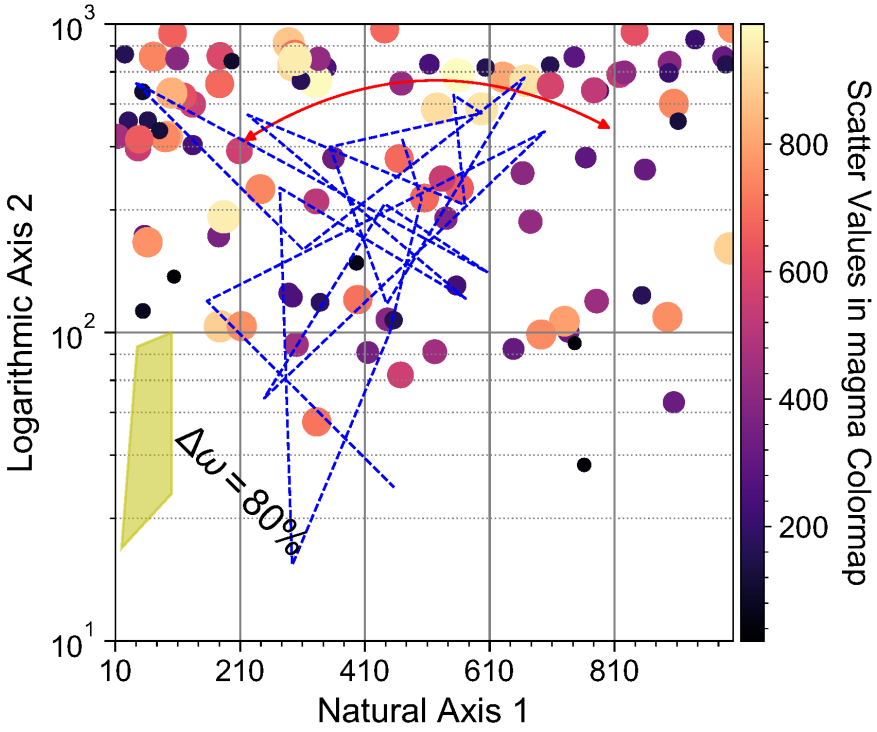


Figure 2: An example of rendering a set of randomly distributed points with different attribute values using `showgraph`. The plot also shows that `showgraph` can add several different types of annotations, including curves, arrows, text, and polygons, to the plot, using simple options.

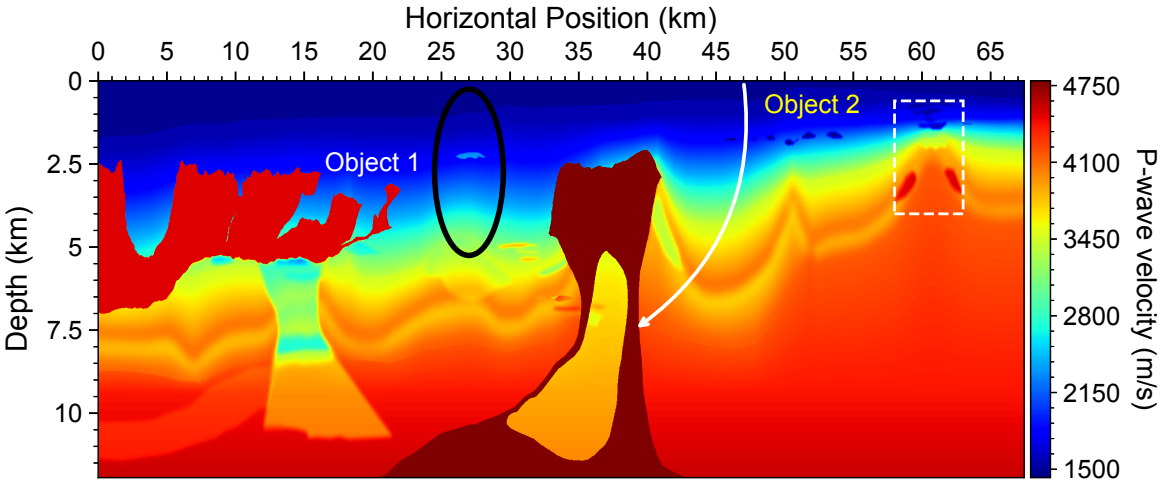


Figure 3: An example of 2D image plot with different types of annotations generated using `showmatrix` for the BP-2004 model. The colorbar can be optionally placed on the left, right, top, or bottom of the plot. By default it is placed on the right of the plot with a height matching the height of the entire image.

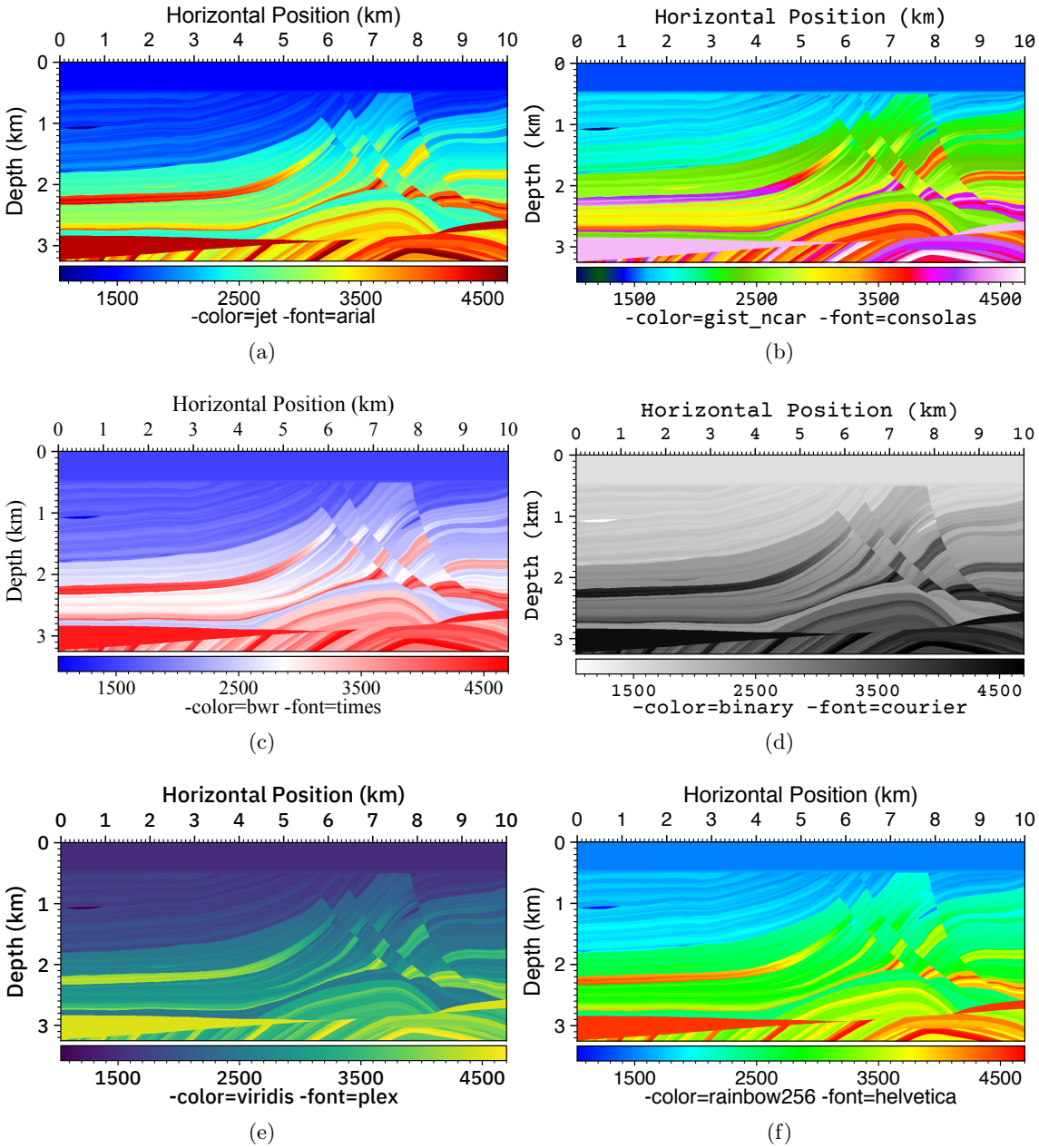
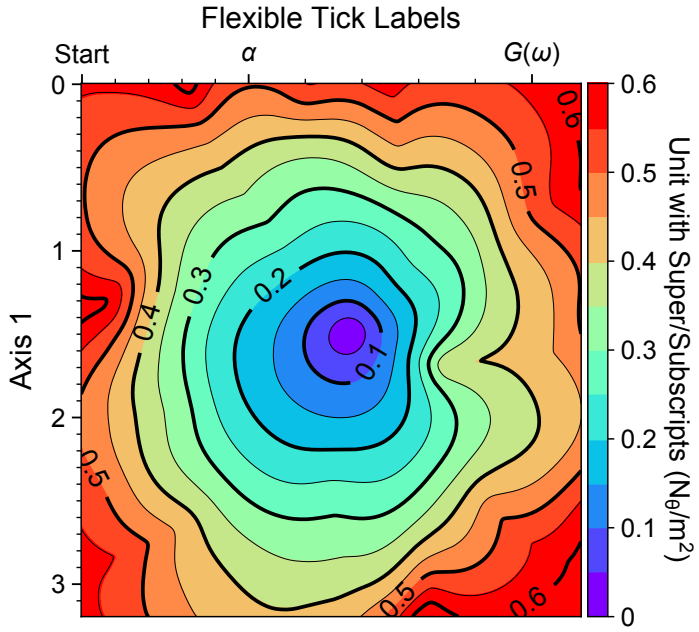
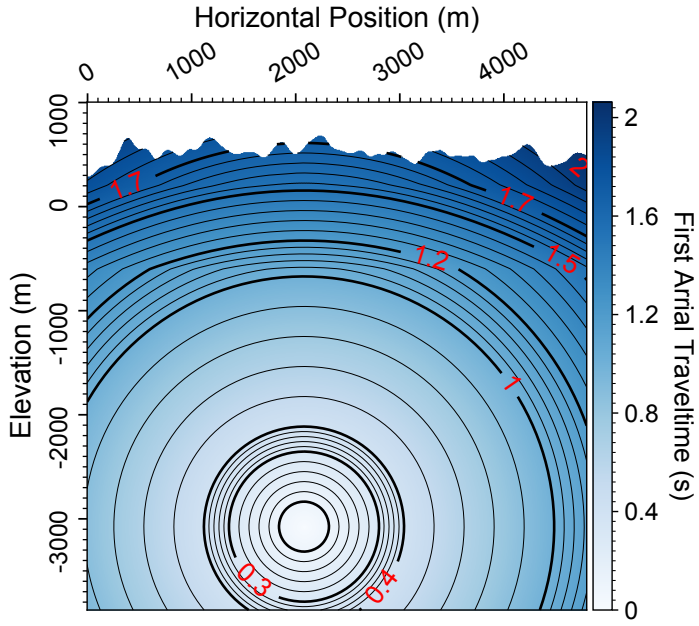


Figure 4: An example of 2D image plot with different colormaps and fonts generated using `showmatrix`. Panels (a)-(f) show the Marmousi-II velocity model using the `jet` (jet, the default colormap), `gist_ncar`, `bwr` (blue-white-red), `binary` (black-white) colormap, `viridis`, and `rainbow256`, respectively, and with `arial` (Arial, the default font), `consolas` (Consolas), `times` (Times New Roman), `courier` (Courier Prime), `plex` (IBM Plex), and `helvetica` (Helvetica) font, respectively. All colormaps and fonts are available in the other functions of the plotting package `pymplot`.



(a)



(b)

Figure 5: An example of 2D contour plot without (a) and with (b) surface topography generated using `showcontour` for a first-arrival traveltime dataset. In Panel (a), the contours have a uniform major contour interval of 0.1 s with one minor contour. In Panel (b), the contours have non-uniform major contour intervals based on a series of given values, and there are three minor contours with an equal spacing between every two adjacent major contours.

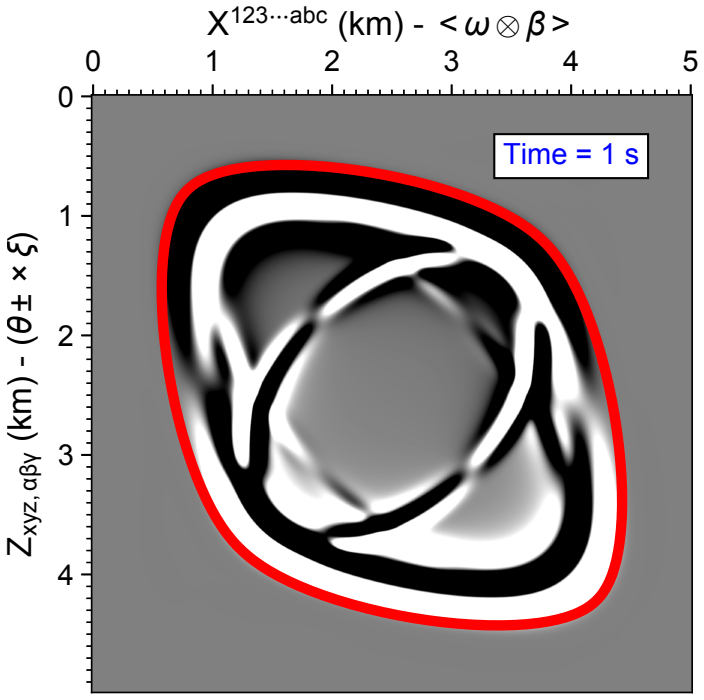
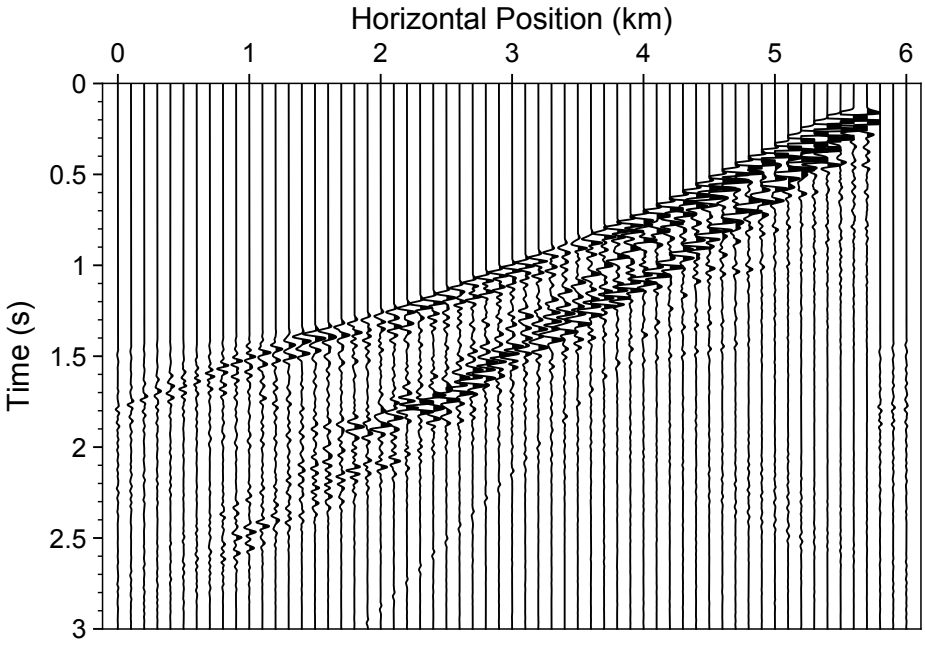
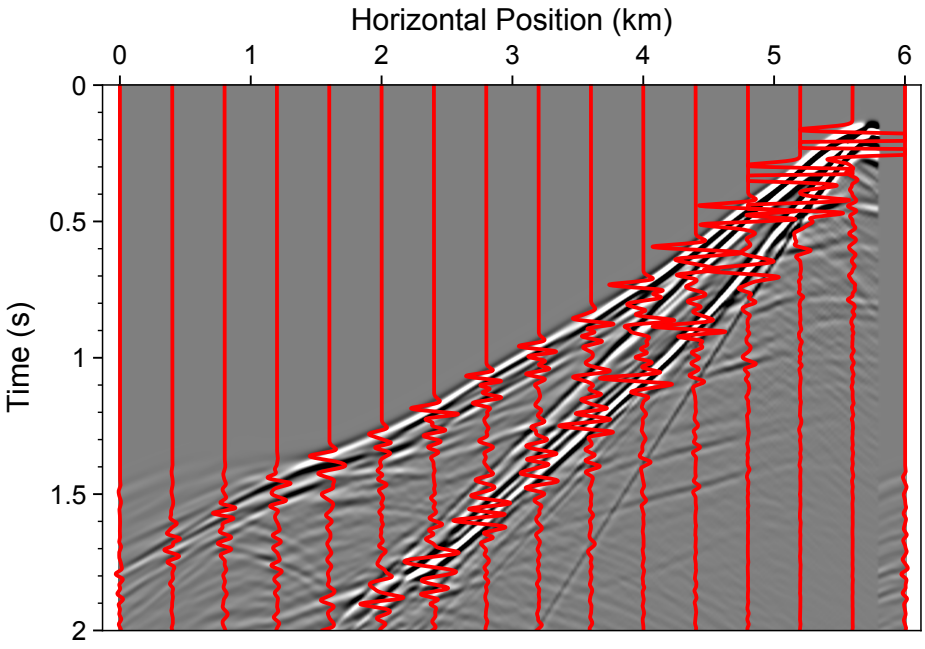


Figure 6: An example of a 2D contour superimposed on a 2D image generated using `showcontour`. The contour is a first-arrival traveltime solved using the anisotropic eikonal equation, while the background image is a wavefield snapshot obtained using the finite-difference anisotropic elastic wave equation. We display only one contour of the traveltime field (the red curve). We also use this plot to demonstrate the capability of our plotting package in handling complex axis labels with Greek letters, subscripts, superscripts, and LaTeX mathematical symbols in a consistent manner with a uniform font typeface.



(a)



(b)

Figure 7: (a) An example of wiggle plot generated using `showwiggle` with option `-fill=1` to fill the wiggle waveforms with positive values. Option `-fill=-1` fills the wiggle waveforms with negative values. (b) An example of wiggle plot generated using `showwiggle` superimposed onto an background image, a different visualization form of the same dataset, to show the correspondence between the image and the waveforms. In this plot, the wiggles are not filled (option `-fill=0`).

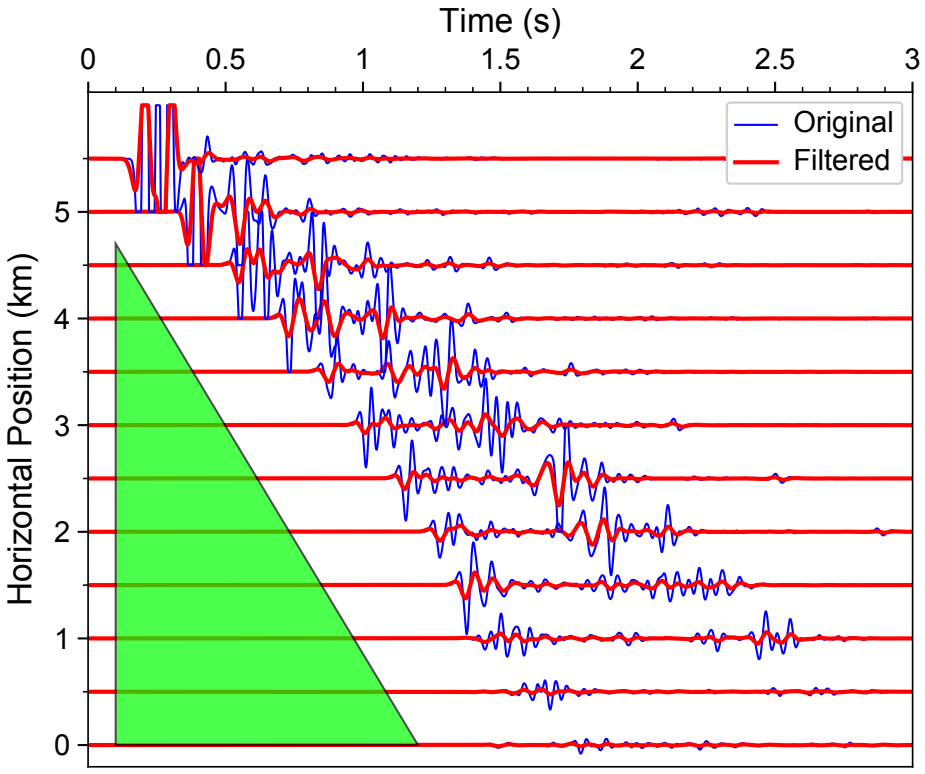


Figure 8: An example of wiggle plot generated by `showwiggle` for two datasets, represented by blue and red wiggles, respectively, with option `-wigglecolor=b,r`. The two wiggle plots have different wiggle line widths, specified with option `-wigglewidth=1,2`. The green triangle annotation in this plot is for demonstration of the annotation versatility of our plotting package.

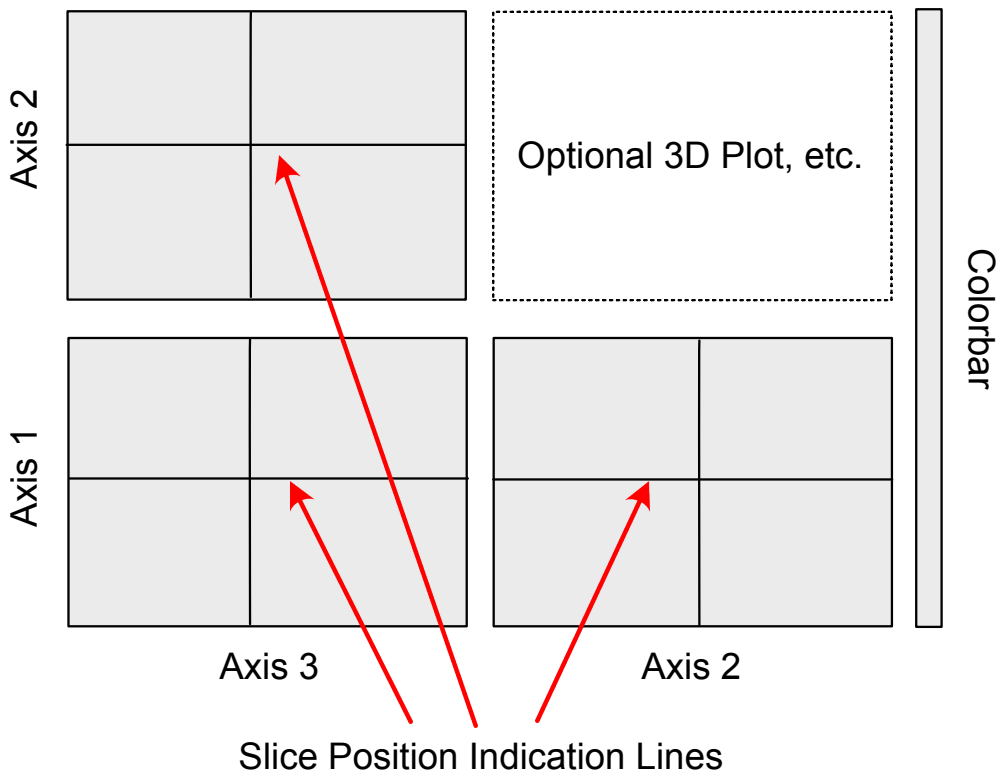


Figure 9: An illustration of the plot layout in the plotting functions `showslice` and `showslicon`.

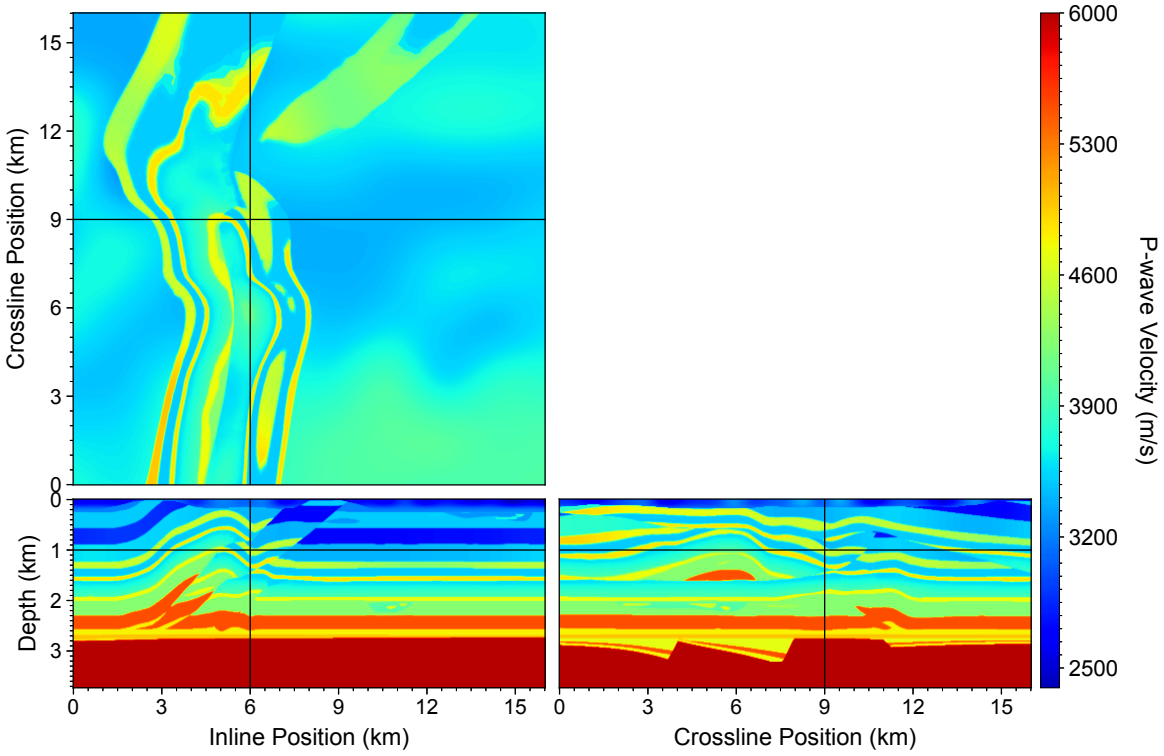


Figure 10: An example of the slice view of the overthrust model generated using `showslice`. The colorbar can be optionally placed on the left, right, top, or bottom of the plot. By default, it is placed on the right of the plot with a height matching the height of the entire image.

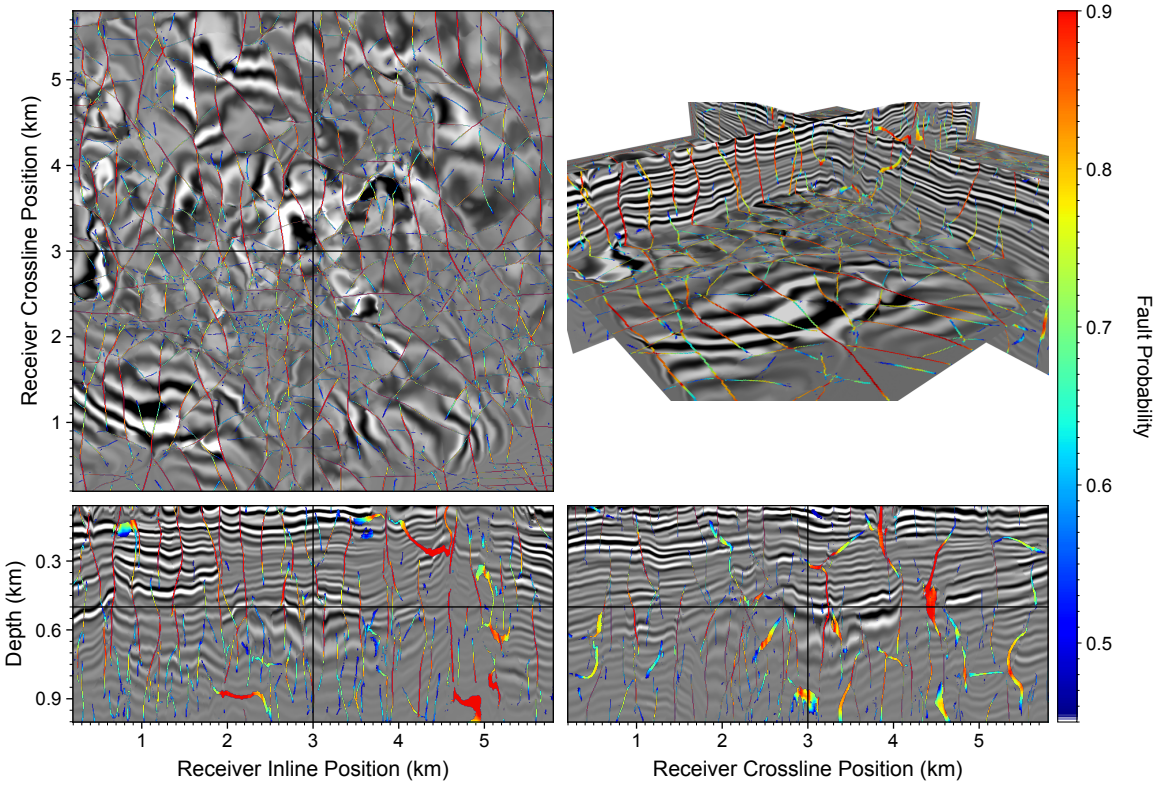


Figure 11: An example of the slice view of an subsurface structural image generated using `showslice`. We plot two datasets in this plot. The data on the top is a fault probability image colored with option `-color=jet`. The bottom is the structural image colored with option `-backcolor=binary`. The top image has a specific opacity setting with option `-alphas=...` to show only the high-probable faults.

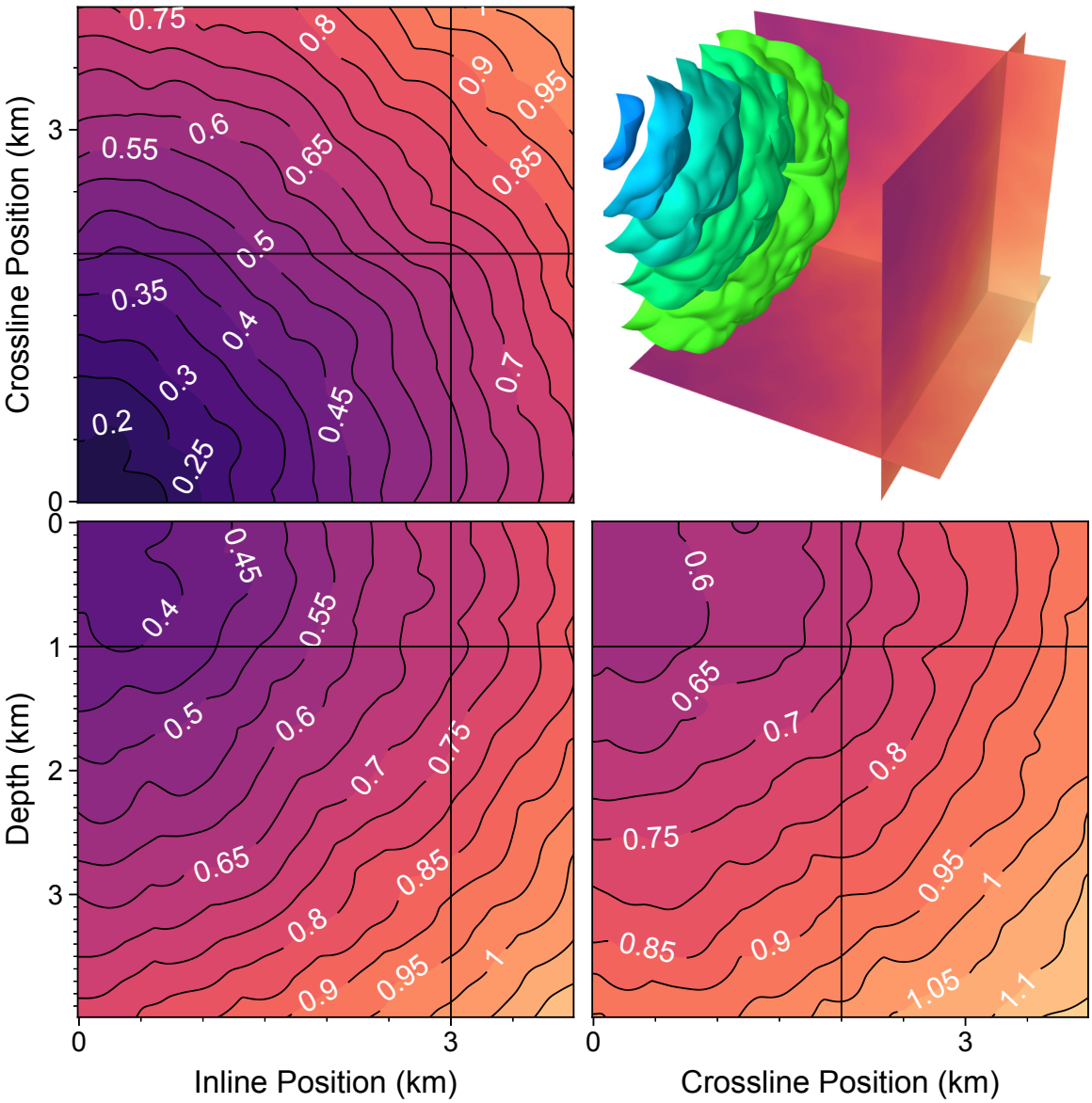


Figure 12: An example of the slice view of traveltime field contours generated using `showslicon`. Similar to that in Figure 11, the top right panel is optional and the image shown in the top right panel is generated using external visualization tool.

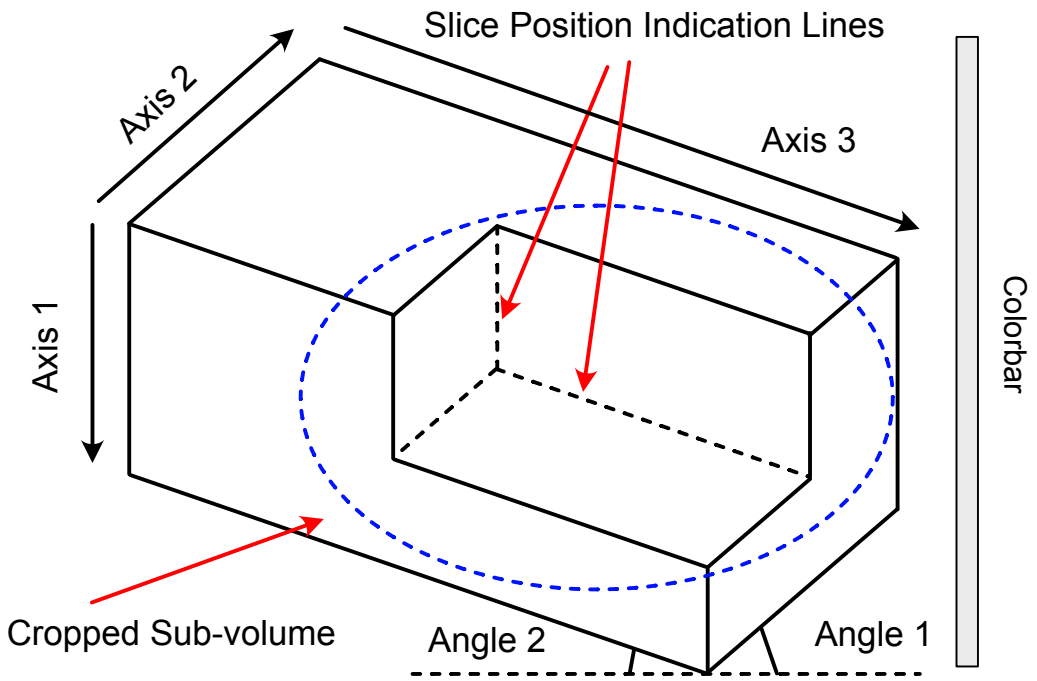
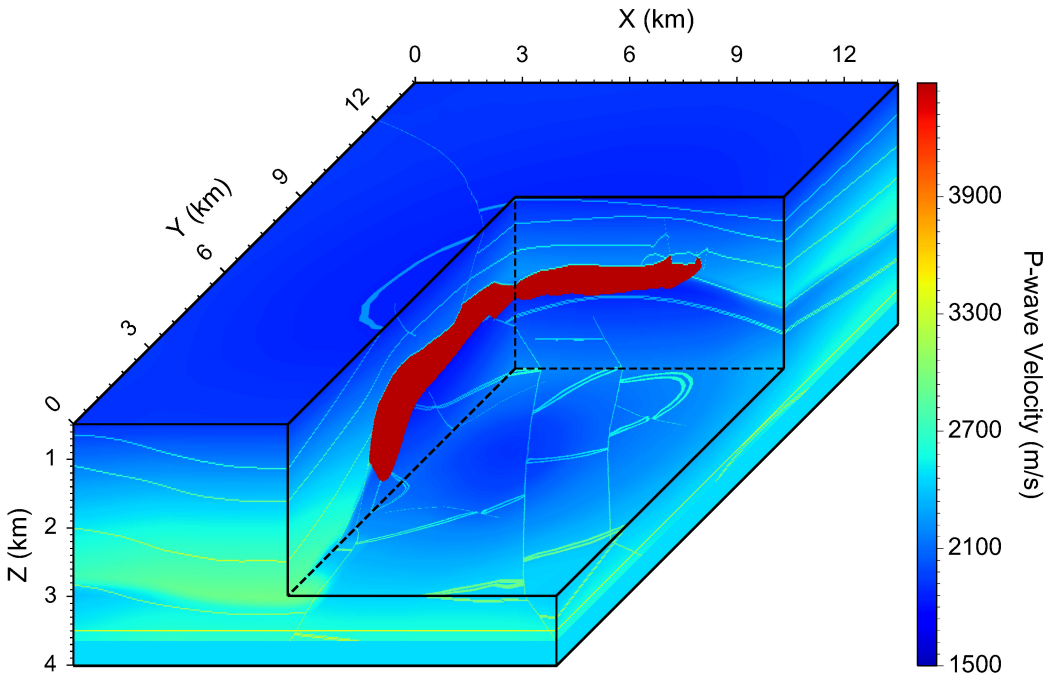
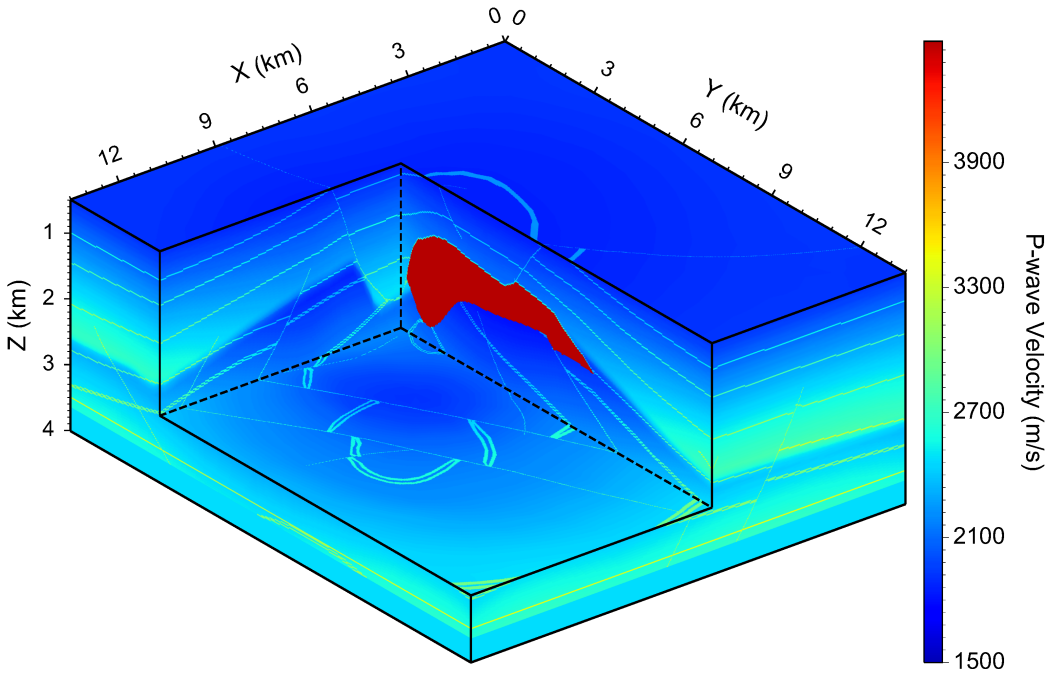


Figure 13: The layout of the plot in the plotting functions `showvolume` and `showvolcon`. The blue dashed ellipse indicates the position of the cropped sub-volume.

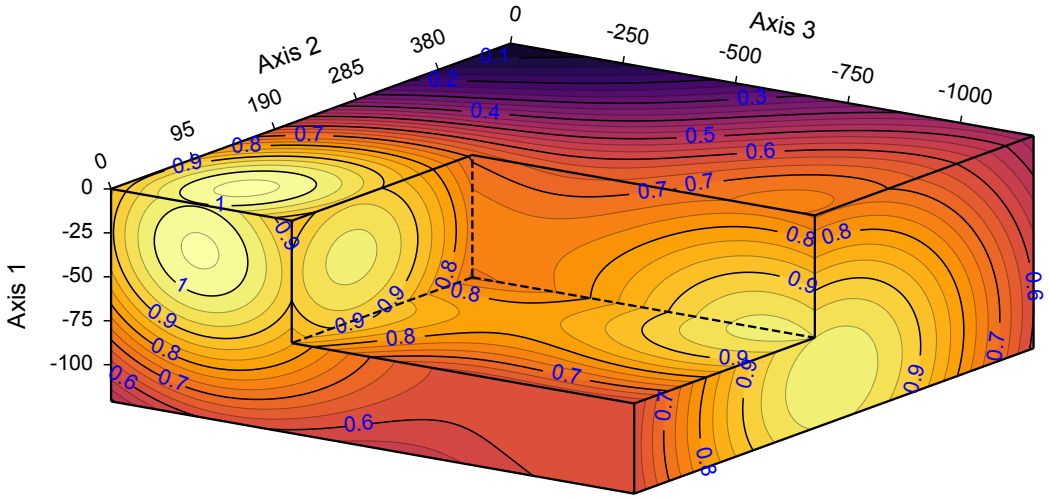


(a)

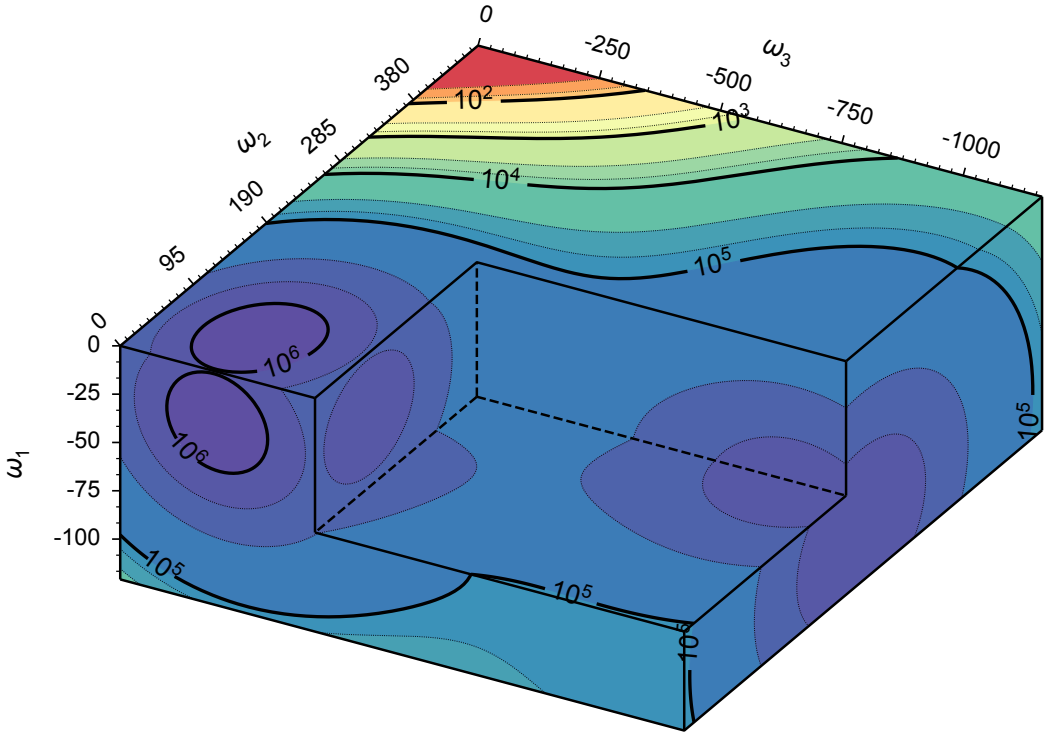


(b)

Figure 14: An example of the 3D volume view of the SEG/EGAE salt model generated using `showvolume` at two different view angles and for different slicing positions. In Panel (b), we set non-zero values for the two angles in option `-angle=angle1,angle2`. By default, `angle2=0`, as in Panel (a).



(a)



(b)

Figure 15: An example of the 3D volume view of the contours of two datasets generated using `showvolcon`. Panel (a) shows a set of contours of equal spacing in the linear scale, while Panel (b) displays a set of contours in the logarithmic scale. The two plots have different values of `-angle=angle1,angle2`. Both panels contain two equal-spaced minor contours between any two adjacent major contours in their respective scale.