Manual of plotastrodata

Yusuke Aso

1 Read Data

The data class AstroData can take a fits file.

```
from plotastrodata.analysis_utils import AstroData

d = AstroData(fitsimage='file_name.fits')
```

AstroData can take more arguments, such as `Tb` and `sigma`. `Tb=True` means the data values will be converted from flux densities in the unit of Jy beam^{-1} to brightness temperatures in the unit of K. `sigma` specifies how to measure the noise level of the data values: for example, the default option of 'hist' means to use the histgram of the data values. The argument of `pvpa` is the position angle of the PV cut direction in the unit of degree, which will be used to calculate the spatial resolution of the PV diagram.

The data class AstroFrame is necessary to form the AstroData instance in a useful format. AstroFrame can take quantities related to coordinate ranges.

```
from plotastrodata.analysis_utils import AstroFrame

f = AstroFrame(vmin=-5.0, vmax=5.0, vsys=2.0,
               center='00h00m00s 00d00m00s', rmax=3.0)
```

`vmin`, `vmax`, and `vsys` are in the unit of km s^{-1}. `rmax` is in the unit of arcsec. Instead of `rmax`, other arguments (`xmin`, `xmax`, `ymin`, `ymax`, `xoff`, `yoff`) can be used to adjust the x and y ranges in more detail. When an argument of `fitsimage` is given, the central coordinates `center` is read from the given fits file; the rest frequency is also read from the fits file, which is used for the frequency-velocity conversion. Moreover, an argument of `dist` can specifies the distance used to change the unit of spatial coordinates from arcsec to au. An argument of `swapxy` is used to swap the x and y coordinates. An argument of `pv` must be `True` when the AstroData instance to be read is a position-velocity (PV) diagram, i.e., the first and second axes are the spatial and velocity coordinates. When `quadrants=True`, the first and third (or second and fourth) quadrants of the PV diagram will be averaged. `xflip` and `yflip` will be used to determine the plotting direction of the x and y axes, respectively; these have a meaning only when a figure is made from the data set.

Forming the AstroData instance needs the following command.

```
f.read(d)
```

After this command, the AstroData instance has useful attributes in the format of numpy array: a 1D array of `d.x` (as well as `d.y` and `d.v`), a 2D or 3D array of `d.data`, a 1D array of `d.beam`, and a float of `d.sigma`. `d.x` and `d.y` are the relative coordinates in the unit of arcsec from the given center. Similarly, d.v is the relative coordinate in the unit of km s^{-1} from the given vsys. d.dx, d.dy, and d.dv are d.x[1]-d.x[0], d.y[1]-d.y[0], and d.v[1]-d.v[0], respectively. d.beam is the beam components array([bmaj, bmin, bpa]), where bmaj and bmin are in the unit of arcsec, while bpa is in the unit of degree. When Tb=True above, the data values are converted to the brightness temperature and stored as d.data. d.sigma is the noise level measured in the way specified above in the same unit as d.data. AstroData can take two more arguments, restfreq and cfactor. restfreq is used to explicitly set the rest frequency in the unit of Hz, which is used to convert frequency to velocity and Jy beam^{-1} to K. When this argument is zero (restfreq=0), the frequency axis is not converted to velocity. When the unit of the frequency/velocity axis in the input fits header is m/s, f.read(d) divides the coordinate by 1000. The argument cfactor specifies a constant factor that d.data is multiplied by. This will be useful when one wants to change the intensity unit from Jy beam^{-1} to mJy beam^{-1}.

In addition to the input attributes, f.read(d) adds three attributes to the AstroData instance d: fitsimage_org, sigma_org, and fitsheader. fitsimage_org saves the input fitsimage, while fitsimage is updated to None after f.read(d). Similarly, sigma_org saves the input sigma, which may be a string, while sigma is updated to the calculated value. fitsheader is the fits header in the format of astropy.io.fits.open(fitsimage)[0].

When pv=True in AstroFrame, the read method changes the beam attribute of the AstroData instance. The first element will be the velocity resolution. The second element of beam will be $1/\sqrt{\cos^2(\mathrm{bpa}-\mathrm{pvpa})/\mathrm{bmaj}^2+\sin^2(\mathrm{bpa}-\mathrm{pvpa})/\mathrm{bmin}^2}$. This is calculated from the intersection of the beam ellipse and the PV cut line. The third element will be 0; thus, the velocity resolution (first element) is regarded as the vertical length by default when this "beam" is plotted in a PV diagram. The original beam will be stored in beam_org.

2 Analyze Data

AstroData also has handy methods to analyze the 2D/3D data.

The binning method rebins the 2D/3D data with a given width for each coordinate.

```
d.binning(width=[5, 4, 2])
```

This command takes an average over 5 channels in the velocity direction, 4 pixels in the y direction, and 2 pixels in the x direction. The number of channels and pixels are decreased accordingly after this method. If the length of width is 2, the two values are regarded as the widths for the y and x directions.

The centering method sets the coordinates so that the spatial center and the systemic velocity have the exact zero coordinates by interpolation.

```
d.centering(includexy=True, includev=False)
```

This command adjusts the x and y coordinates but does not adjust the velocity coordinate. This method will be useful when one wants to quickly get a radial profile along a line passing the center or a PV diagram (using the rotate method together).

The circularbeam method makes the beam shape circular by additional 2D Gaussian convolution. This method takes no argument.

```
d.circularbeam()
```

The new beam has the major and minor axes same as the old major axis. This method also updates the attribute of `d.beam` accordingly.

The `deproject` method deprojects the 2D/3D data with a given position angle (P.A.) and inclination angle.

```
d.deproject(pa=45, incl=45)
```

`pa` is the position angle from the north to the east in the unit of degree. `incl` is the inclination angle; `incl=0` means the face-on configuration and thus no deprojection. This command replaces `d.data` and `d.beam` with the deprojected data and the deprojected beam, respectively.

The `fit2d` method performs MCMC fitting to the 2D image of `d.data` or `d.data[chan]`, where `chan` is the channel number, by using emcee or ptemcee through the class of `plotastrodata.fitting_utils.EmceeCorner`.

```
res = d.fit2d(chan=12, model=model, bounds=bounds,
              kwargs_fit={'nwalkersperdim': 2, 'nsteps': 1000, 'nburnin': 500},
              kwargs_plotcorner={'savefig': 'corner.png'})
```

`model` is a function with the shape of `model(par, x, y)`, where `par` is the list of parameters. `bounds` is the 2D list of the parameter boundary in the shape of $[[p_{0,\mathrm{min}}, p_{0,\mathrm{max}}], [p_{1,\mathrm{min}}, p_{1,\mathrm{max}}], ...]$. `kwargs_fit` and `kwargs_plotcorner` are the arguments for the methods of `EmceeCorner.fit` and `EmceeCorner.plotcorner`, respectively. `nsteps` includes the number the burn-in steps. The output `res` is a dictionary consisting of `popt`, `plow`, `pmid`, `phigh`, `model`, and `residual`. The first four keys provide the best, lower percentile, 50 percentile, and higher percentile parameters. The last two keys provide the model and residual 2D images.

The `histogram` method returns the bins and the histogram in the bins of the attribute of `d.data` by using numpy.histogram().

```
hbin, hist = d.histogram(bins=10)
```

The arguments are the same as those for numpy.histogram(). The bins (`hbin`) have the same length as the histogram (`hist`), which is different from the original numpy.histogram.

The `gaussfit2d` method performs the 2D Gaussian fitting to the 2D image of `d.data` or `d.data[chan]`, where `chan` is the channel number, by using `scipy.optimize.curve_fit`.

```
res = d.gaussfit2d(chan=12)
```

This command performs the fitting to the channel of 12 in `d.data`. For 2D data, the `chan` argument can be omitted. The output (`res` here) is a dictionary having keys of `popt`, `pcov`, `model`, `residual`, and `center`. `popt` and `pcov` are the optimized parameters (peak intensity, central x, central y, major FWHM, minor FWHM, and P.A.) and their covariance. The `model` and `residual` are 2D arrays with the same shape of the fitted 2D array. `center` is the best-fit center in the format of "hms dms" string. This central coordinates are calculated from `d.center` and the best-fit central x and y through `xy2coord`.

The `mask` method puts numpy.nan on pixels that satisfies a given condition.

```
d.mask(dataformask=d2.data, includepix=[1e-3, 100], excludepix=[50, 200])
```

This command puts numpy.nan on pixels of `d.data` where `d2.data` is outside [1e-3, 100] or inside [50, 200], where `d2` is an AstroData instance different from `d`. This method will be useful when one wants to put a mask on a moment 1 map using the values of a moment 0 map.

The `profile` method makes line profiles at given spatial positions.

```
v, f, g = d.profile(xlist=[-0.5, 0.2], ylist=[1, 0.8], ellipse=[0.3, 0.2, 45])
```

This command makes line profiles at (x,y)=(-0.5, 1) and (0.2, 0.8) in the unit of arcsec. The intensity of each profile is an average over a boxcar ellipse with the major axis of 0.3 arcsec, the minor axis of 0.2 arcsec, and the P.A. of 45 degrees. Instead of `xlist` and `ylist`, coordinate strings can be input using an argument of `coords`: `coords=['00h00m00.0s 00d00m00.0s', '11h11m11.1s 11d11m11.1s']`. When `ellipse` is omitted, the profile is made by picking up the values at the closest pixel to the given position. When the ellipse size is not so large compared to the pixel size, the integer argument `ninterp` may be useful; this makes the pixel size `ninterp` times finer by interpolartion. When the boolian argument `flux` is True, the output profile has the unit of Jy. Additionally, when the boolian argument `gaussfit` is True, the profiles will be fitted with a 1D Gaussian function. The output `v` above is `d.v`. The output `f` above is a list of 1D-array profiles, i.e., 2D array. The output `g` is a list of dictionaries; each dictionary has keys of `best` and `error` (square root of the diagonal components of the covarience).

The `rotate` method rotates the attribute `d.data` by the given angle in the unit of degree by interpolation.

```
d.rotate(pa=45)
```

When a pixel refers to a position outside the original image, this pixel has numpy.nan after this method.

The `slice` method makes a radial profile for a 2D image or a PV diagram for a 3D image along a given direction and length by interpolation.

```
r, f = d.slice(length=3, pa=45, dx=0.2)
```

This commands makes a radial profile (or PV diagram) along a cut with a length of 3 arcsec at P.A.=45 degrees with a separation of 0.2 arcsec. The output `r` and `f` are both 1D arrays of the positional offsets and the intensity at the positions. When the separation `dx` is omitted, the absolute value of the x pixel size is adopted.

The `todict` method returns the attributes as a dictionary.

```
a = d.todict()
```

This output includes keys of `data`, `x`, `y`, `v`, `fitsimage`, `beam`, `Tb`, `restfreq`, `cfactor`, `sigma`, and `center`. This dictionary can be input to methods of PlotAstroData as `**a`.

The `writetofits` method exports the instance as a fits file.

```
d.writetofits(fitsimage='new_file_name.fits')
```

The output fits file reuses the header components of the fits file used to make the AstroData instance (`'file_name.fits'` above); some header components are automatically updated after the above-mentioned methods for analysis, such as `CDELT1`.

3 Plot Data

The class PlotAstroData can take an AstroData instance through the method of `d.todict()`.

```
from plotastrodata.plot_utils import PlotAstroData

p = PlotAstroData(rmax=3.0)
p.add_color(**d.todict())
p.add_scalebar(length=50 / 140, label='50 au')
p.set_axis()
p.savefig('figure_name.png')
```

These commands make a color map using the AstroData instance `d`. PlotAstroData can take the same arguments as AstroFrame to define the plotting ranges; particularly `rmax` (also, `vmin` and `vmax` for a cube or a PV diagram) is necessary. The method `p.add_color()` can take a fits file directly instead of the AstroData instance, as `p.add_color(fitsimage='file_name.fits')`. This method can actually take the same arguments as AstroData to specify the data to be plotted as a color map; `**d.todict()` does this indirectly. The command `p.add_scalebar()` can be omitted if the map does not need to show a scale bar. The command `p.set_axis()` (or `p.set_axis_radec()`) is necessary even without any argument.

The class PlotAstroData can take more arguments. `veldigit` specifies the number of digits of the velocity label on the channel maps. `channelnumber` specifies which channel is used to make a 2D image; when this is None (default), channel maps are made instead of a 2D image. `nrows` and `ncols` specify the grid shape of the channel maps. `fontsize` specifies the basic font size in the figure. `nancolor` specifies which color is used when the value is nan. `dpi` specifies the resolution of a raster image file. `figsize` is the same as the argument for Figure.figure of matplotlib. `fig` and `ax` can be used to input external Figure and Axes objects. More detailed usage can be found in the `example.py` file and `https://plotastrodata.readthedocs.io/en/latest/#`. The following is the explanation of each method of PlotAstroData.

3.1 Maps

The `add_color` method plots the given data in the format of a color map.

```
p.add_color(**d.todict(), stretch='log', show_cbar=True, cblabel=r'mJy beam$^{-1}$',
            cbticks=[0.01, 0.03, 0.1, 0.3], cbticklabels=['10', '30', '100', '300'],
            cblocation='right')
```

The `stretch` argument can be 'linear', 'log', 'asinh', or 'power'. The Arcsin hyperbolic stretch can be adjusted by another argument of `stretchscale` as asinh(data / stretchscale). The power-law stretch can be adjusted by another argument of `stretchpower` as ((data / vmin)^(1 - stretchpower) - 1) / (1 - stretchpower) / ln(10), where vmin is a given minimum value. The arguments starting with 'cb' adjust the colorbar. In addition, two arguments, `xskip` and `yskip`, can be used to skip spatial pixels in this method as well as `add_contour`, `add_segment`, and `add_rgb`. Similarly, `show_beam`, `beamcolor`, and `beampos` can be used in these four methods to adjust beam color and position, as well as switch whether to show the beam.

The `add_contour` method plots the given data in the format of a contour map.

```
p.add_contour(**d.todict(), levels=[-3, 3, 6, 9])
```

The `levels` argument spedifies the contour levels in the unit of `d.sigma`.

The `add_rgb` method plots the given data in the format of three-color maps. The input three data sets are mixed as Red, Green, and Blue.

```
p.add_rgb(**d.todict(), stretch=['linear', 'log', 'linear'])
```

For this method, the input is a combination of three data sets, and thus `d.data` here must be a 1D list (not a numpy array) of three 2D/3D numpy arrays. In the same way as `add_color`, `stretchscale` and `stretchpower` are available in this method.

The `add_segment` method plots the given data in the format of a segment map, as is often used for polarization maps.

```
p.add_segment(Qfits='Qfile_name.fits', Ufits='Ufits_name.fits',
              ampfactor=3.8, angonly=False, rotation=90, cutoff=3.0)
```

The input format of the data is different from that for `add_color` and `add_contour`. One of the following pairs must be given: (`ampfits`, `angfits`), (`Qfits`, `Ufits`), (`amp`, `ang`), or (`stQ`, `stU`). The latter two pairs are supposed to be in the numpy.array format. The `ampfactor` argument can be used to adjust the segment length. When `angonly=True`, the segment length is set to be uniform. The `rotation` can be used to rotate the segments in the unit of degree. The `cutoff` argument specifies the intensity threshold to calculate the amplitude and angle from the Stokes Q and U intensities.

3.2 Patches, markers, text, lines, arrows, and scale bar

The class PlotAstroData also has methods for plotting regions (patches), markers, text, lines, and arrows. The following is the explanation of each method for this purpose.

The `add_region` method overlays an ellipse or a rectangular. the given data in the format of a segment map, as is often used for polarization maps.

```
p.add_region(patch='ellipse',
             poslist=[[0.5, 0.5], [0.1, 0.2]],
             majlist=[0.3, 1.0],
             minlist=[0.2, 0.8],
             palist=[0, 90])
```

The argument `patch` may be 'ellipse' or 'rectangle'. The patch position can be specified by a 2D list like [0.1, 0.2] or a coordinate string like '01h23m45.6s 01d23m45.6s'. The list means a fractional position from left to right and bottom to top: e.g., [0, 0], [0.5, 0.5], and [1, 1] are bottom left, center, and top right, respectively. `majlist`, `minlist`, and `palist` specifies the patch size in the unit of arcsec and the orientation in the unit of degree. Other arguments for matplotlib.patches.Ellipse or matplotlib.patches.Rectangle can be used in this method. Another method `add_beam` uses this method and is executed internally in `add_color`, `add_contour`, `add_segment`, `add_rgb`.

The `add_marker` method overlays markers.

```
p.add_marker(poslist=[[0.5, 0.5], [0.1, 0.2]])
```

Except for the position list `poslist` (same as in `add_region`), the marker properties can be specified by the arguments for Axes.plot of matplotlib.

The `add_text` method overlays text.

```
p.add_marker(poslist=[[0.5, 0.5], [0.1, 0.2]],
             slist=['abs', 'hoge'])
```

Except for the position list `poslist` (same as in `add_region`) and the list of text `slist`, the text properties can be specified by the arguments for Axes.text of matplotlib.

The `add_line` method overlays lines.

```
p.add_marker(poslist=[[0.5, 0.5], [0.1, 0.2]],
             anglelist=[0, 90],
             rlist=[0.4, 1])
```

The position list `poslist` (same as in `add_region`) specifies the start position of the line. `anglelist` and `rlist` specify the position angle in the unit of degree and the length in the unit of arcsec of the line, respectively. Other arguments for Axes.plot of matplotlib can be used in this method.

The `add_arrow` method overlays arrows.

```
p.add_marker(poslist=[[0.5, 0.5], [0.1, 0.2]],
             anglelist=[0, 90],
             rlist=[0.4, 1])
```

`poslist`, `anglelist`, and `rlist` are the same in `add_line`. Other arguments for Axes.quiver of matplotlib can be used in this method.

The `add_scalebar` method overlays a scale bar.

```
p.add_scalebar(length=0.5,
               label='70 au',
               barpos=[0.8, 0.1])
```

The bar length `length` is in the unit of arcsec. The bar label `label` is located above the bar. `barpos` is a fractional position from left to right and from bottom to top (same as `poslist`). In addition, the arguments of `color`, `fontsize`, `linewidth`, and `bbox` can be used in this method. `bbox` is a dictionary argument for Axes.text of matplotlib.

3.3 Setting axes and saving figures

The method `set_axis` (or `set_axis_radec`) is necessary even without any argument. This method provides axes of relative coordinates and can also adjust things related to the axes.

```
p.set_axis(title='figure title')
```

The `title` argument can be either string or dictionary; the latter is a set of arguments for Axes.set_title or Figure.suptitle of matplotlib. This method can also take more arguments related to Axes.set_* of matplotlib: `xscale`, `yscale`, `xlim`, `ylim`, `xlabel`, `ylabel`, `xticks`, `yticks`, `xticklabels`, and `yticklabels` are in the same format as in Axes.set_* of matplotlib. `xticksminor` and `yticksminor` are either list or integer; the latter means how many times more ticks are needed than the major ticks. Two more dictionary arguments, `grid` and `aspect`, available as for Axes.grid and Axes.set_aspect of matplotlib.

The method `set_axis_radec` provides axes in the right ascension and declination coordinates.

```
p.set_axis_radec(title='figure title',
                 xlabel='R.A. (ICRS)',
```

```
                 ylabel='Dec. (ICRS)',
                 nticksminor=2,
                 grid=None)
```

`title`, `xlabel`, `ylabel`, and `grid` are the same as in `set_axis`. `nticksminor` specifies how many times more ticks are needed than the major ticks.

The method `savefig` saves the figure as an image file and/or shows it on the screen.

```
p.savefig(filename='hoge.png',
          show=True)
```

In addition to the two arguments, the arguments for Figure.savefig of matplotlib can be used for this method.

The method `get_figax` returns the Figure and Axes objects.

```
fig, ax = p.get_figax()
```

Figures 1 and 2 show examples obtained through the `plotastrodata.plot_utils` module. They might appear to be messy, but this is because these examples aim to show various features of this module. The module enables you to fine-tune the figures in various aspects.

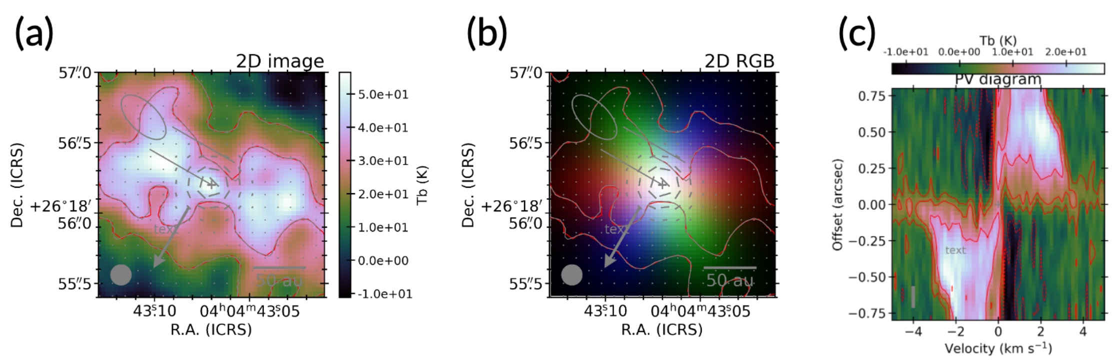

Figure 1: Examples of 2D images. In addition to the color and RGB maps, scale bars, and beams, panels (a) and (b) show contour maps, segment maps, regions, markers, lines, arrows, and text. Panel (c) is a position-velocity diagram (color and contour maps) with a marker and text.

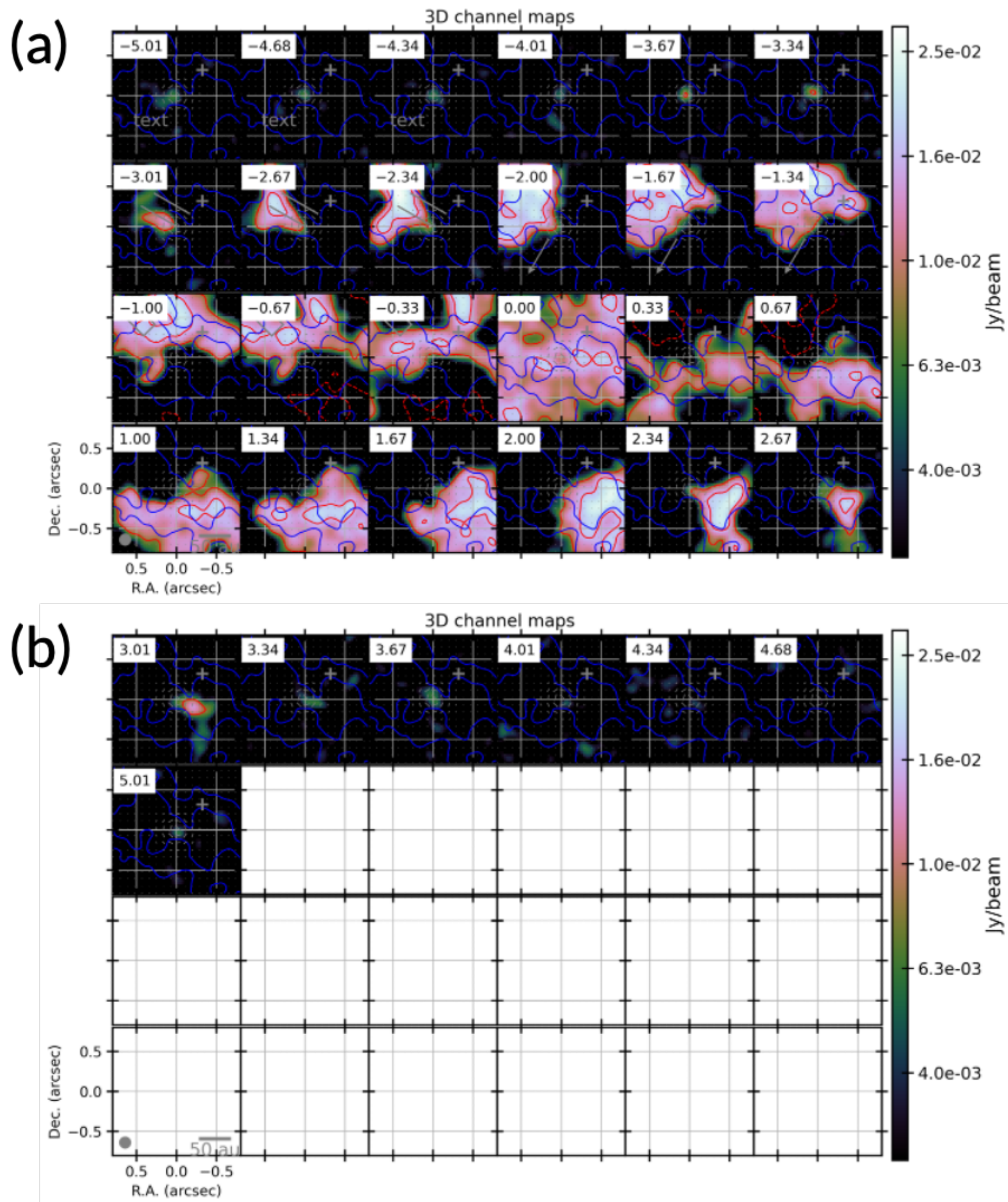

Figure 2: Example of channel maps. In addition to the color maps, scale bars, and beams, panels (a) and (b) show contour maps, segment maps, regions, markers, lines, arrows, and text. The number at the top right corner at each channel denotes the velocity in the unit km s^{-1}. Panels (a) and (b) are in separated image files.

3.4 Animation

The PlotAstroData class can be used together with `matplotlib.animation` to create an animation file from a fits file. The following is an example using a fits file in the GitHub site of `plotastrodata`.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from plotastrodata.plot_utils import PlotAstroData


def update_plot(i):
    f = PlotAstroData(rmax=0.8, vmin=-5, vmax=5 fitsimage='testFITS/test3D.fits',
                      channelnumber=i, fig=fig)
    f.add_color(fitsimage=pre+'test3D.fits')
    f.set_axis()
    f.fig.tight_layout()

nchans = 61
fig = plt.figure()
ani = animation.FuncAnimation(fig, update_plot, frames=nchans, interval=50)
Writer = animation.writers['ffmpeg']
writer = Writer(fps=10, bitrate=128)
ani.save('test_animation.mp4', writer=writer)
plt.close()
```

3.5 Other plotting functions

The function `plot3d` is a function independent of the PlotAstroData class. This function provides an HTML file to show a 3D figure that can be interactively rotated in a web browser.

```
plot3d(**d.todict(),
       levels=[3, 6, 9],
       cmap='Jet',
       alpha=0.08,
       xlabel='R.A. (arcsec)',
       ylabel='Dec. (arcsec)',
       vlabel='Velocity (km/s)',
       eye_p=0,
       eye_i=180,
       outname='filename',
       show=True)
```

The data to be plotted can be an instance of AstroData. `levels` specifies which surfaces are plotted. `cmap` is an argument (named colorscale) used in the 'mesh3d' type data in plotly.graph_objs. `alpha` is also used through the 'mesh3d' type data in plotly.graph_objs. `eye_p` and `eye_i` specify the viewing direction when the file is opened. The output has the extension of .html with the file name of `outname`. When `show=True`, a web browser will be excecuted automatically to show the 3D figure. Figure 3 shows an example with the test fits file in the GitHub site of `plotastrodata`.

The `plot_utils` module has more functions to make other types of figures. `plotprofile` provides a figure of line profiles and `plotslice` provides a 1D slice of a 2D image. These two functions use `AstroData.profile` and `AstroData.slice`, respectively. Hence, it may be more flexible to make figures in one's preferred way using the outputs of these method than using `plotprofile` and `plotslice`.

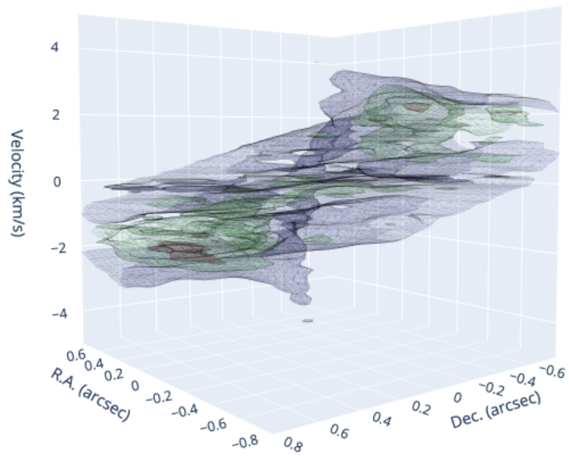

Figure 3: Snapshot of the html file on a web browser, made by `plot_utils.plot3d`. The actual html file can be rotated freely on a web browser.

4 Relative and Absolute Coordinates

The relation between relative and absolute coordinates is not trivial on the plane of the sky. For example, the point 30 degrees (= 2h) east of '00h00m00s 60d00m00s' has coordinates of '03h16m25.6s 48d35m25s', rather than '2h00m00s 60d00m00s'. The module `coord_utils` provides useful functions for the coordinate transformation.

```
from plotastrodata.coord_utils import xy2coord
from plotastrodata.coord_utils import coord2xy
s = xy2coord(xy=[[30, 90], [0, 0]], coordorg='00h00m00s 60d00m00s')
print(s)
#['03h16m25.58528421s +48d35m25.36040662s', '06h00m00s +00d00m00s']
a = coord2xy(coords=s, coordorg='00h00m00s 60d00m00s')
print(a)
#[[30, 90], [0, 0]]
#[x_array, y_array]
```

This example calculates 30 degrees east (and 0 degrees north) and 90 degrees east (and 0 degrees north) of '00h00m00s 60d00m00s' and puts back the resulting coordinates to the relative coordinates in the unit of degree ([offset to the east, offset to the north]). The coordinates may include a frame, like 'ICRS 00h00m00s 60d00m00s' and 'B1950 00h00m00s 60d00m00s'.

5 Rotation

Comparison of a physical model and an observational result often requires 3D rotation between the model coordinates and the observational coordinates. The `los_utils` module provides a useful coordinate transformation. It starts with a parabolic streamer, as an example, on a coordinate set, $(x_{\rm sys}, y_{\rm sys}, z_{\rm sys})$, named system coordinates, as shown in Figure 4(a). The direction of such a streamer is defined with polar and azimuthal angles (θ_0 and ϕ_0, respectively) in a

3D coordinate, $(x_{\rm phy}, y_{\rm phy}, z_{\rm phy})$, named physical coordinates. Figures 4(b) and 4(c) show the relation between $(x_{\rm sys}, y_{\rm sys}, z_{\rm sys})$ and $(x_{\rm phy}, y_{\rm phy}, z_{\rm phy})$, through an intermediate coordinate set, $(x_{\rm azi}, y_{\rm azi}, z_{\rm azi})$, named azimuthal coordinates.

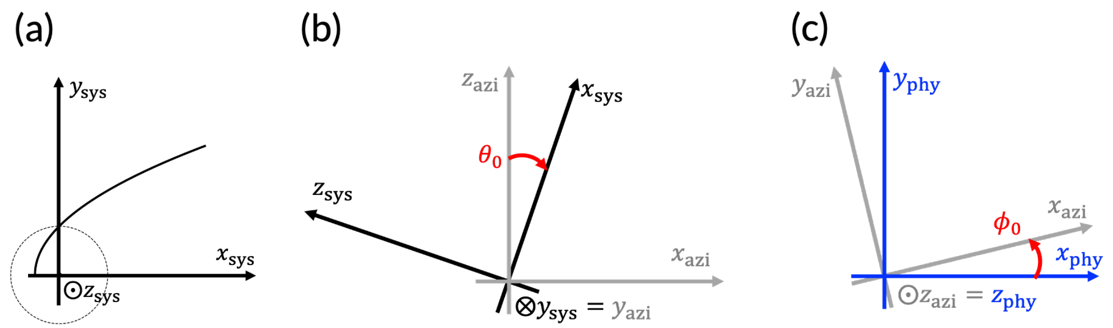

Figure 4: (a) The streamer system. (b) The system and azimuthal coordinates. (c) The azimuthal and physical coordinates.

When a model in the physical coordinates is observed, the reltaion beteen the physical and observational coordinates $(x_{\rm obs}, y_{\rm obs}, z_{\rm obs})$ requires inclination and position (or orientation) angles. Although this relation also requires another azimuthal angle, the ϕ_0 devined above will work for the azimuthal rotation. Figures 5(a) and 5(b) show the relation between $(x_{\rm phy}, y_{\rm phy}, z_{\rm phy})$ and $(x_{\rm obs}, y_{\rm obs}, z_{\rm obs})$, through an intermediate coordinate set, $(x_{\rm ori}, y_{\rm ori}, z_{\rm ori})$, named oriented coordinates. In this definition, when the zero position angle (P.A.= 0) means a configuration where the $z_{\rm phy}$ direction is projected to the $y_{\rm obs}$ direction. In other words, *this P.A. is not that of the disk major axis* but that of the blueshifted outflow in a typical protostellar system.

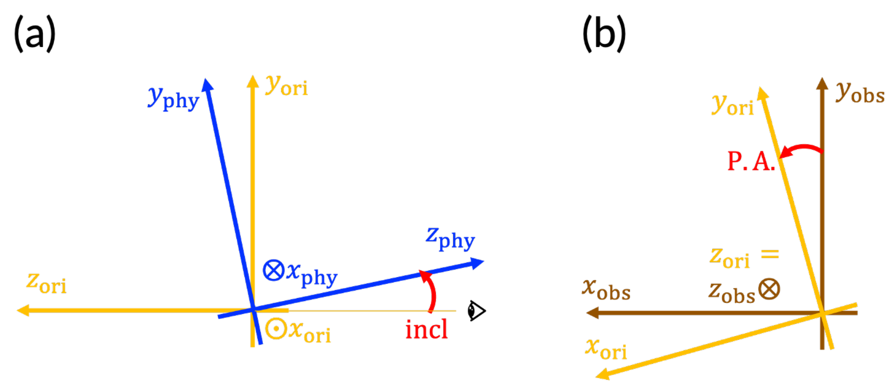

Figure 5: (a) The physical and oriented coordinates. (b) The oriented and observational coordinates.

Figure 6 summarizes the relation between the system and observational coordinates with the four angles (θ_0, ϕ_0, incl., and P.A.) through the physical coordinates. The relation can be expressed

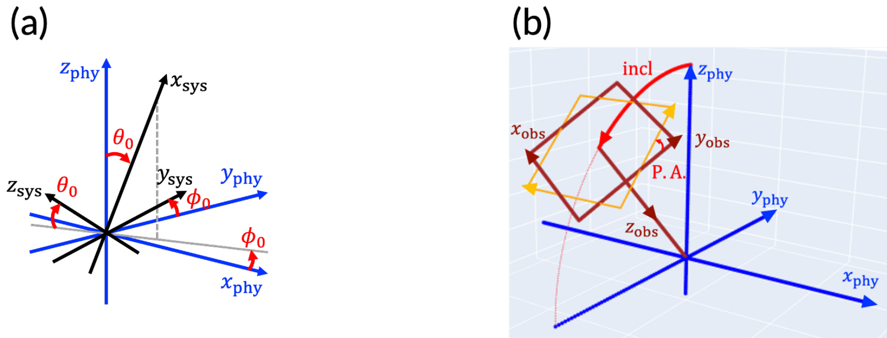

Figure 6: (a) The system and physical coordinates. (b) The physical and observational coordinates.

by the rotational matrices $R_x(\theta)$, $R_y(\theta)$, and $R_z(\theta)$ as follows:

$$\boldsymbol{x}_{\rm sys} = R_y(\frac{\pi}{2} - \theta_0)\boldsymbol{x}_{\rm azi}, \tag{1}$$

$$\boldsymbol{x}_{\rm ori} = R_z(-\phi_0)\boldsymbol{x}_{\rm phy}, \tag{2}$$

$$\boldsymbol{x}_{\rm phy} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix} R_x(-{\rm incl.})\boldsymbol{x}_{\rm ori}, \tag{3}$$

$$\boldsymbol{x}_{\rm ori} = R_z({\rm P.A.})\boldsymbol{x}_{\rm obs}. \tag{4}$$

The transformation between $\boldsymbol{x}_{\rm obs}$ and $\boldsymbol{x}_{\rm sys}$ can be done through the functions `obs2sys` and `sys2obs` in `los_utils`.

```
from plotastrodata.los_utils import obs2sys
from plotastrodata.los_utils import sys2obs
import numpy as np

xobs, yobs, zobs = sys2obs(xsys=1/np.sqrt(2), ysys=1/np.sqrt(2), zsys=0,
pa=0, incl=30, phi0=0, theta0=90, polar=False)
print(xobs, yobs, zobs)
#-0.707107 0.612372 0.353553
#-1/sqrt(2) cos(30deg)/sqrt(2) sin(30deg)/sqrt(2)
xsys, ysys, zsys = obs2sys(xobs=xobs, yobs=yobs, zobs=zobs, pa=0, incl=30,
phi0=0, theta0=90, polar=False)
print(xsys, ysys, zsys)
#0.7071077 0.707107 0.000000
#1/sqrt(2) 1/sqrt(2) 0
```

The input coordinate (e.g., `xsys` or `xobs`) may be a numpy array. When `polar=True`, `xsys`, `ysys`, and `zsys` are the radius, polar angle, and azimuthal angle, respectively; the radius is in the unit of arcsec, and the polar and azimuthal angles are in the unit of radian. `xobs`, `yobs`, and `zobs` are the Cartesian coordinates regardless of the `polar` argument. In particular, the combination of `polar=True` and $\theta_0 = 90$ is useful when a model is defined in the physical coordinates in the polar format.

In addition to the functions for the spatial coordinate transformation, `los_utils` has another function, `polarvel2losvel`, to transform polar velocities (v_r, v_θ, v_ϕ) to line-of-sight velocity (i.e., $v_{z,\mathrm{obs}}$).

```
from plotastrodata.los_utils import polarvel2losvel
import numpy as np

v = polarvel2losvel(v_r=-1/np.sqrt(2), v_theta=0, v_phi=1/np.sqrt(2),
theta=np.pi/2, phi=np.pi/2, incl=30, phi0=0, theta0=90)
print(v)
#-0.353553
#-1/sqrt(2)/2
```

`theta` and `phi` may be numpy arrays and in the unit of radian. `incl`, `phi0`, and `theta0` are the same as in `obs2sys` and `sys2obs`. This velocity transformation is useful for calculating the line-of-sight velocity when the 3D velocity field of a model is written in the polar format as a function of the polar coordinates. This velocity transformation can be derived from the following ideas. First, a vector is expressed as a matrix product of a basis matrix and a component column vector.

$$E_1\boldsymbol{x}_1 := \begin{pmatrix}\boldsymbol{e}_{x,1} & \boldsymbol{e}_{y,1} & \boldsymbol{e}_{z,1}\end{pmatrix}\begin{pmatrix}x_1\\ y_1\\ z_1\end{pmatrix} = \begin{pmatrix}\boldsymbol{e}_{x,2} & \boldsymbol{e}_{y,2} & \boldsymbol{e}_{z,2}\end{pmatrix}\begin{pmatrix}x_2\\ y_2\\ z_2\end{pmatrix} =: E_2\boldsymbol{x}_2 \tag{5}$$

$$\boldsymbol{x}_1 = E_1^{-1}E_2\boldsymbol{x}_2 \tag{6}$$

This provides the change-of-basis matrix. This matrix can also change the basis for velocity vectors.

$$\boldsymbol{v}_1 = E_1^{-1}E_2\boldsymbol{v}_2 \tag{7}$$

Hence, we already know the relation between $\boldsymbol{v}_{\mathrm{ori}}$ and $\boldsymbol{v}_{\mathrm{sys}}$.

$$\boldsymbol{v}_{\mathrm{ori}} = R_x(\mathrm{incl.})\begin{pmatrix}-1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & -1\end{pmatrix}R_z(\phi_9)R_y(\theta_0 - \frac{\pi}{2})\boldsymbol{v}_{\mathrm{sys}} \tag{8}$$

The polar velocities can be expressed with the Cartesian velocities as follows:

$$\begin{pmatrix}v_r\\ v_\theta\\ v_\phi\end{pmatrix} = \begin{pmatrix}\sin\theta\cos\phi & \sin\theta\sin\phi & \cos\theta\\ \cos\theta\cos\phi & \cos\theta\sin\phi & -\sin\theta\\ -\sin\phi & \cos\phi & 0\end{pmatrix}\begin{pmatrix}v_x\\ v_y\\ v_z\end{pmatrix} =: E_r^{-1}\boldsymbol{v}. \tag{9}$$

Then, the polar velocities are linked to $\boldsymbol{v}_{\mathrm{sys}}$.

$$\boldsymbol{v}_{\mathrm{sys}} = E_r\begin{pmatrix}v_{r,\mathrm{sys}}\\ v_{\theta,\mathrm{sys}}\\ v_{\phi,\mathrm{sys}}\end{pmatrix} = (E_r^{-1})^T\begin{pmatrix}v_{r,\mathrm{sys}}\\ v_{\theta,\mathrm{sys}}\\ v_{\phi,\mathrm{sys}}\end{pmatrix} \tag{10}$$

Finally, the line-of-sight velocity is calculated as the z component of $\boldsymbol{v}_{\mathrm{z,ori}}$ because the rotation about P.A. does not change the line-of-sight velocity.

$$v_{\mathrm{los}} = v_{z,\mathrm{obs}} = v_{z,\mathrm{ori}} \tag{11}$$

6 Fourier Transform

The Fourier transform frequently appears in the context of (radio) astronomy. Although it can be easily done by a computer, the numerical way of calculating the Fourier transform is different from the mathematical way. The main difference is the phase center. The `fft_utils` module provides an output closer to the mathematical way. This module is a wrapper of `numpy.fft`. The following shows a comparison between the `plotastrodata.fftcentering` module and the original `numpy.fft` module.

```
# Example of numpy.fft.fft
import numpy as np

x = np.linspace(-99.5, 99.5, 200)
f = np.where(np.abs(x) < 10, 1, 0)

u = np.fft.fftshift(np.fft.fftfreq(len(x), d=x[1]-x[0]))
F = np.fft.fftshift(np.fft.fft(f))
```

```
# Example of plotastrodata.fft_utils
from plotastrodata.fft_utils import fftcentering
import numpy as np

x = np.linspace(-99.5, 99.5, 200)
f = np.where(np.abs(x) < 10, 1, 0)

F, u = fftcentering(f=f, x=x, xcenter=0)
```

Figure 7 shows the input and output functions obtained by the example scripts above using `numpy.fft.fft` and `plotastrodata.fft_utils.fftcentering`. One will expect that the phase center is at $x = 0$ or the center of x, and then the expected Fourier transform of Figure 7(a) will be Figure 7(c). The phase center is indeed specified in `fftcentering` by the argument of `xcenter=0`. In comparison, Figure 7(b) (i.e., the output of `numpy.fft.fft` alone) can be produced by `xcenter=x[0]`.

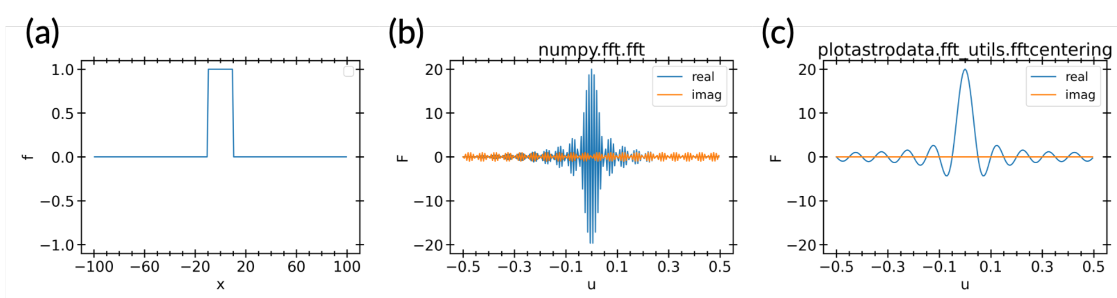

Figure 7: (a) The input function for the Fourier transform. (b) The real and imaginary parts obtained by `numpy.fft.fft`. (c) The real and imaginary parts obtained by from `plotastrodata.fft_utils.fftcentering`.

The inverse Fourier transform can be done by `ifftcentering` as follows.

```
from plotastrodata.fft_utils import fftcentering
from plotastrodata.fft_utils import ifftcentering
import numpy as np

x = np.linspace(-99.5, 99.5, 200)
f = np.where(np.abs(x) < 10, 1, 0)

F, u = fftcentering(f=f, x=x, xcenter=0)
ynew, xnew = ifftcentering(F=F, u=u, xcenter=0, x0=x[0], outreal=True)
print(np.max(np.abs(xnew - x)) < 1e-10, np.max(np.abs(ynew - y)) < 1e-10)
#True, True
```

The last two outputs are the same as the two inputs within the numerical error, which is consistent with $\mathrm{iFT}[\mathrm{FT}[f]] = f$. `ifftcentering` also requires the phase center value and additionally the starting value of the iFTed coordinate (xnew in the above example). This is because the Fourier transform is independent of absorlute values of the input coordinate. The argument of `outreal=True` means that the output ynew has only real values.

It is also possible for `fft_utils` to shift the phase of given FTed function and coordinate, by `fft_utils.shiftphase`.

```
from plotastrodata.fft_utils import shiftphase
import numpy as np

x = np.linspace(-99.5, 99.5, 200)
f = np.where(np.abs(x) < 10, 1, 0)

u = np.fft.fftshift(np.fft.fftfreq(len(x), d=x[1]-x[0]))
F = np.fft.fftshift(np.fft.fft(f))
F = shiftphase(F=F, u=u, xoff=-x[0])
```

The output of `shiftphase` is the same as the first output (F in the example) obtained by `fftcentering`. The offset argument of `xoff` is determined by "new center (0) − old center (x[0])".

The functions introduced above (`fftcentering`, `ifftcentering`, and `shiftphase`) have counterparts for 2D arrays: `fftcentering2`, `ifftcentering2`, and `shiftphase2`. `fftcentering2` requires an argument of `y` as well as `x`; these two arguments can be either 1D or 2D arrays (after `numpy.meshgrid`). The center is also expressed by two arguments `xcenter` and `ycenter`. Similarly `ifftcentering` takes arguments of `u`, `v`, `xcenter`, `ycenter`, `x0`, and `y0`, and `shiftphase` takes arguments of `u`, `v`, `xoff`, and `yoff`.

7 Fitting

Data analysis may lead a fit with a model to the data. For this purpose, the `plotastrodata` package has a module of `fitting_utils`, a wrapper of `emcee` (and `ptemcee`), `corner`, and `dynesty`; all of these packages are widely used in astronomical research for MCMC (Markov chain Monte Carlo) fitting.

Fitting process usually maximize a log-likelihood function. Here, a very simple function is used as the log-likelihood function with three variables.

```
def logl(p):
    x1, x2, x3 = p
```

```
    chi2 = (x1 / 1)**2 + (x2 / 2)**2 + (x3 / 4)**2
    return -0.5 * chi2
```

Then, the class of EmceeCorner in `fitting_utils` needs to be instantiated with a list of parameter boundaries (`bounds`) as well as the log-likelihood function.

```
from plotastrodata.fitting_utils import EmceeCorner

fitter = EmceeCorner(bounds=[[-5, 5], [-10, 10], [-20, 20]],
                     logl=logl, progressbar=False, percent=[16, 84])
```

If `progressbar=True`, some fitting methods show a progress bar in the terminal. `percent` specifies which percentiles are used to evaluate the uncertainty of each parameter. The fitting process is mainly performed by the `fit` method with arguments of `nwalkersperdim` (number of walkers per parameter), `nsteps` (number of steps including the burned-in steps), and `nburnin` (number of steps to be burned-in).

```
fitter.fit(nwalkersperdim=30, nsteps=11000, nburnin=1000,
           ntemps=1, savechain='chain.npy')
print('best:', fitter.popt)
#best: [-0.008 -0.026  0.008]
print('lower percentile:', fitter.plow)
#lower percentile: [-0.988 -1.954 -3.976 ]
print('50 percentile:', fitter.pmid)
#50 percentile: [ 0.007  0.030 -0.020]
print('higher percentile:', fitter.phigh)
#higher percentile: [0.990 2.012 3.955]
```

When `ntemps>1`, the `fit` method uses `ptemcee`, rather than `emcee`, to incorporate multiple temperatures. `savechain` specifies a file name to which the obtained chain is saved; if this is None, the chain is not saved to any file.

When two fitting results of the same data are compared between different models, a quantity of "evidence" can be used. The evidence is defined as a fractional volume of the likelihood function L in the parameter $(p_1 \cdots p_n)$ space: $\int \cdots \int L dp_1 \cdots dp_n / \int \cdots \int dp_1 \cdots dp_n$. Because the integrated range is not from infinity to infinity, the evidence value should be slightly smaller than $(\sqrt{2\pi} \cdot 2\sqrt{2\pi} \cdot 4\sqrt{2\pi})/(10 \cdot 20 \cdot 40) = 0.0157$ in the example. The evidence can be calculated using the method of `getDNSevidence`. This method uses DynamicNestedSampler in `dynesty`. The arguments for DynamicNestedSampler can also be used for this method.

```
fitter.getDNSevidence()
print('evidence:', fitter.evidence)
#evidence: 0.0153
```

The result of MCMC fitting can be plotted in two ways. One is the so-called corner plot, which shows the likelihood function as a function of each parameter or each pair of parameters. This can be done through the method of `plotcorner`. `show` specifies whether to show the figure in a window. `savefig` specifies the file name to which the figure is saved. `labels` specifies the parameter names that will be shown in the figure. `cornerrange` is a list of the boundaries for the figures or a list of fractions to be plotted.

```
fitter.plotcorner(show=True, savefig='corner.png',
                  labels=['par1', 'par2', 'par3'],
                  cornerrange=[[-4, 4], [-8, 8], [-16, 16]])
```

Another way is plotting the chains in the MCMC run as a function of the step. This can be done through the method of `plotchain`. The arguments have the same meaning as those in `plotcorner`.

```
fitter.plotchain(show=True, savefig='chain.png',
                 labels=['par1', 'par2', 'par3'],
                 ylim=[[-2, 2], [-4, 4], [-8, 8]])
```

The figure obtained through `plotchain` is simplified from the original chain. First, this method calculates the two percentiles given in the instantiation and the 50th percentile in the chain direction at each step. The two given percentiles are shown in a fainter color, while the 50th percentile is shown in a darker color. In addition, this method calculates the three percentiles in the step direction for each of the three percentiles derived above. For example, if 16th and 84th percentiles are given, this method calculates 16th percentile of 16th percentile, 50th percentile of 16th percentile, 84 percentile of 16th percentile, 16th percentile of 50th percentile, ..., and 84 percentile of 84 percentile. The second calculation in the step direction uses 1% of the total number of steps; if the total number of steps is < 200, the second calculation is skipped. In the figure, the 50th percentile of each percentile is shown with a thicker line, while the other two percentiles of each percentile is shown with a thinner line. As a result, the figure shows one thick-dark, two thick-faint, two thin-dark, and four thin-faint lines. Figures 8(a) and 8(b) show the outputs of `plotcorner` and `plotchain`, respectively.

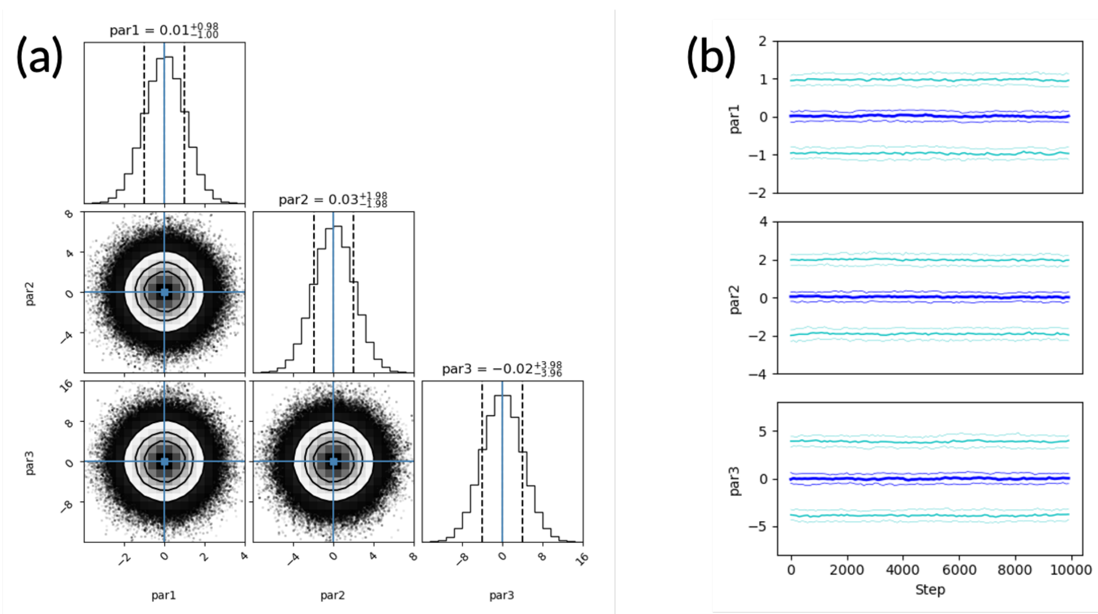

Figure 8: The results of (a) `plotcorner` and (b) `plotchain` in the example.

7.1 Grid serach

When the number of parameters in a fit is not many, it might be an option to calculate the log-likelihood function simply on a parameter grid, without the MCMC technique. This simple calculation also helps to check the overall shape of the log-likelihood function independently from the number of walkers or the number of steps of the MCMC run. This grid search can be performed using the method of `posteriorongrid` with an argument of `ngrid`, which specifies the number of grids for each parameter.

```
fitter.posteriorongrid(ngrid=[101, 201, 401])
print('best:', fitter.popt)
#best: [0. 0. 0.]
print('lower percentile:', fitter.plow)
#lower percentile: [-1. -2. -4.]
print('50th percentile:', fitter.pmid)
#50th percentile: [0. 0. 0.]
print('higher percentile:', fitter.phigh)
#higher percentile: [1. 2. 4.]
print('evidence:', fitter.evidence)
#evidence: 0.0155
```

The `posteriorongrid` method directly substitutes the meshed grid of the parameters to the log-likelihood function: for example, `np.meshgrid(p3, p2, p1, indexing='ij')` for three parameters. For this reason, the log-likelihood function must separate the parameter grid and the data grid. For example, if the log-likelihood function includes a subtruction of `x - p2`, where `x` is a data grid, this must be expressed as `np.subtract.outer(x, p2)` to separate the dimensions for the parameters from the dimension(s) for the data.

The result of the grid search can be visualized, as a corner plot, by the method of `plotongrid`. The arguments are the same as for `plotcorner`.

```
fitter.plotongrid(show=True, savefig='grid.png',
                  labels=['par1', 'par2', 'par3'],
                  cornerrange=[[-4, 4], [-8, 8], [-16, 16]])
```

Figure 9 shows the corner plot output from `plotongrid`. The first parameter has the coarsest grid compared to the width of the log-likelihood function.

8 Others

The `plotastrodata` package has a bit more things useful for astronomical research. Even though they might not be worth using from this package, it could make your code simpler to import them from the same package when other modules are imported from `plotastrodata`.

`const_utils` is a wrapper of `astropy.constants` and `astropy.units`. The variables defined in `const_utils` are simply in the float format and in SI units. These values are obtained through the `.to('unit name')` method in `astropy.units` and the `.value` attribute in `astropy.constants`. In addition, `const_utils` has variables with the name of metric prefixes from quecto (10^{-30}) to quetta (10^{30}). These variables are also simply in the float format.

```
import plotastrodata.const_utils as cu
print(cu.pc)
#3.085677581491367e+16
```

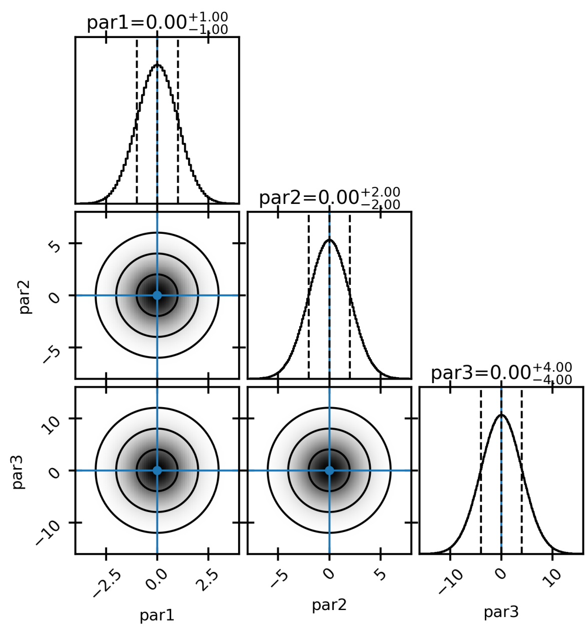

Figure 9: The results of `plotongrid`.

```
print(cu.M_sun)
#1.988409870698051e+30
print(cu.centi)
#0.01
```

`ext_utils` provides functions that are often used in astronimcal research. These functions only use the four arithmetic operations, `numpy.exp`, and variables imported from `const_utils`, meaning that the inputs may be N-D numpy arrays.

```
from plotastrodata.ext_utils import BnuT
# Planck function. 30 K, 230e9 Hz.
x = BnuT(T=30, nu=230e9)
print(x)
#np.float64(4.033705316844142e-16)
```

```
from plotastrodata.ext_utils import JnuT
# Brightness temperature (=wavelenth^2 / (2k_B) * BnuT). 30 K, 230e9 Hz.
x = JnuT(T=30, nu=230e9)
print(x)
#np.float64(24.818562479617647)
```