eddy

Release 1.2.1

Aug 18, 2023

Tutorials

eddy provides a suite of tools to extract kinematical information from spatially and spectrally resolved line data.

The `eddy.rotationmap.rotationmap` class enables the fitting of a Keplerian rotation pattern to a rotation map to infer geometrical properties of the disk, a dynamical mass of the star, or to infer the presence of a flared emission surface or warp in the disk.

CHAPTER 1

Installation

Installation is as easy as,

```
pip install astro-eddy
```

If you want to live on the edge, you can clone the repository and install that,

```
pip install .
```

but I cannot promise everything will work as described here.

CHAPTER 2

Contents

2.1 Frequently Asked Quetsions

This Notebook goes through some frequently asked questions about `eddy`.

How do I change the priors?

You want to use the `set_prior` function. This has three arguements, the name of the prior, the values of the prior and the type of prior. Two types of priors can be used: flat, uniform priors, `'flat'`, or Gaussian priors, `'gaussian'`. For the flat priors the two values define the minimum and maximum value, while for the Gaussian priors the values specify the mean and standard deviation of Gaussian.

```
cube.set_prior('vlsr', [-20e3, 20e3], 'flat')   # uniform prior
cube.set_prior('x0', [0.0, 0.05], 'gaussian')   # gaussian prior
```

2.2 1 - Fitting Rotation Maps

A common task for analyses of protoplanetary disks is to infer their geometrical properties, namely the source centre, $(x_0,\, y_0)$, the inclination i, position angle, PA, and stellar (dynamical) mass, M_{star}, by fitting a simple Keplerian rotation pattern to a measured rotation pattern. In this notebook we'll look at how to use `eddy` to fit rotation maps, provide some constraints on these model parameters and search for structure in the residuals.

2.2.1 TW Hya - A Geometrically Thin Case

Getting the Data

In this tutorial, we'll use $^{12}\mathrm{CO}\ (2-1)$ observations of TW Hya, described in Huang et al. (2018), and is available for download here.

```
[1]: import os
     if not os.path.exists('TWHya_CO_cube.fits'):
         !wget -O TWHya_CO_cube.fits -q https://dataverse.harvard.edu/api/access/datafile/:
     ↪persistentId?persistentId=doi:10.7910/DVN/PXDKBC/QULHRK
```

Making a Rotation Map

The first thing we need is a rotation map, i.e. a map of the line-of-sight velocity for each pixel. There's a few different ways to make this from your data, for example a traditional intensity-weighted average velocity (first moment map), which is what CASA produces with its immoments command, or 'quadratic' method as advocated for in Teague & Foreman-Mackey (2018) and implemented in bettermoments, or analytical fits, such as in GMoments. Each of these methods have their benefits, and you should consider which is the most appropriate statistic for the scienece you want to do.

As we are performing model fitting, we also want some idea of the uncertainties on the line-of-sight velocities. Unfortunately, CASA does not calculate these for you, but `bettermoments` does, following the uncertainties described in Teague (2019). These aren't essential for use with `eddy` as you are able to assume the uncertainties are some fraction of the line-of-sight velocity - not ideal, but it usually works.

This cell below will use `bettermoments` to create a rotation map using the `quadratic` method, masking all points which are $< 5\sigma$.

```
[ ]: !bettermoments TWHya_CO_cube.fits -clip 5
```

Downloading the Rotation Map

If you do not want to make your own line-of-sight velocity maps, you can download ones already made with `bettermoments` through the eddy Dataverse. This will contain the line-of-sight velocity map, `TWHya_CO_cube_v0.fits` and the associated uncertainty, `TWHya_CO_cube_dv0.fits`. If you have gone through the steps above you'll be ready, otherwise we can grab the necessary files from Dataverse.

```
[1]: import os
     if not os.path.exists('TWHya_CO_cube_v0.fits'):
         !wget -O TWHya_CO_cube_v0.fits -q https://dataverse.harvard.edu/api/access/
     ↪datafile/:persistentId?persistentId=doi:10.7910/DVN/KXELJL/VPLAM7
     if not os.path.exists('TWHya_CO_cube_dv0.fits'):
         !wget -O TWHya_CO_cube_dv0.fits -q https://dataverse.harvard.edu/api/access/
     ↪datafile/:persistentId?persistentId=doi:10.7910/DVN/KXELJL/BLMBH0
```

Loading up the Data

Within `eddy` we have defined a `rotationmap` class which provides all the functionality we'll need. Let's load it up along with other standard imports.

```
[1]: import matplotlib.pyplot as plt
     from eddy import rotationmap
     import numpy as np
```

In addition to the path to the data cube we want to fit, there are three extra arguments we need.

Uncertainties

If you have a map of the uncertainties for each pixel, they can be included with the `uncertainty` argument. If you don't, that's OK as it was assume a 10% uncertainty on each pixel by default. The uncertainty can be changed on the fly through the `cube.error` parameter. Note that this is actually a pretty bad approximation so it is highly recommend to include proper uncertainties. If you use the `bettermoments` naming convention, you can skip the `uncertainty` argument as it will search for the file `*_dv0.fits` in the same directory.

Downsampling

We have also downsampled the data so that we only included (roughly) spatially independent pixels. Additionally you can enter any integer to downsample by that factor. This is optional but useful for speeding up things when you're playing around.

Field of View

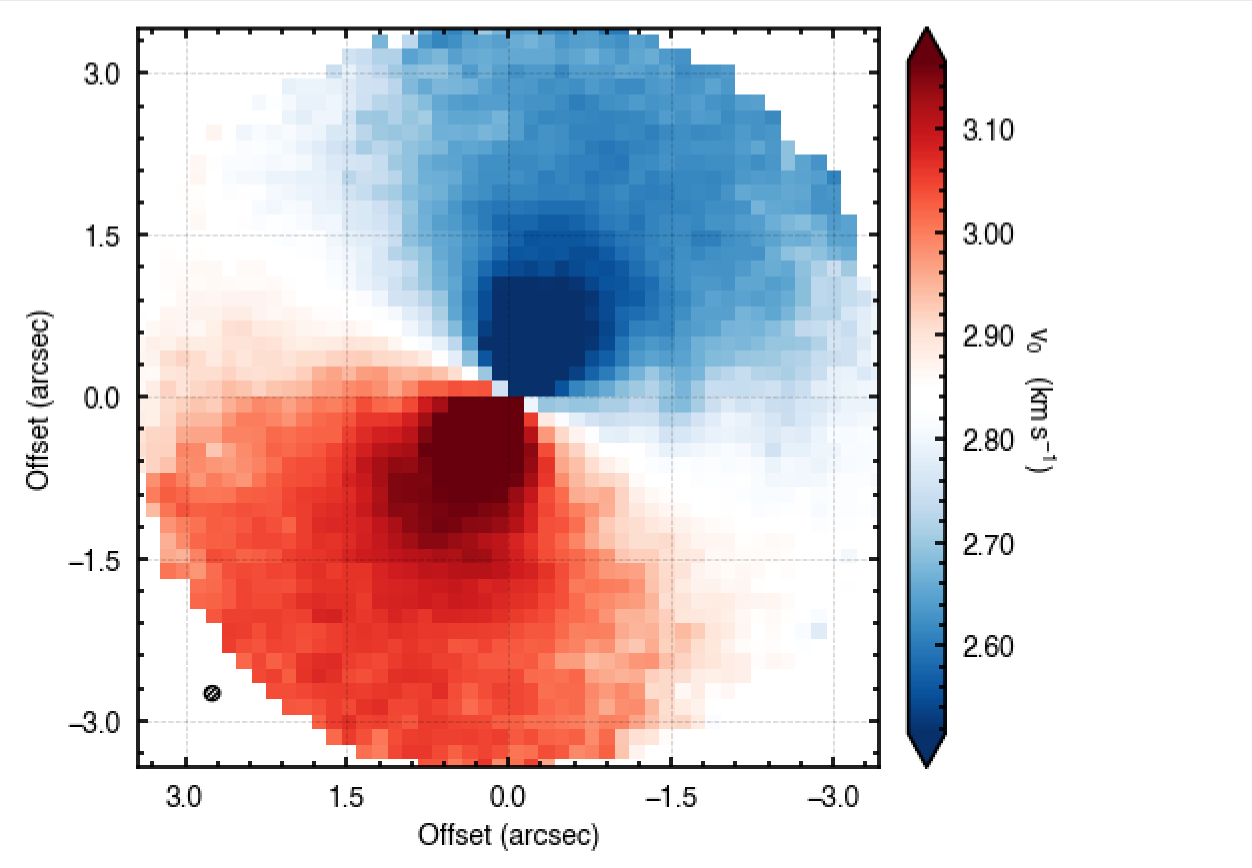
Using the `FOV` argument we have also clipped the cube down to the region of interest. Be aware, when `eddy` is making a rotation map, it makes it for the full image, even if you're only fitting an inner region defined by a mask. Clipping down to the region of interest (masked region) will considerably speed up the process.

[2]:
```
cube = rotationmap(path='TWHya_CO_cube_v0.fits',
                   uncertainty='TWHya_CO_cube_dv0.fits',
                   downsample='beam',
                   FOV=7.0)
```

Data Inspection

We can also inspect the data to make sure it looks how we'd expect it to. All we're looking for here is that the downsampling and field of view that were chosen look reasonable, and you can see the typical dipolar morphology indicative of a rotating source.

[3]:
```
cube.plot_data()
```



There are a few other tools that may be useful for a quick inspection of the data described in TBD.

Fitting the Rotation Map

Here we describe how to fit the data using the `fit_map` function. For this introductory case, we assume a geometrically thin, Keplerian disk model where

$$v_0 = \sqrt{\frac{GM_{\rm star}}{r}} \cdot \cos\phi \cdot \sin i + v_{\rm LSR} \quad \text{or} \quad v_0 = v_{\rm kep} \cdot \cos\phi \cdot \sin i + v_{\rm LSR}$$

where r is the cylindrical distance from the star (converted from angular distances in arcseconds to au by multiplying by the source distance, d), ϕ is the polar angle of the pixel (measured east of north relative to the redshifted major axis) and $v_{\rm LSR}$ is the systemic velocity.

For the fitting, we need to know which of these values we're fixing and which we want to fit for. Given the extreme degeneracy between i and $M_{\rm star}$ for low inclination disks, a good idea is to either fix i to a value found from fitting the continuum, or adopt a literature value for the dynamical mass. For this example we want to find the source center, $(x0,\, y0)$, the position angle of the disk, PA, the stellar mass $M_{\rm star}$ and the systemic velocity, $v_{\rm LSR}$, while holding the inclination fixed at $i = 5.8°$, the value adopted in Teague et al. (2019).

Thus we have our free model parameters, $\Theta = \{x_0,\, y_0,\, {\rm PA},\, M_{\rm star},\, v_{\rm LSR}\}$, and our fixed model parameters, $\Theta_{\rm fixed} = \{i,\, d\}$.

We the provide two things to `fit_map`:

- `p0`: a list of the initial guesses for the free parameters, Θ.

- params: a dictionary containing both the indices of the free variables in p0 as an **integer** and the fixed values for all other variables as a **float**.

With this dictionary framework it is possible to hold certain parameters fixed and others free. In particular, if you know the rotation direction of the disk (controlled by the sign of i, discussed in a later tutorial) then this is a good parameter to fix.

[4]:
```
# Dictionary to contain the disk parameters.

params = {}

# Start with the free variables in p0.

params['x0'] = 0
params['y0'] = 1
params['PA'] = 2
params['mstar'] = 3
params['vlsr'] = 4

# Provide starting guesses for these values.

p0 = [0.0, 0.0, 151., 0.81, 2.8e3]

# Fix the other parameters. All values which are to be fixed must be floats.

params['inc'] = 5.8     # degrees
params['dist'] = 60.1    # parsec
```

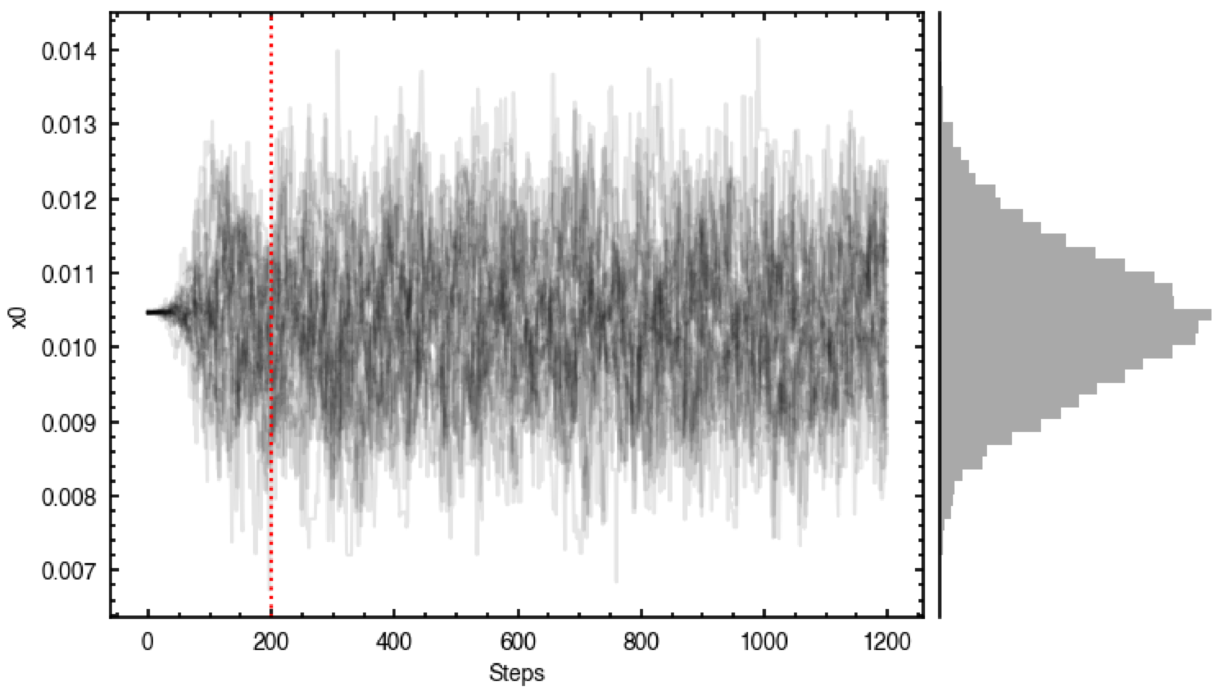
The fit_map function has several steps:

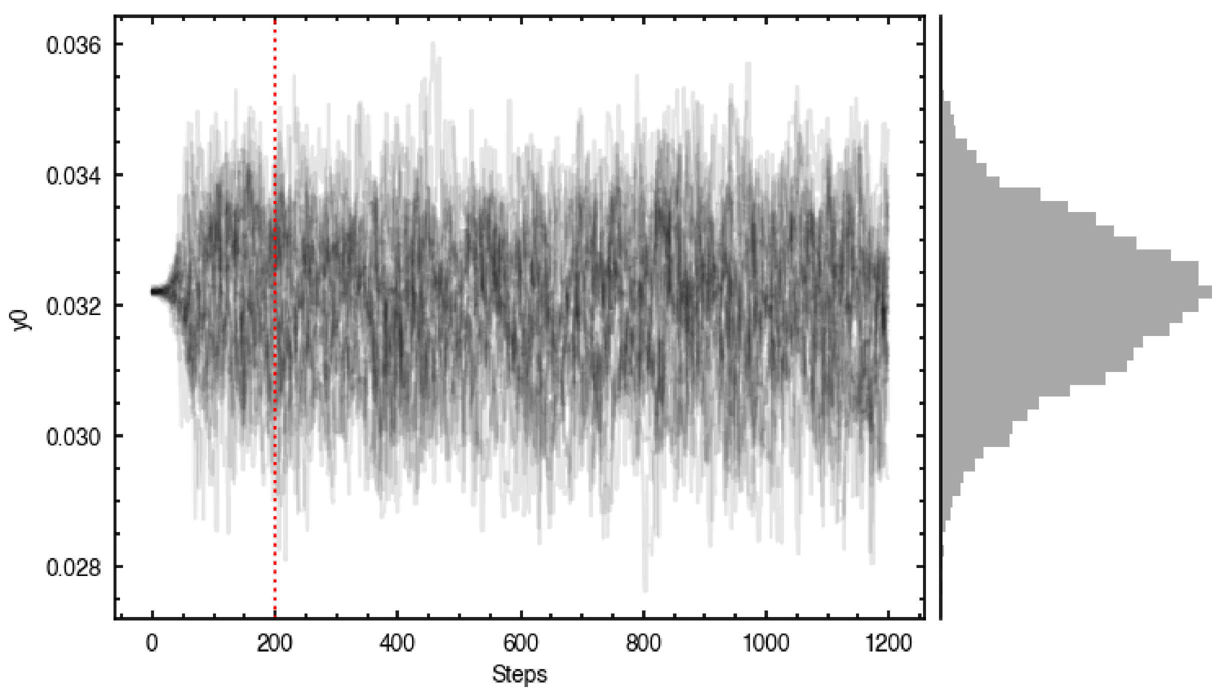
1. Using the initial values in p0 to deproject the data, it will create a mask of the regions to fit. Note that if the initial guesses are poor, the mask will not be well defined. This can be circumvented with the niter argument, *discussed later*.
2. If optimize=True, which is strongly recommended, it tries to find the values in p0 which maximize the likelihood function. Using these updated p0 values it will then recalculate the regions to fit.
3. It will the make any specified diagnostic plots.
4. It will return any requested products.

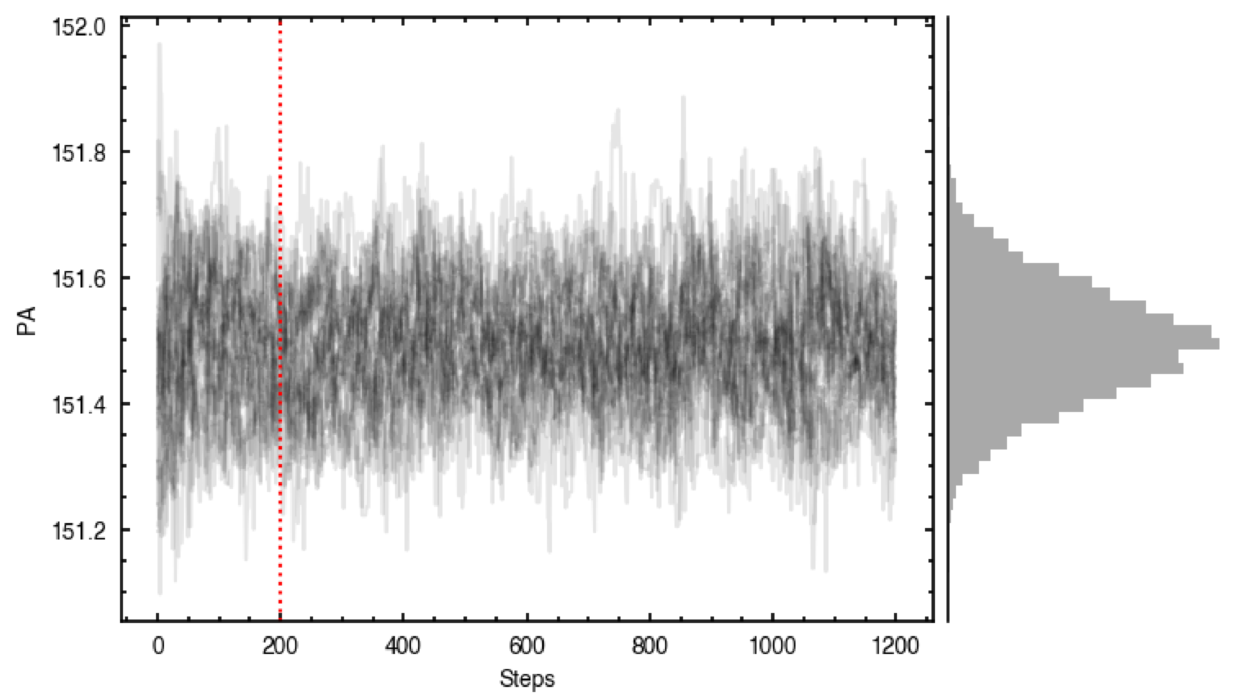
[5]:
```
samples = cube.fit_map(p0=p0, params=params,
                       nwalkers=32, nburnin=200, nsteps=1000)
```

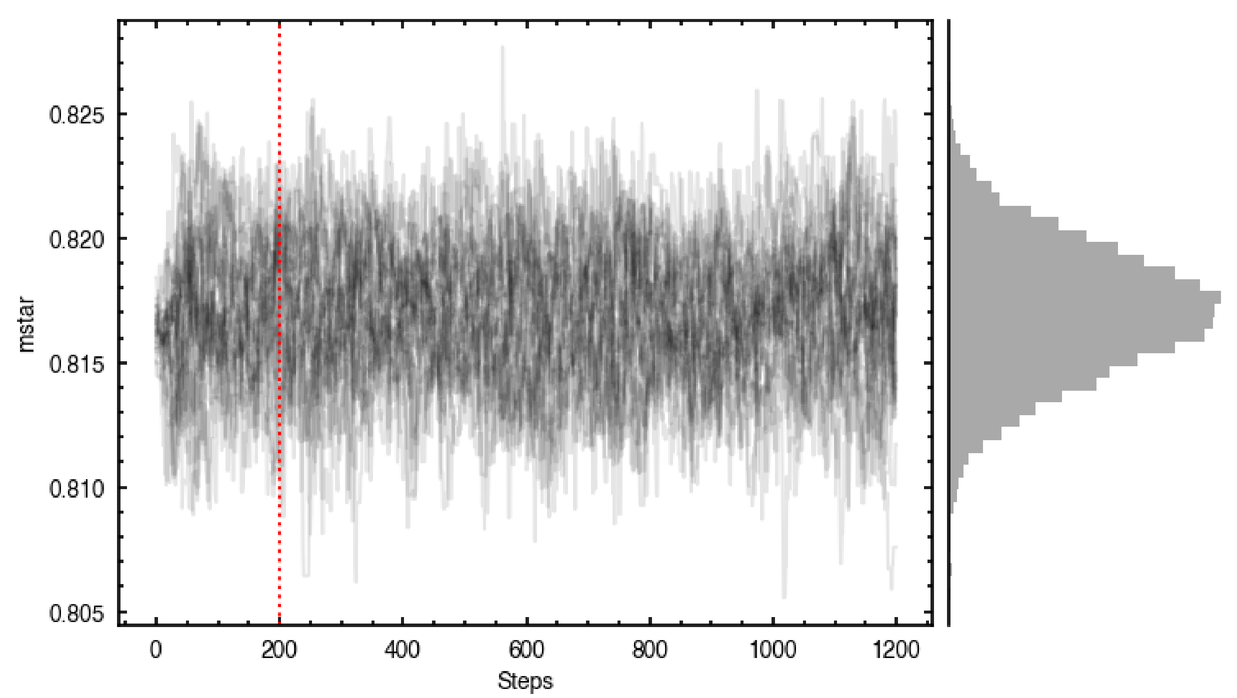
```
Assuming:
        p0 = [x0, y0, PA, mstar, vlsr].
Optimized starting positions:
        p0 = ['1.05e-02', '3.22e-02', '1.51e+02', '8.17e-01', '2.84e+03']
```

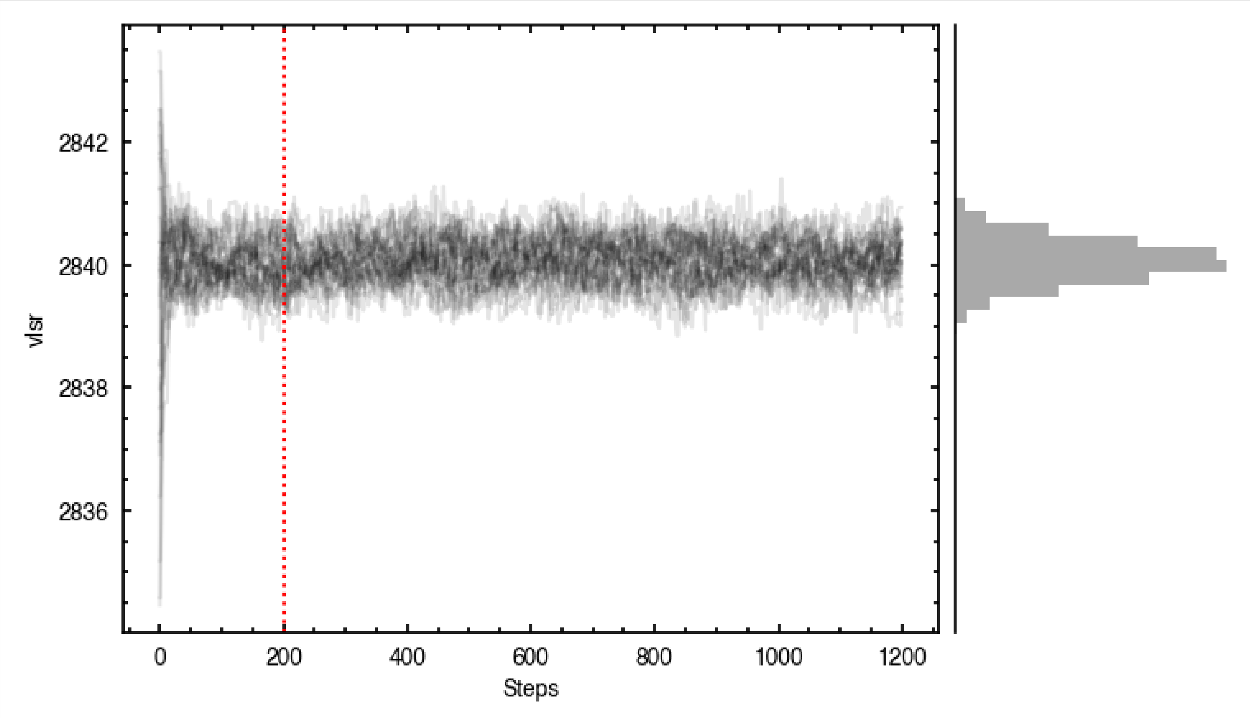
```
100%|| 1200/1200 [00:06<00:00, 187.49it/s]
```

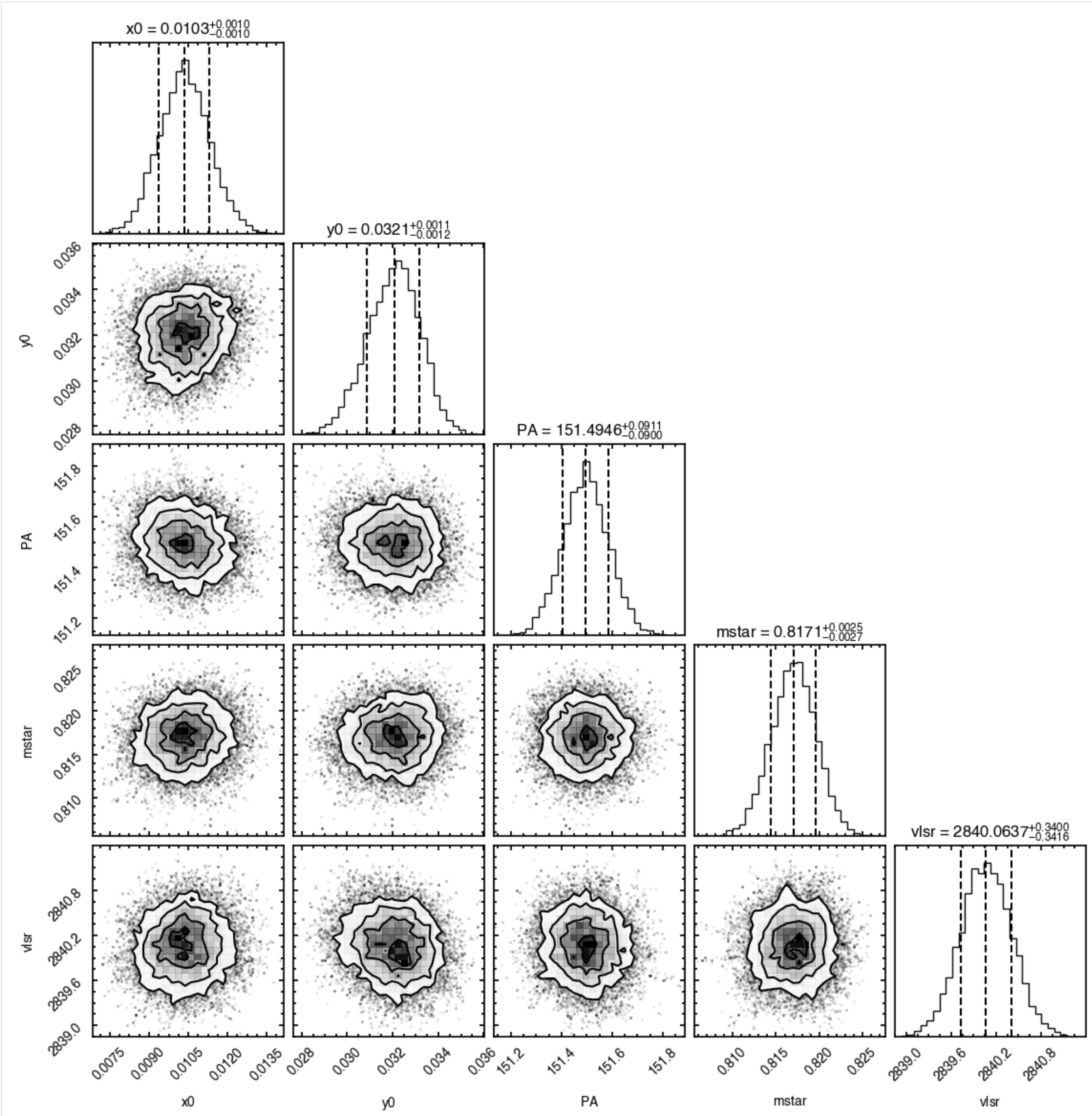

0.014
0.013
0.012
0.011
0.010
0.009
0.008
0.007
x0
0
200
400
600
800
1000
1200
Steps


0.036
0.034
0.032
0.030
0.028
y0
0
200
400
600
800
1000
1200
Steps

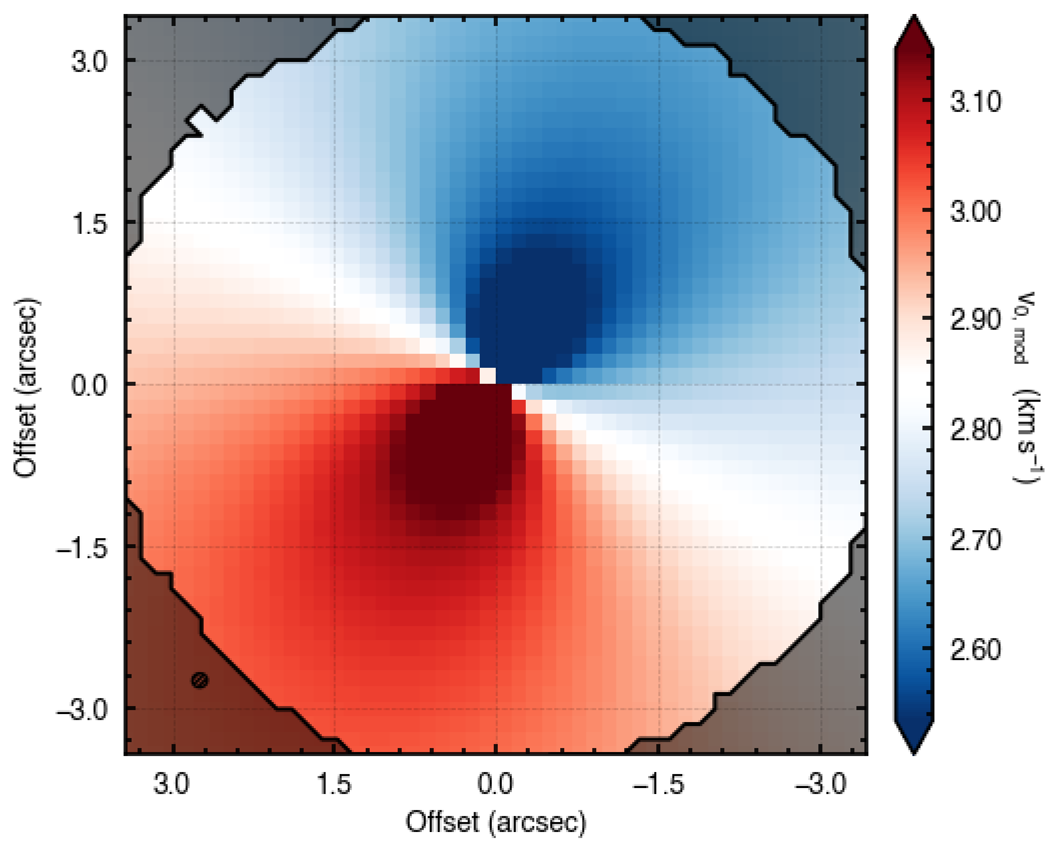

PA
Steps

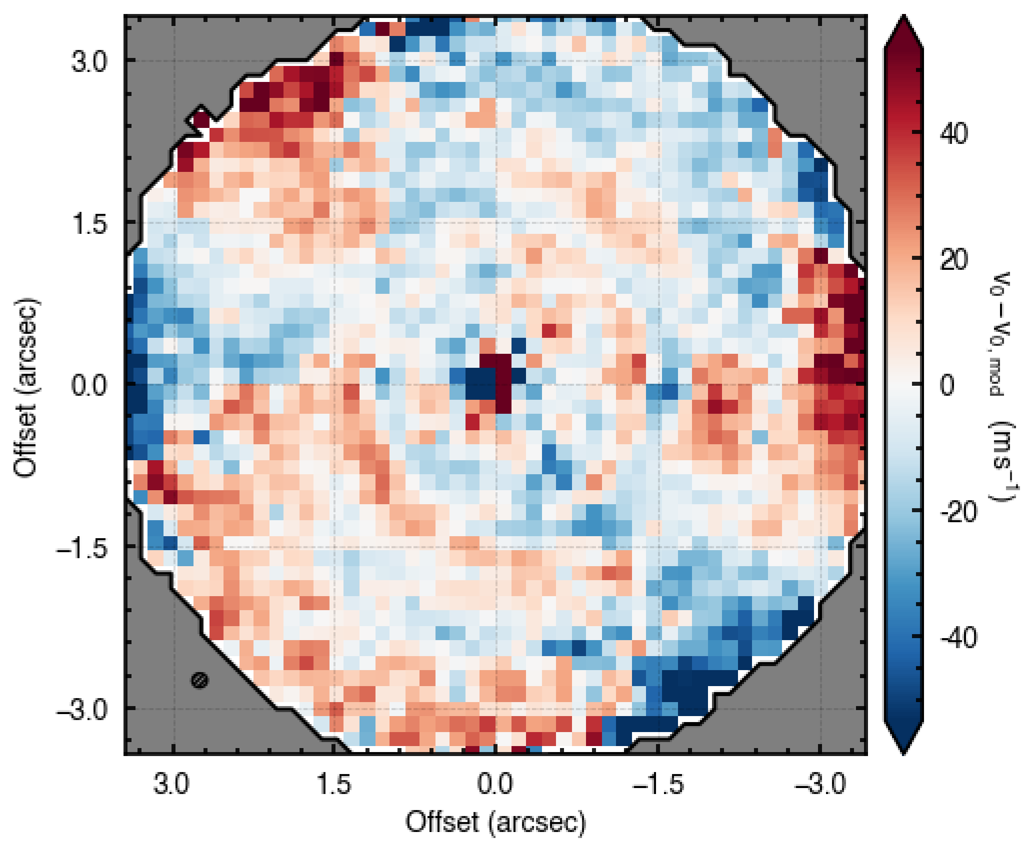

mstar
Steps


2842
2840
2838
2836
vlsr
0
200
400
600
800
1000
1200
Steps



x0 = 0.0103$^{+0.0010}_{-0.0010}$
y0 = 0.0321$^{+0.0011}_{-0.0012}$
PA = 151.4946$^{+0.0911}_{-0.0900}$
mstar = 0.8171$^{+0.0025}_{-0.0027}$
vlsr = 2840.0637$^{+0.3400}_{-0.3416}$
x0
y0
PA
mstar
vlsr


3.0
1.5
0.0
–1.5
–3.0
Offset (arcsec)
3.0
1.5
0.0
–1.5
–3.0
Offset (arcsec)
3.10
3.00
2.90
2.80
2.70
2.60
$v_{0,\,\mathrm{mod}}$ (km s^{-1})



Note that through these tutorials, we use only a small number of steps for the MCMC as we just want to demonstrate the functionality of `eddy`. When using this for publishable results, it is strongly recommended to perform checks on the walkers to make sure they have sufficiently burnt in and are converged.

Diagnostic Plots

By default, `fit_map` will produce all the diagnostic plots. You can select which plots you want using the `plots` argument, which takes either a single (string) value, or a list of strings. The available options are:

- `'walkers'` - Shows the traces of the walkers for each parameter. In each panel, the dashed red line shows the end of the burn-in period. The histogram shows the collapsed posterior distribution for all samples taken after the burn-in period.
- `'corner'` - A typical corner (covariance) plot using corner.py from Dan Foreman-Mackey. The labels will show the median value with uncertainties representing the 16th to 84th percentile range.
- `'bestfit'` - A plot of the best-fit model (using the median value of the posteriors) with the mask overlaid.
- `'residual'` - A plot of the residuals between the data and the best fit model with the mask overlaid.

For example, if you wanted to just plot the residual you would use, `plots='residual'`, while for plots of the walkers and the corner plot you would use `plots=['walkers', 'corner']`. If you would like no plots, you can use `plots='none'` (careful to use a string here as `plots=None` is interpreted as the default of all plots).

Returned Products

There are several different statistics or data products that can be returned after the MCMC, controlled with the `returns` argument. As for the plots, this takes either a single string or a list of strings. The available options are:

- `'samples'` - A `(nsteps, nparams)` shaped array of the posterior samples after the burn-in period. This has not been thinned, so must be done by hand if necessary.
- `'percentiles'` - A `(3, nparams)` shaped array of the 16th, 50th and 84th percentiles of each parameter's posterior distribution, a proxy of the standard deviation for a Gaussian distribution.
- `'lnprob'` - A `nsteps` sized array of the log-probability for each posteroir sample.
- `'model'` - A 2D array of the best-fit model using the median value from the samples.
- `'residuals'` - A 2D array of the residuals after subtracting the best-fit model from the data.
- `'dict'` - A `params` dictionary where the free parameters have been replaced with the median values of the posterior distributions.

By default `eddy` will just return the samples from the MCMC.

Masking Data

Oftentimes it is useful to only fit a specific region of your rotation map, perhaps because you are only interested in a spatial radial region. Within `eddy`, you can define a map based on the disk-frame $(r,\,\phi)$ coordinates and / or on the value of v_0 to avoid cloud-contaminated velocities.

For each of these optionn, `r`, `phi` and `v`, you can specify the minimum and maximum value to include or exclude in your mask. For example, to only include the regions between $0.5''$ and $1.0''$ we would use,

```
params['r_min'] = 0.5
params['r_max'] = 1.0
```

If instead we wanted to fit everywhere except a radial region between $0.5''$ and $1.0''$ and an azimuthal region where $|\phi| \leq 30°$ we would use,

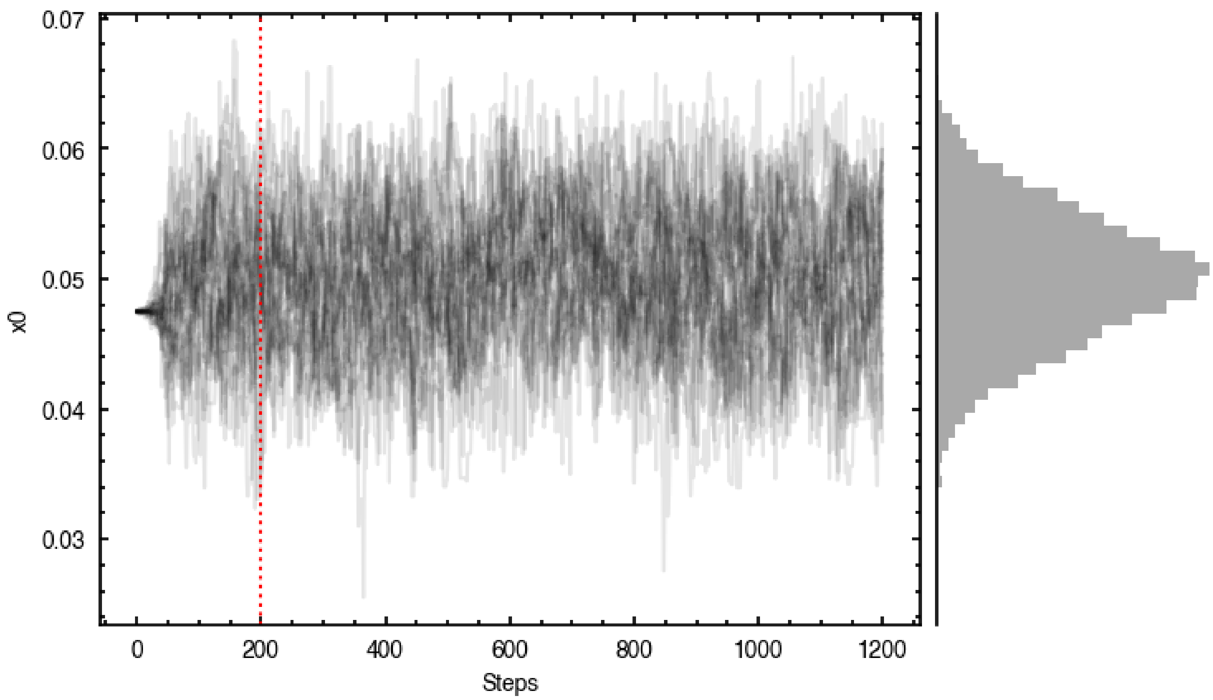
```
params['r_min'] = 1.0
params['r_max'] = 2.0
params['exclude_r'] = True
params['phi_min'] = -30.0
params['phi_max'] = 30.0
params['exclude_phi'] = True
```

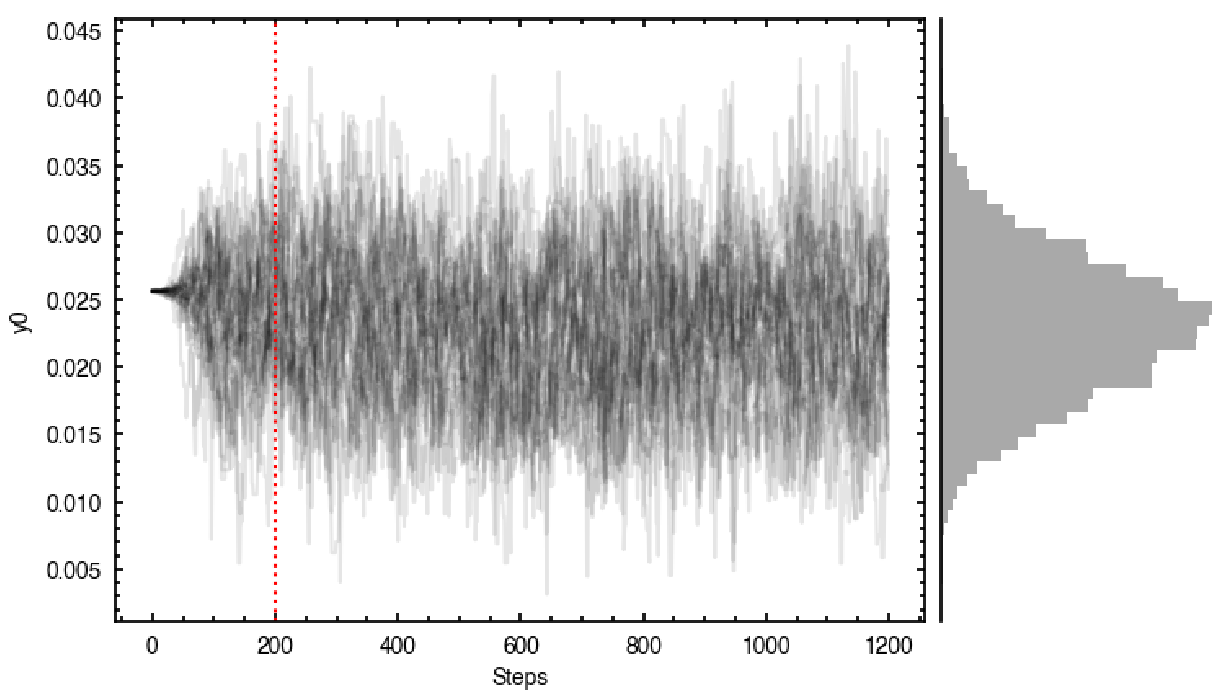
In addition, there is the key `'abs_phi'` which will assume ϕ is mirrored about the major axis of the disk to aid in defining regions on either side of the minor axis, for example. Note that, as mentioned before, the mask generation will adopt the initial geometrical properties provided to the `fit_map` function. If these are wildly off the true values, the resulting mask will not be very good.

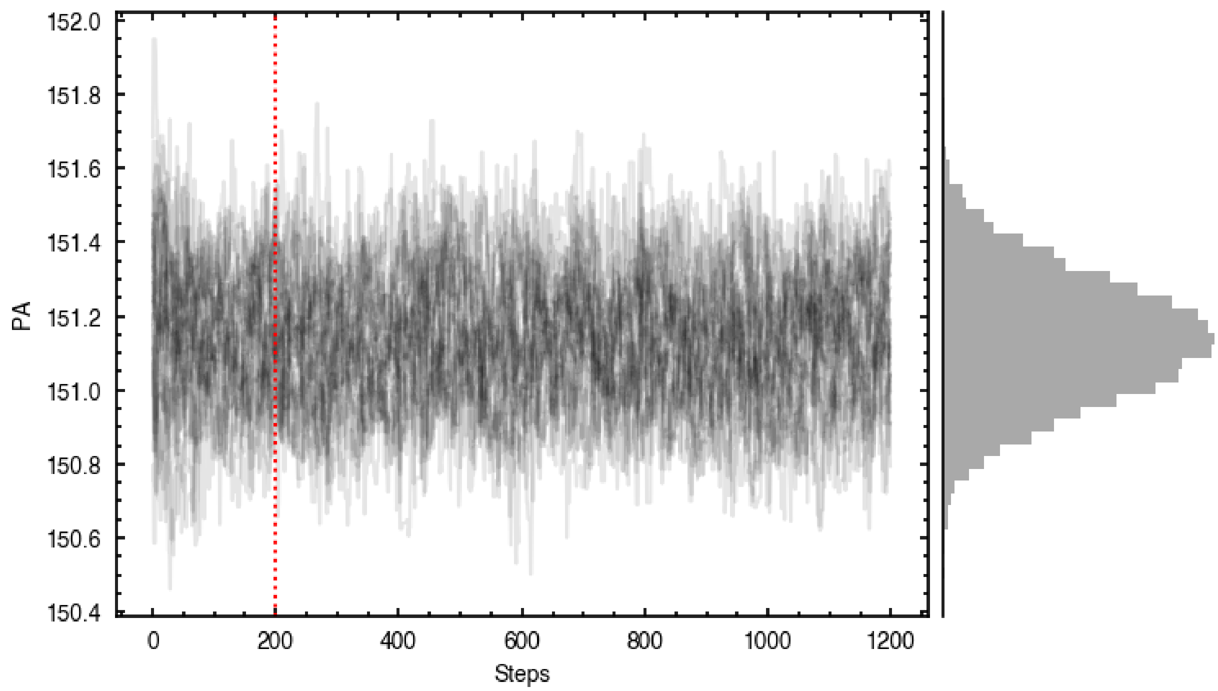
```
[6]: # fit only an annulus of points

params['r_min'] = 1.0
params['r_max'] = 2.0

samples = cube.fit_map(p0=p0, params=params,
                       nwalkers=32, nburnin=200, nsteps=1000,
                       plots=None)
```

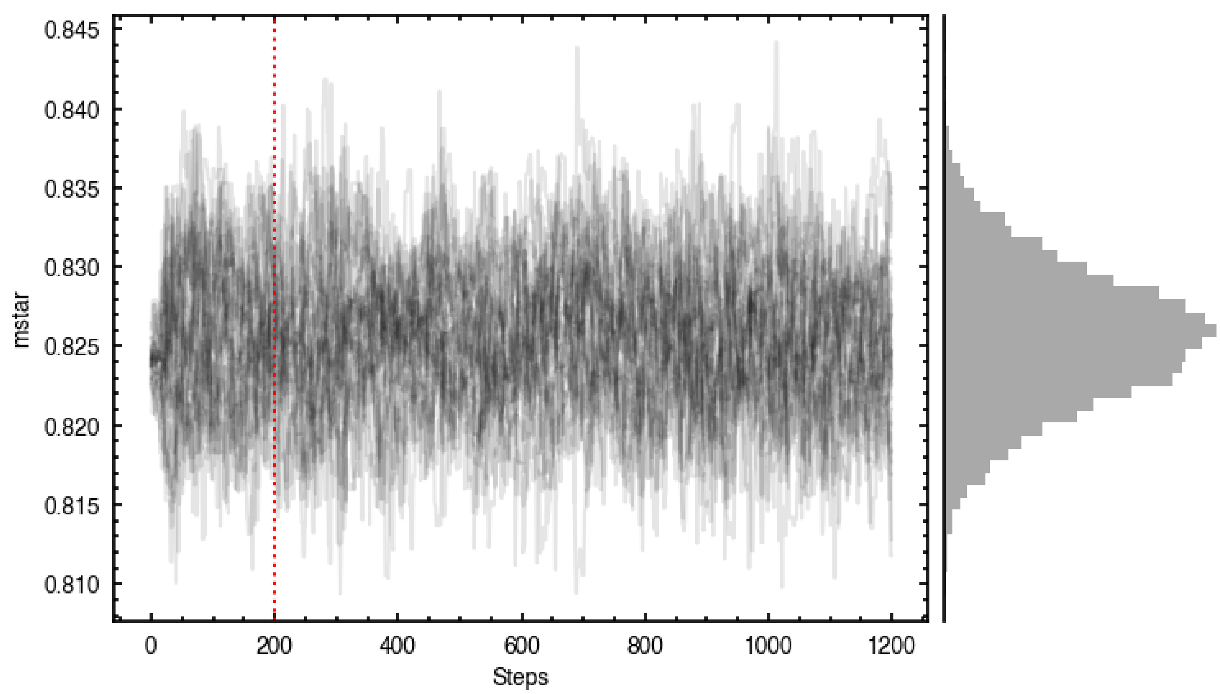
```
Assuming:
        p0 = [x0, y0, PA, mstar, vlsr].
Optimized starting positions:
        p0 = ['4.75e-02', '2.56e-02', '1.51e+02', '8.24e-01', '2.84e+03']
```

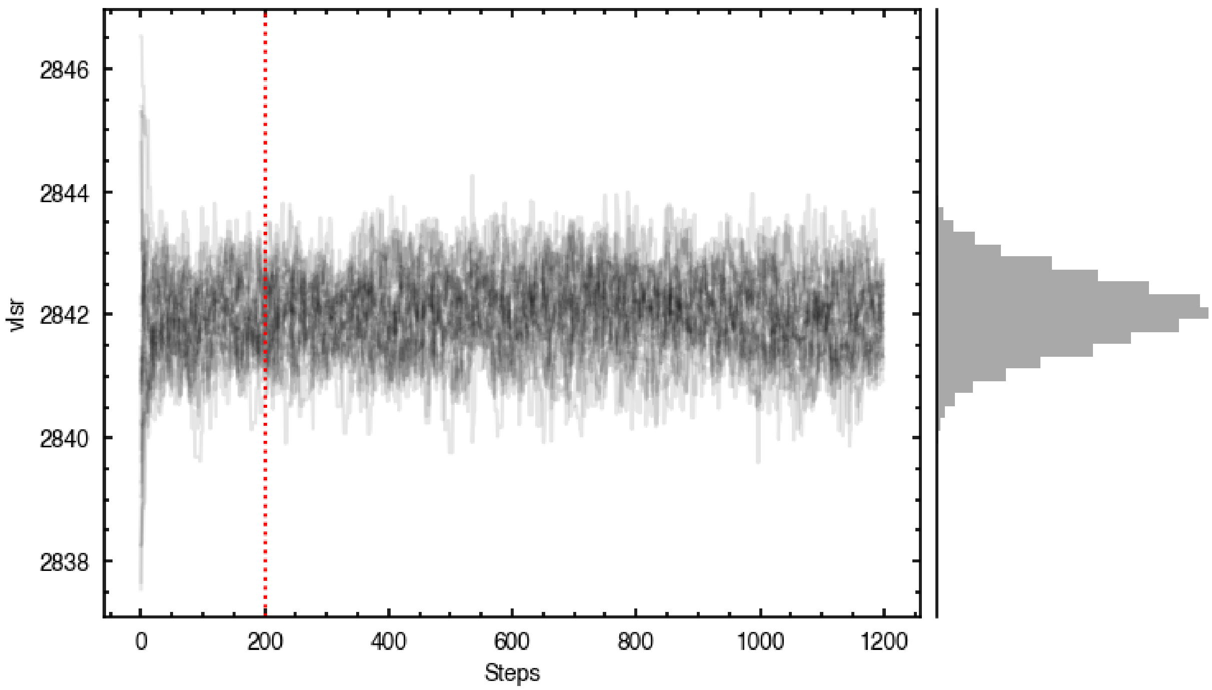
```
100%|| 1200/1200 [00:06<00:00, 187.79it/s]
```

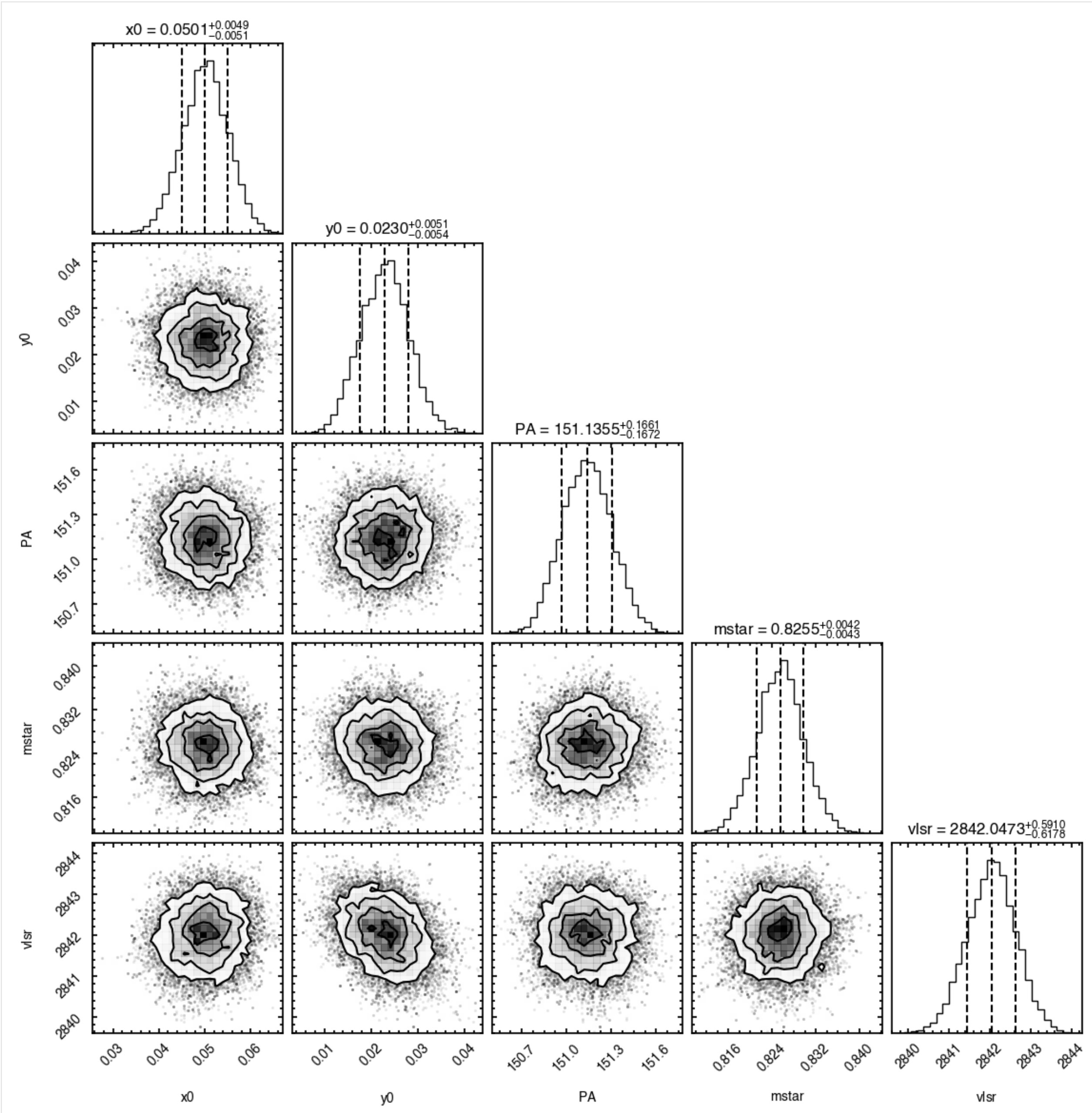




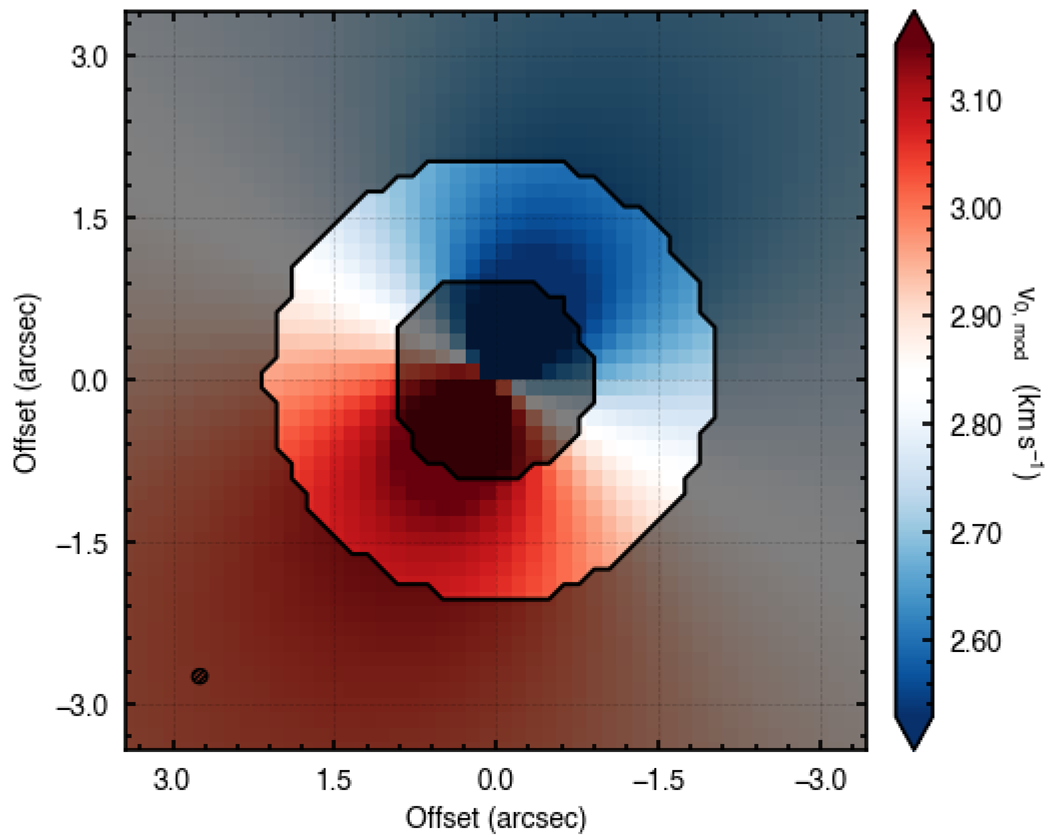

152.0
151.8
151.6
151.4
151.2
151.0
150.8
150.6
150.4
PA
0
200
400
600
800
1000
1200
Steps

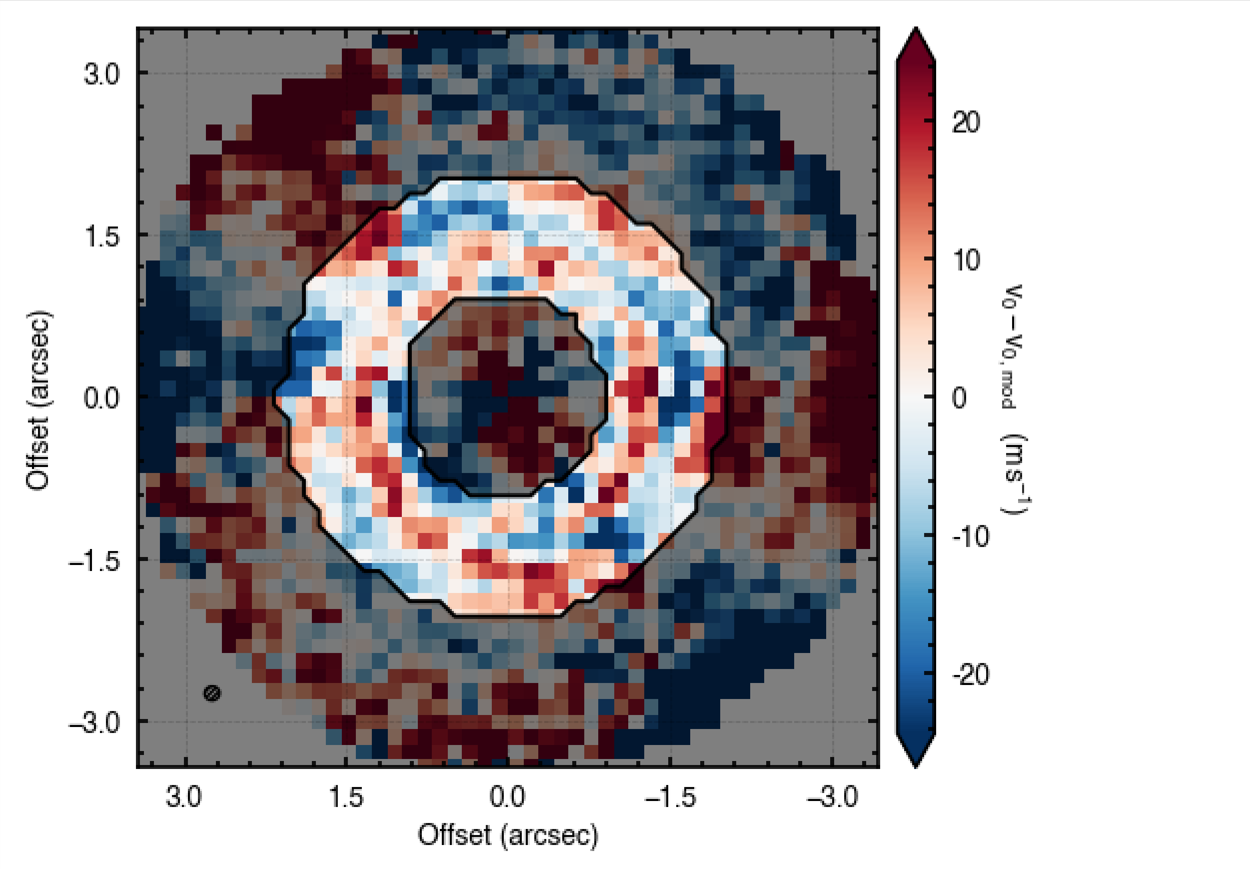

0.845
0.840
0.835
0.830
0.825
0.820
0.815
0.810
mstar
0
200
400
600
800
1000
1200
Steps


2846
2844
2842
2840
2838
vlsr
0
200
400
600
800
1000
1200
Steps



x0 = 0.0501$^{+0.0049}_{-0.0051}$
y0 = 0.0230$^{+0.0051}_{-0.0054}$
PA = 151.1355$^{+0.1661}_{-0.1672}$
mstar = 0.8255$^{+0.0042}_{-0.0043}$
vlsr = 2842.0473$^{+0.5910}_{-0.6178}$
x0
y0
PA
mstar
vlsr


3.0
1.5
0.0
–1.5
–3.0
Offset (arcsec)
3.0
1.5
0.0
–1.5
–3.0
Offset (arcsec)
3.10
3.00
2.90
2.80
2.70
2.60
v0, mod (km s−1)



As you can see here, the data is only fit to an annuli of points between 1″ and 2″, but the model is produced for the entire field of view. This is why choosing an appropriate `FOV` value will greatly speed up the fitting process.

Iterations

As mentioned several times, the definition of the mask will depend on the initial guesses for your model parameters. Sometimes these are hard to guess, particularly with noisy data or if you're using an elevated emission surface (see Tutorial 2). One brute-force approach for tackling this problem is to use the `niter` argument in `fit_map`. This will run `niter` iterations of the MCMC sampling, each time updating the `p0` values with the median values of the posteriors from the previous run. This should help nudge the `p0` values to more reasonable starting ones.

To demonstrate this, consider starting the attempt we had above with poor `p0` values.

[7]:
```
params = {}

params['x0'] = 0
params['y0'] = 1
params['PA'] = 2
params['mstar'] = 3
params['vlsr'] = 4

p0 = [0.4, -0.2, 54., 1.5, 2.3e3]

params['inc'] = 5.8
params['dist'] = 60.1
params['r_min'] = 1.0
```

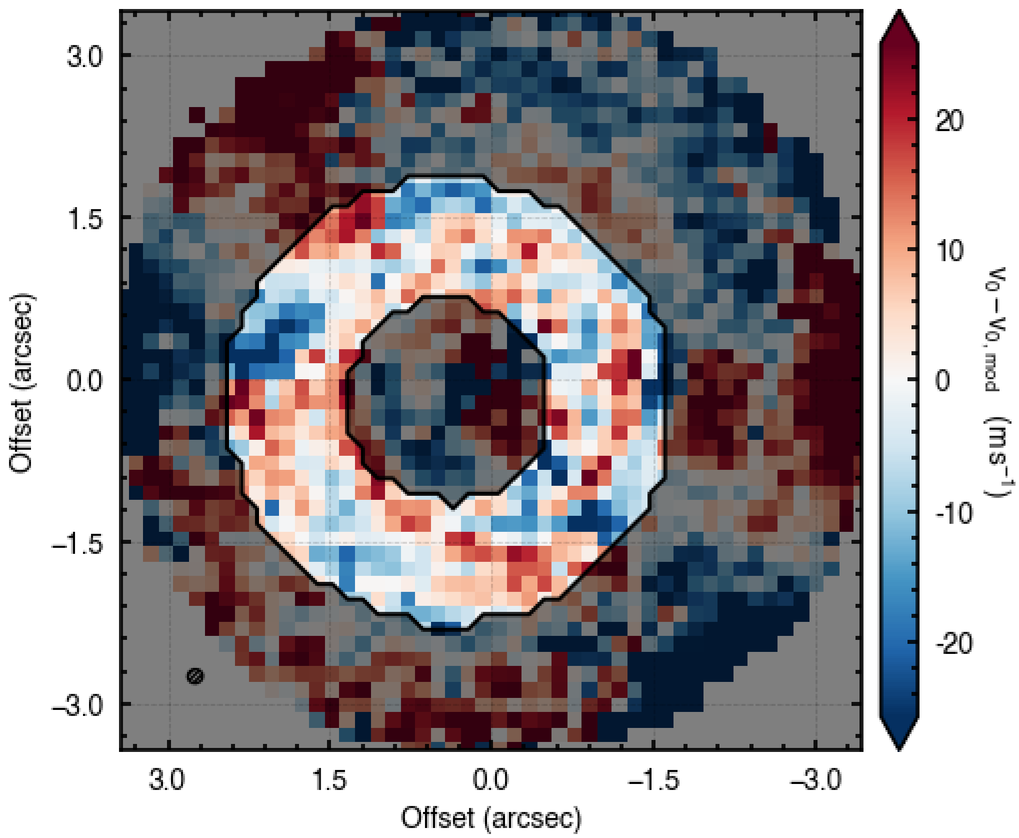
(continues on next page)

(continued from previous page)

```
params['r_max'] = 2.0

samples = cube.fit_map(p0=p0, params=params, optimize=False,
                       nwalkers=32, nburnin=200, nsteps=1000,
                       plots='residual')
```

```
Assuming:
        p0 = [x0, y0, PA, mstar, vlsr].
```

```
100%|| 1200/1200 [00:06<00:00, 187.52it/s]
```



Clearly this is not such a good fit. With enough time, the walkers should converge to the correct value (particularly as this is very good data), however they will still be fitting an incorrect disk region. Trying above, but now using `niter=2`, we can see this does a better job.

```
[8]: params = {}

params['x0'] = 0
params['y0'] = 1
params['PA'] = 2
params['mstar'] = 3
params['vlsr'] = 4

p0 = [0.4, -0.2, 54., 1.5, 2.3e3]

params['inc'] = 5.8
params['dist'] = 60.1
```

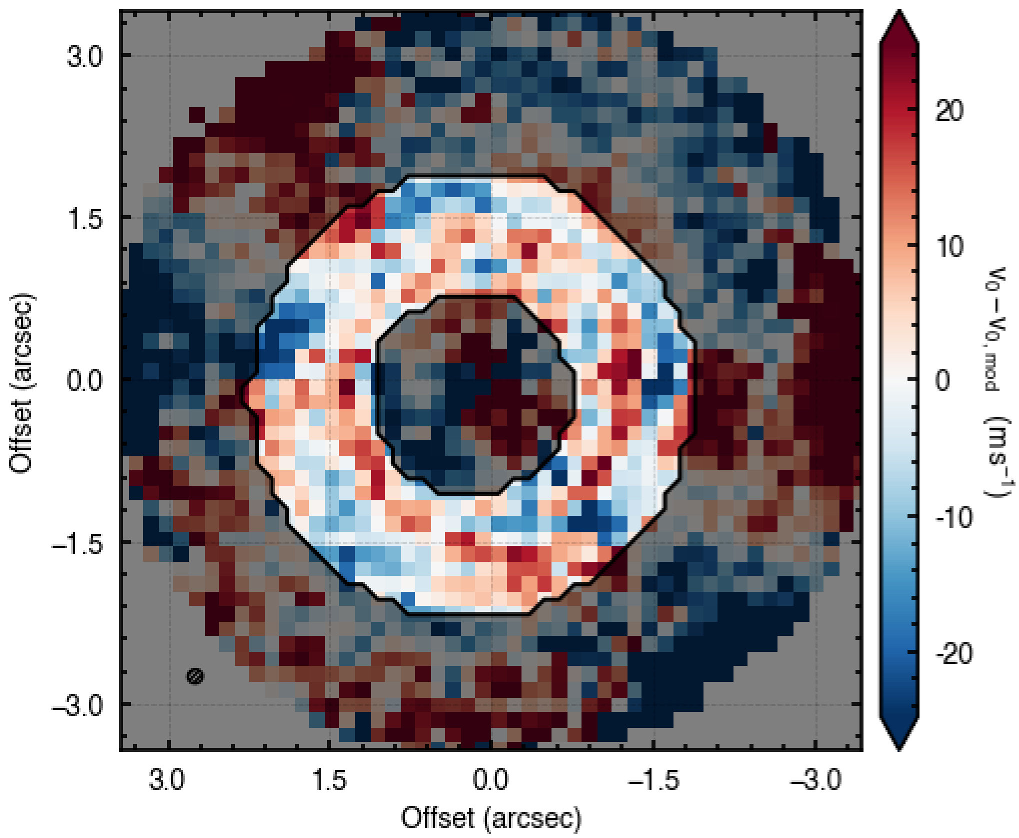
(continues on next page)

(continued from previous page)

```
params['r_min'] = 1.0
params['r_max'] = 2.0

samples = cube.fit_map(p0=p0, params=params, optimize=False,
                       nwalkers=32, nburnin=200, nsteps=[200, 1000],
                       plots='residual', niter=2)
```

```
Assuming:
        p0 = [x0, y0, PA, mstar, vlsr].
```

```
100%|| 400/400 [00:02<00:00, 198.86it/s]
100%|| 1200/1200 [00:06<00:00, 184.35it/s]
```



Here we have provided two values for `nsteps`, one for each of the iterations. `nwalkers`, `nburnin` and `nsteps` can accept lists of values that would be used for each iteration. It's important to note that `niter` should not be used in place of more walkers or steps, but rather to nuge things in the right direction.

Parallelization

It's sometimes useful to parallelize the fitting. We can implement a naive approach using the `multiprocessing` package. We can provide a `pool` variable to `fit_map` which will interface with the MCMC. Note that this is implemented using the `with` statement so that it deals with the starting and closing of the pool.

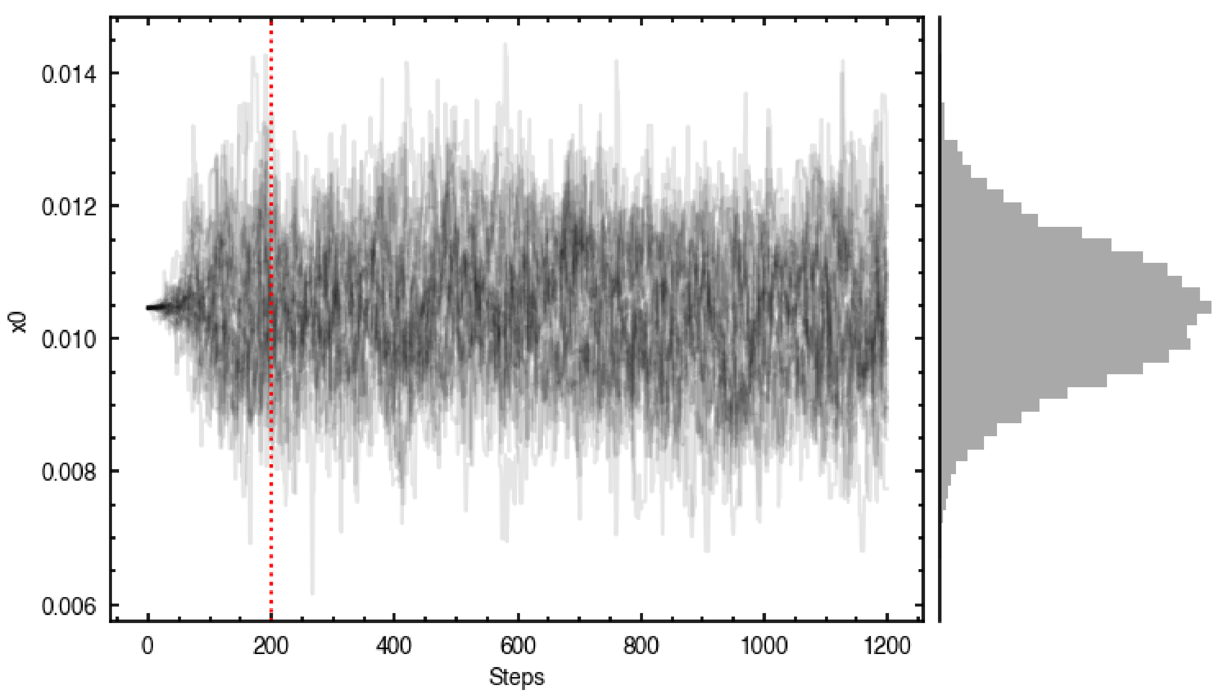
NOTE: It seems that currently the priors do not work correctly with parallelization. This is being investigated.

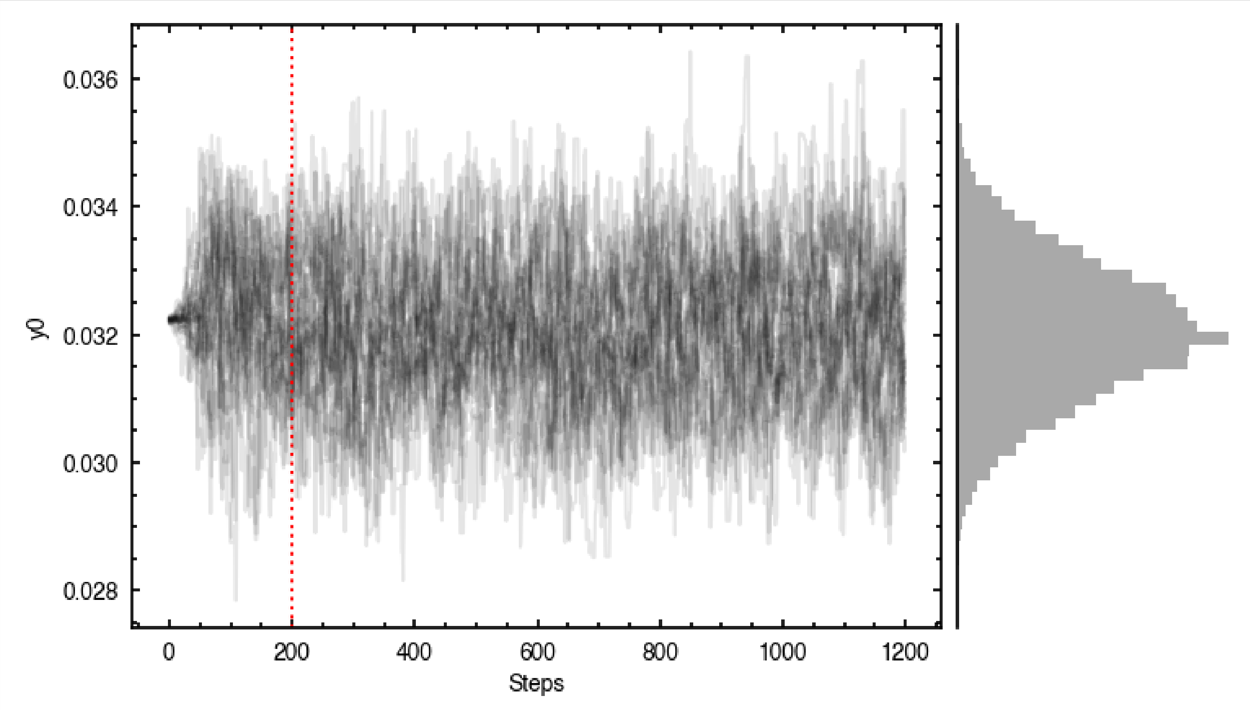
---

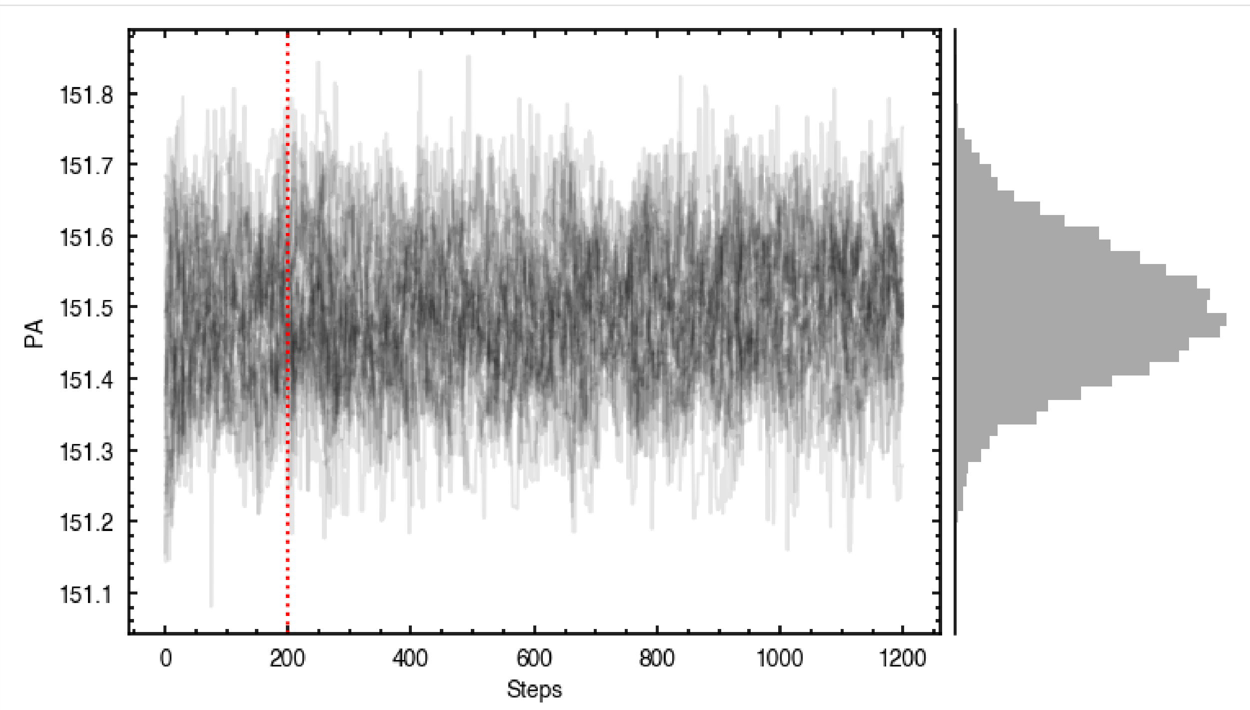
[9]:
```
from multiprocessing import Pool

params = {}

params['x0'] = 0
params['y0'] = 1
params['PA'] = 2
params['mstar'] = 3
params['vlsr'] = 4

p0 = [0.0, 0.0, 151., 0.81, 2.8e3]

params['inc'] = 5.8
params['dist'] = 60.1

with Pool() as pool:
    cube.fit_map(p0=p0, params=params,
                 nwalkers=32, nburnin=200, nsteps=1000,
                 pool=pool)
```

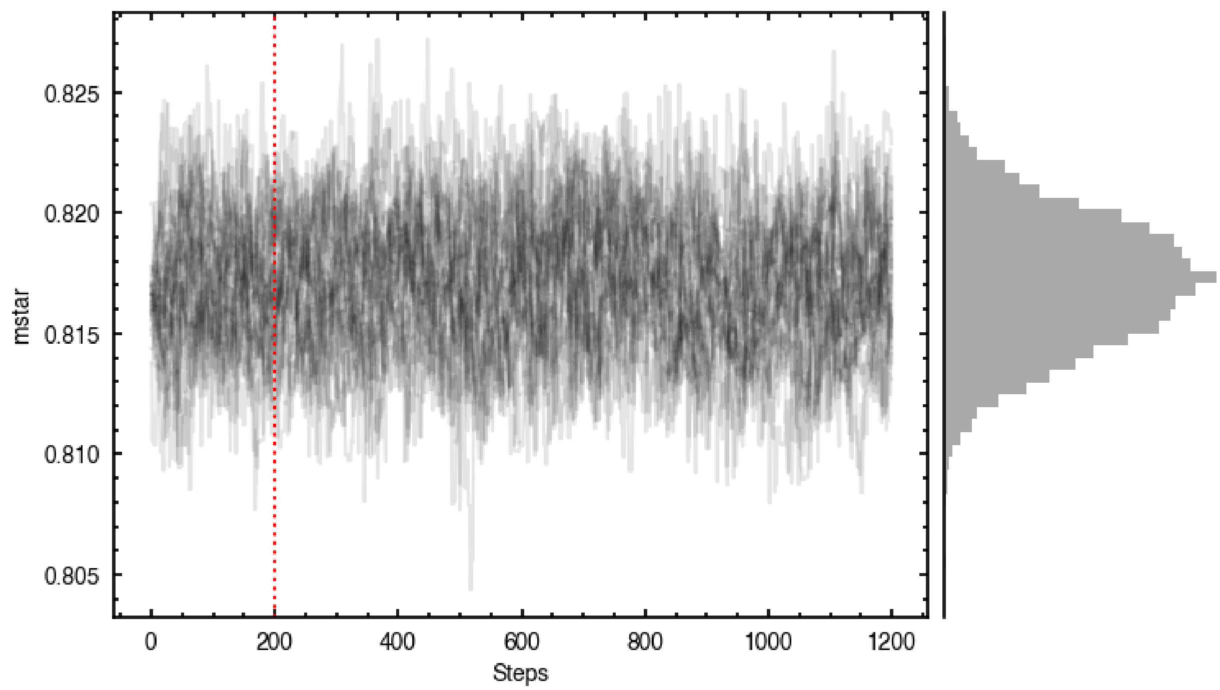
```
Assuming:
        p0 = [x0, y0, PA, mstar, vlsr].
Optimized starting positions:
        p0 = ['1.05e-02', '3.22e-02', '1.51e+02', '8.17e-01', '2.84e+03']
```

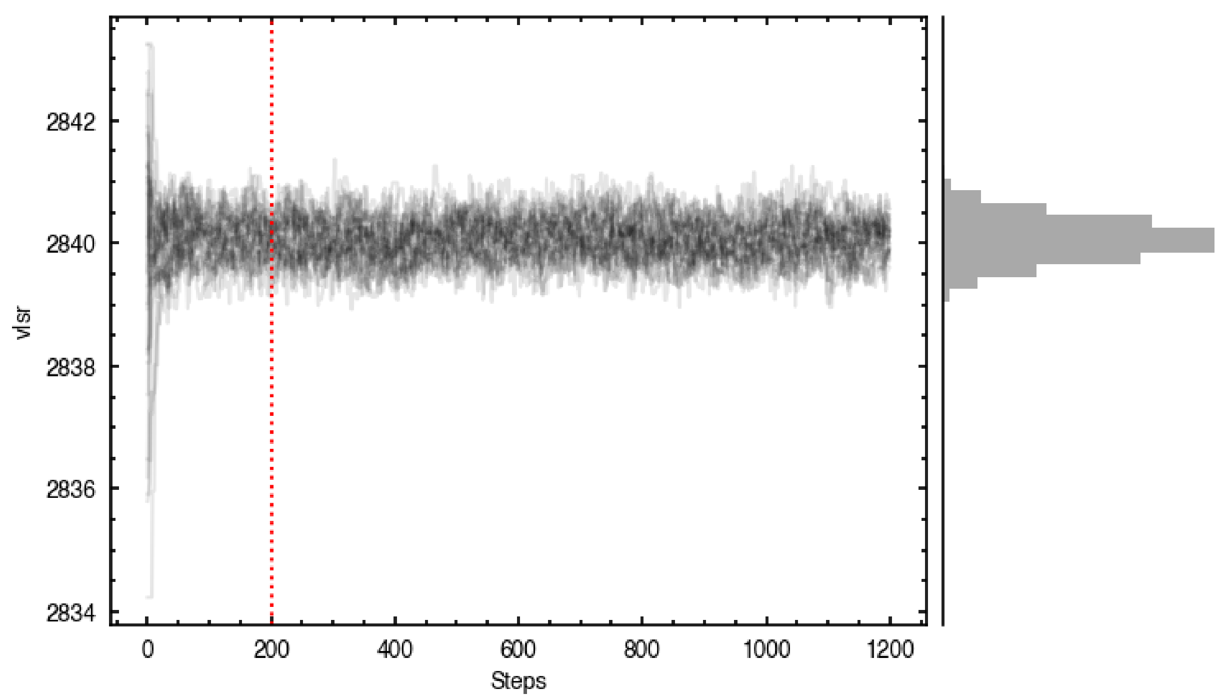
```
100%|| 1200/1200 [00:07<00:00, 161.47it/s]
```

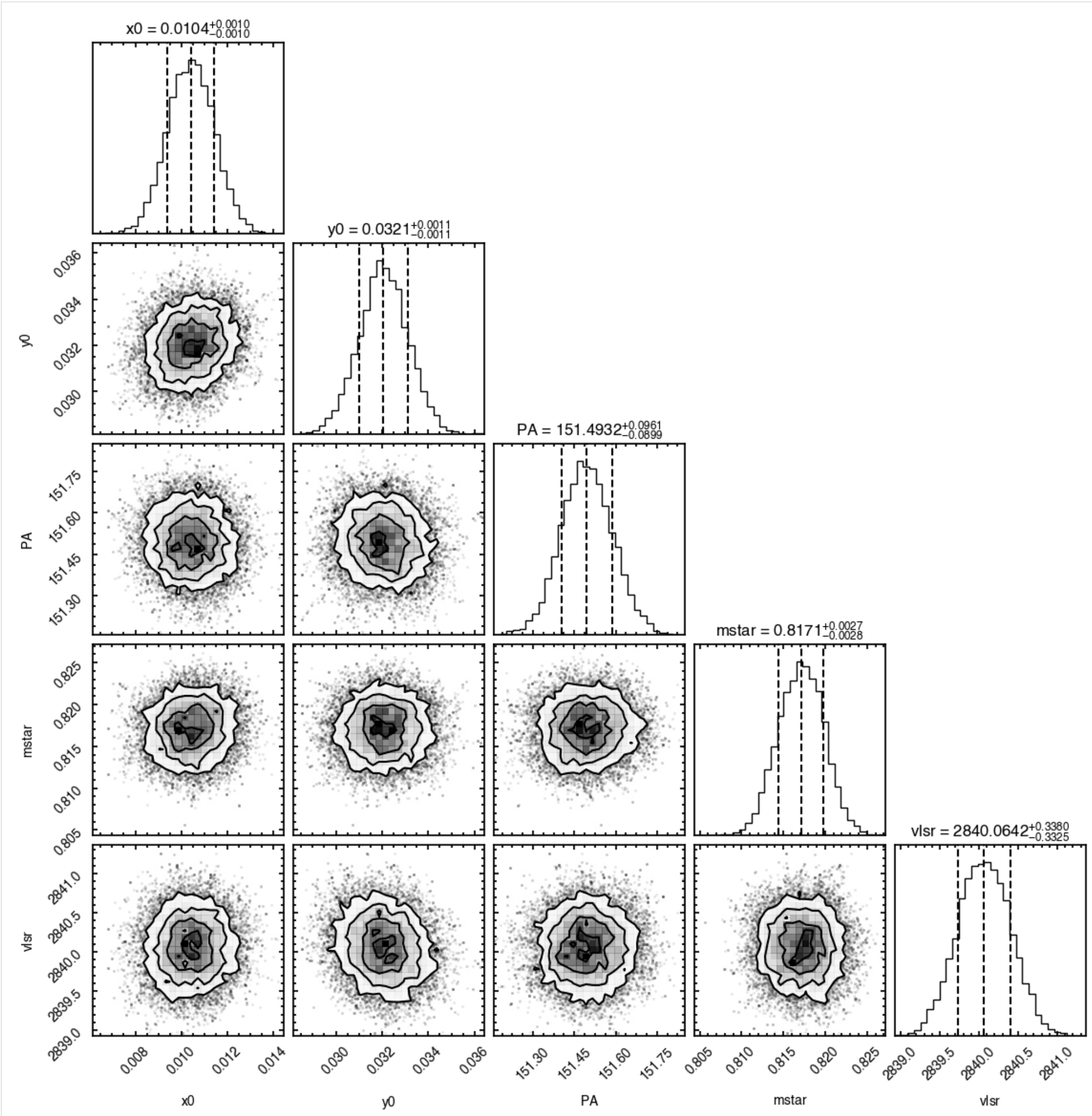


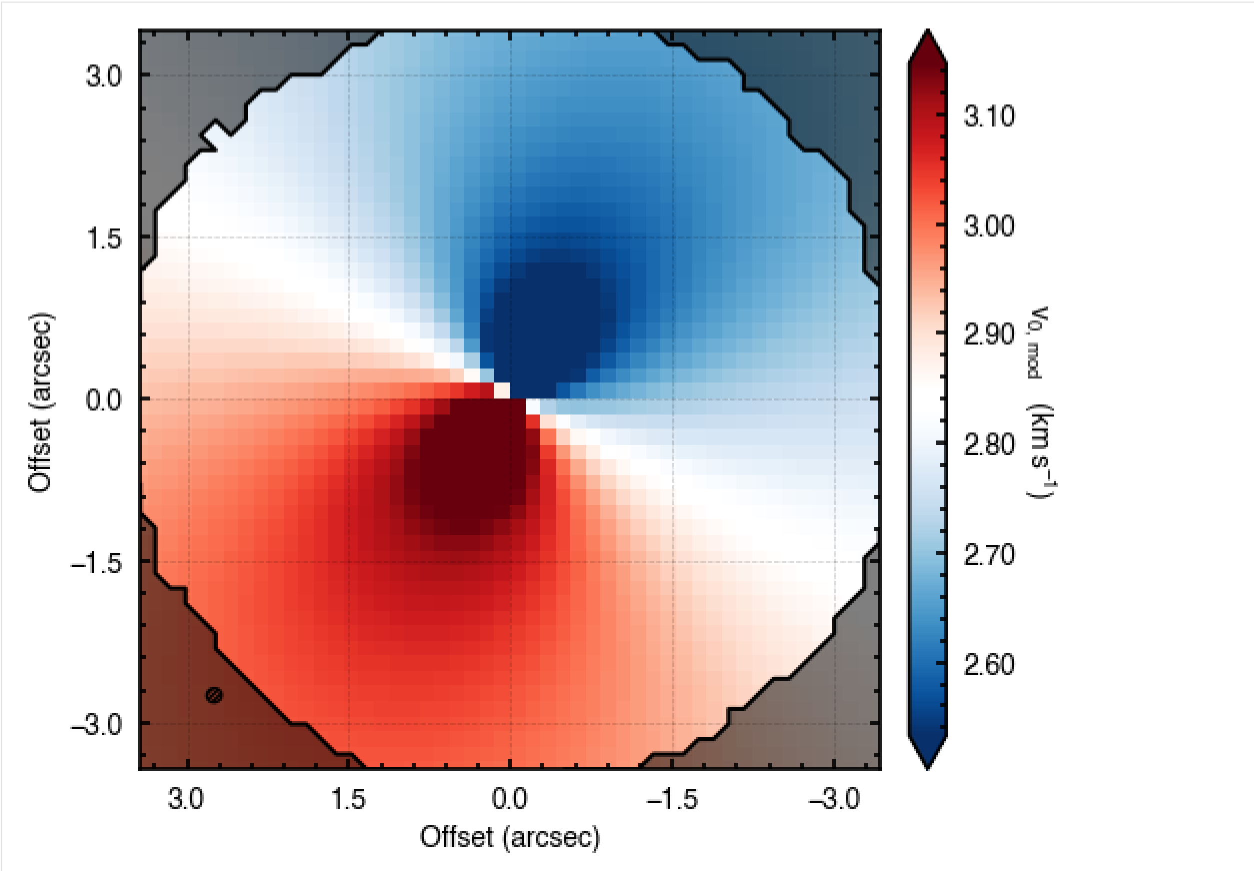

0.036
0.034
0.032
0.030
0.028
y0
0
200
400
600
800
1000
1200
Steps


151.8
151.7
151.6
151.5
151.4
151.3
151.2
151.1
PA
0
200
400
600
800
1000
1200
Steps

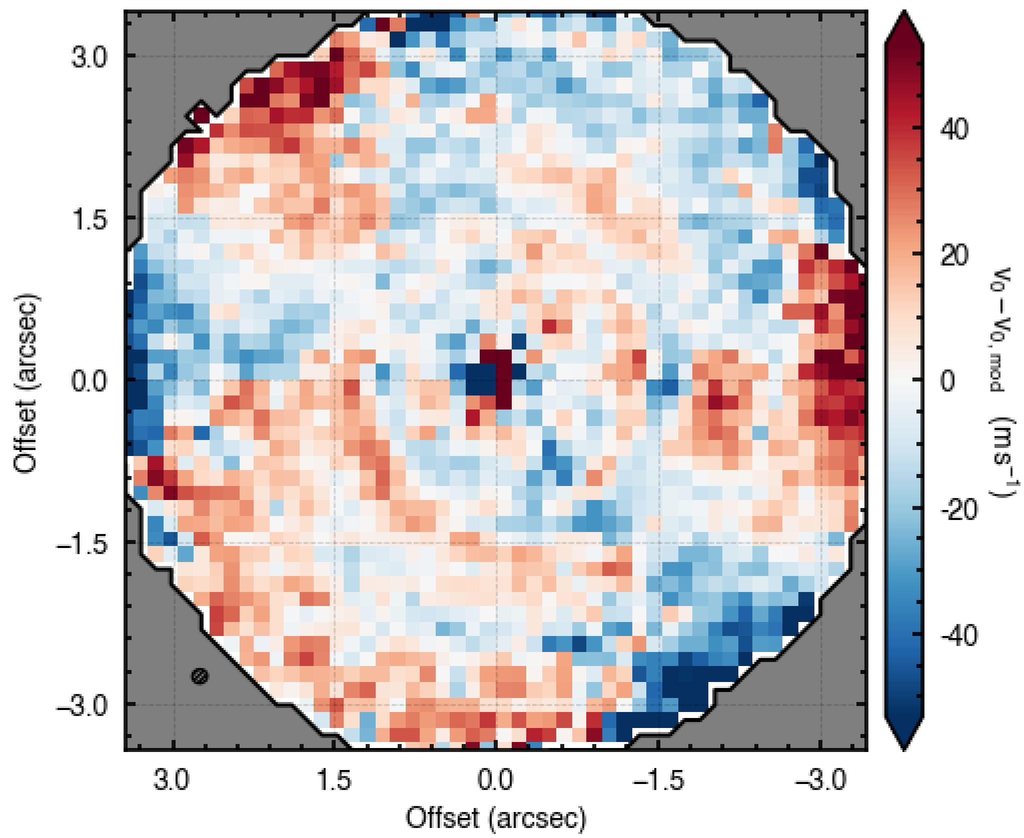

mstar
0.825
0.820
0.815
0.810
0.805
0
200
400
600
800
1000
1200
Steps


vlsr
2842
2840
2838
2836
2834
0
200
400
600
800
1000
1200
Steps


x0 = 0.0104$^{+0.0010}_{-0.0010}$
y0 = 0.0321$^{+0.0011}_{-0.0011}$
PA = 151.4932$^{+0.0961}_{-0.0899}$
mstar = 0.8171$^{+0.0027}_{-0.0028}$
vlsr = 2840.0642$^{+0.3380}_{-0.3325}$
y0
PA
mstar
vlsr
x0
y0
PA
mstar
vlsr



3.0
1.5
0.0
–1.5
–3.0
Offset (arcsec)
3.0
1.5
0.0
–1.5
–3.0
Offset (arcsec)
3.10
3.00
2.90
2.80
2.70
2.60
$v_{0,\,\mathrm{mod}}$ (km s^{-1})



You'll notice this is not substantially faster than the serial attempt because the whole class has to be pickled and read in and out, however we're able to run more walkers taking the same time. The improvement is more noticable for more complex models, either ones with an emission surface, or those with simply more pixels to fit.

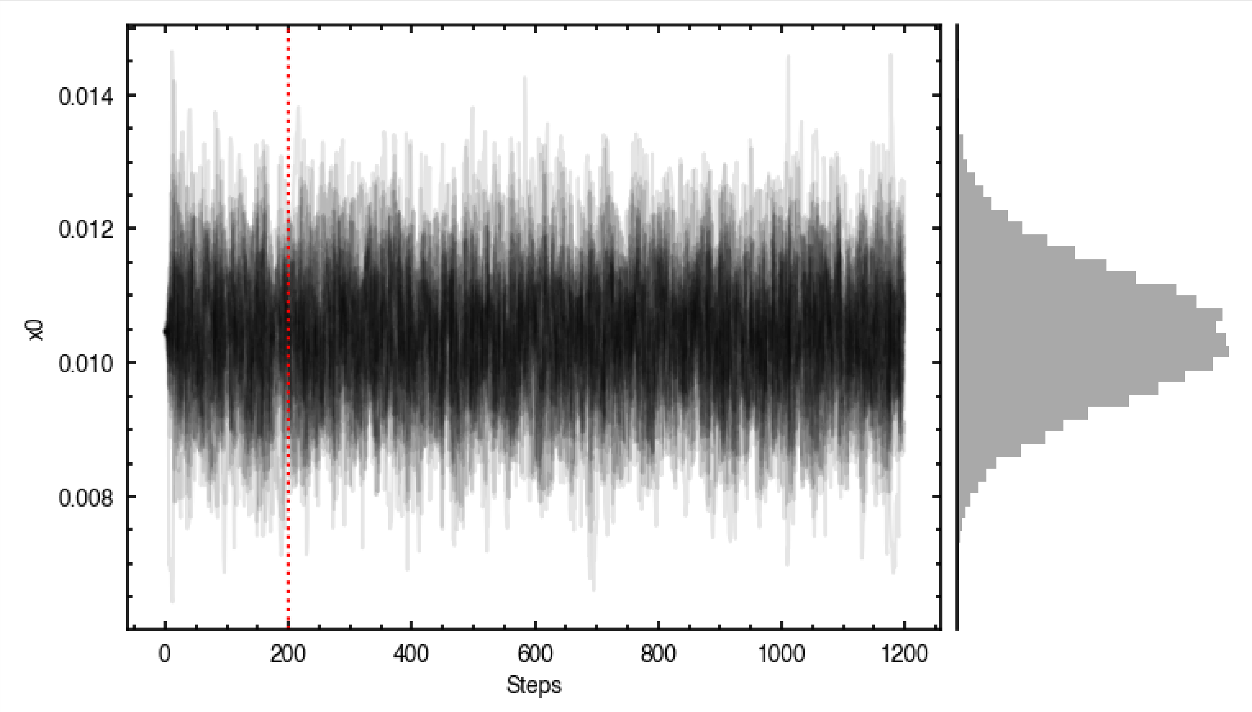
MCMC Packages

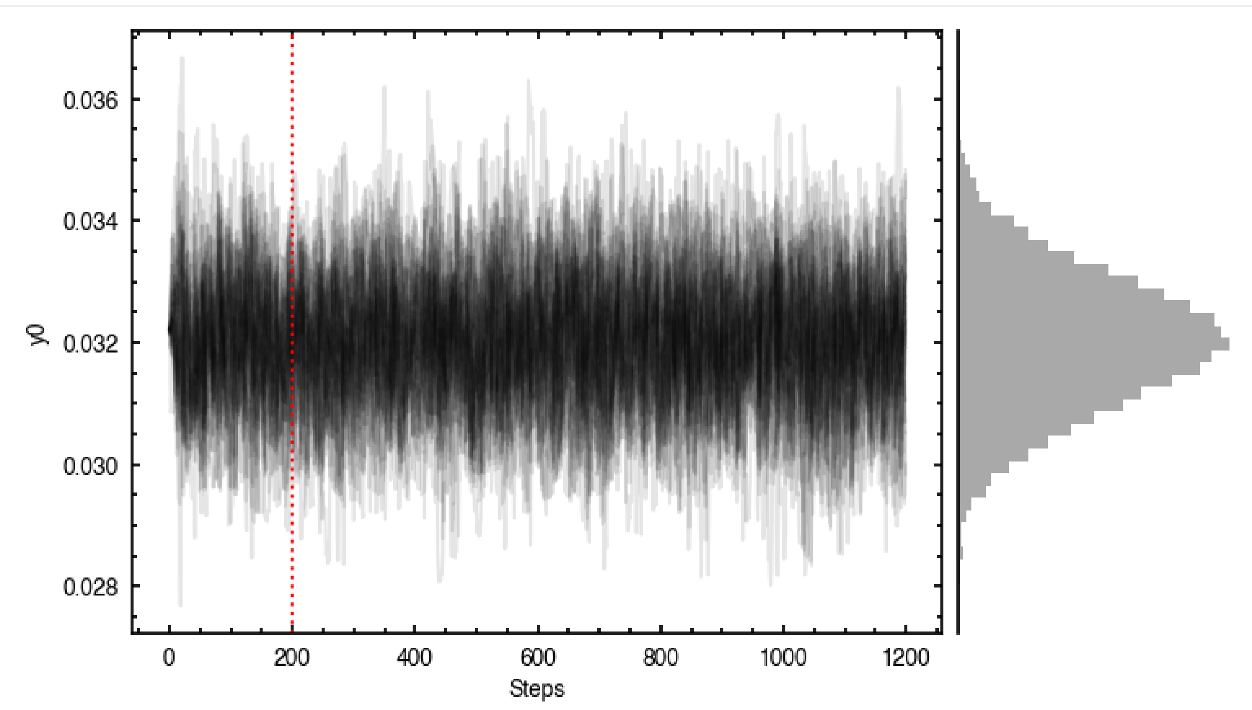
By default `eddy` uses the `EnsemblerSampler` provided by emcee. This can be changed to zeus through the `mcmc` argument. To provide better control for the sampler, keyword arguments can be passed directly to the `EnsembleSampler` instance, for example to change the type of walker move.

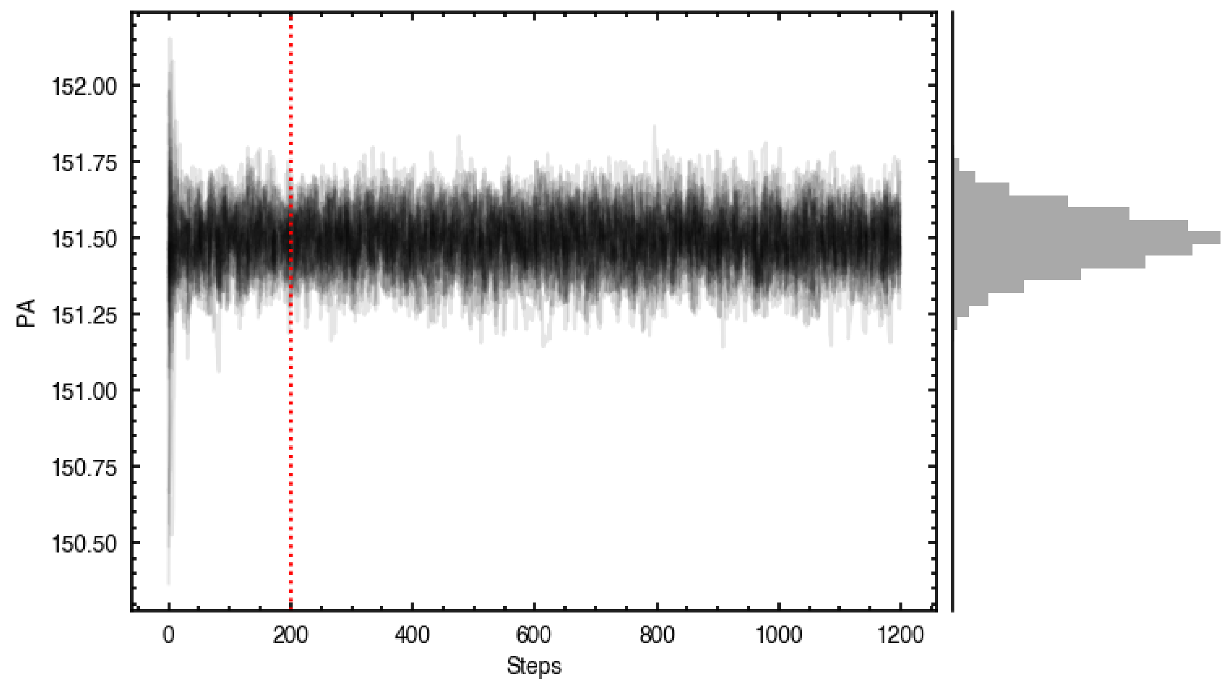
[10]:
```
samples = cube.fit_map(p0=p0, params=params,
                       nwalkers=32, nburnin=200, nsteps=1000,
                       mcmc='zeus')
```

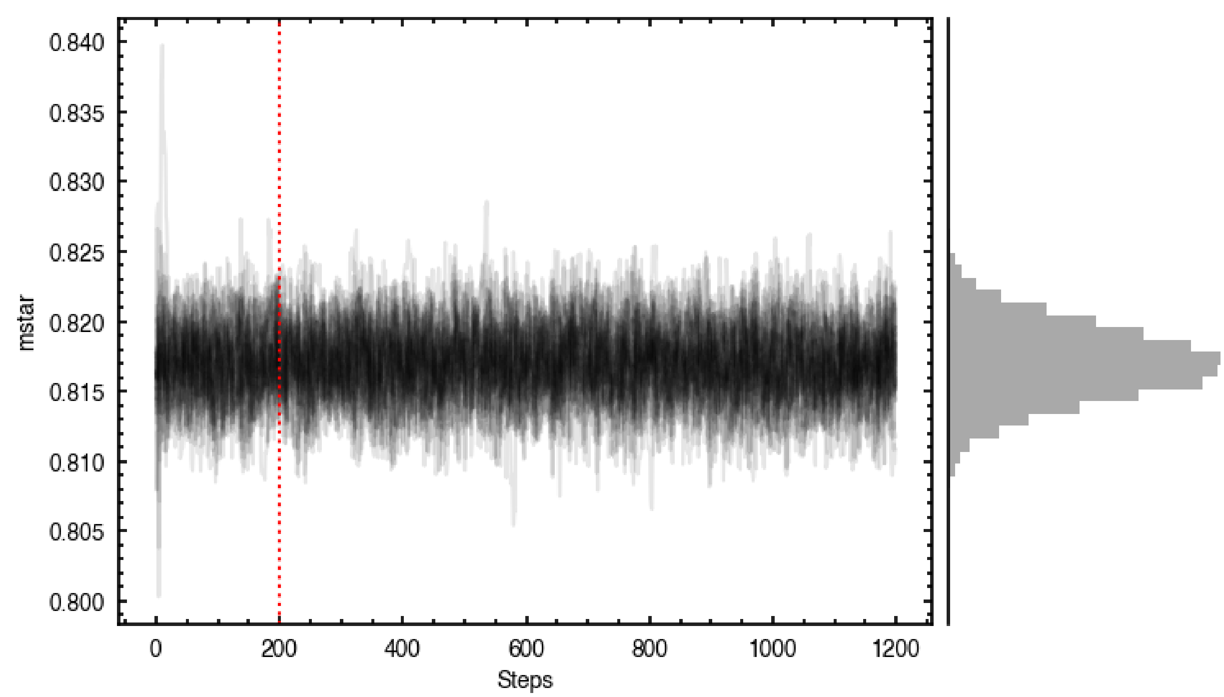
```
Assuming:
        p0 = [x0, y0, PA, mstar, vlsr].
Optimized starting positions:
        p0 = ['1.05e-02', '3.22e-02', '1.51e+02', '8.17e-01', '2.84e+03']
```

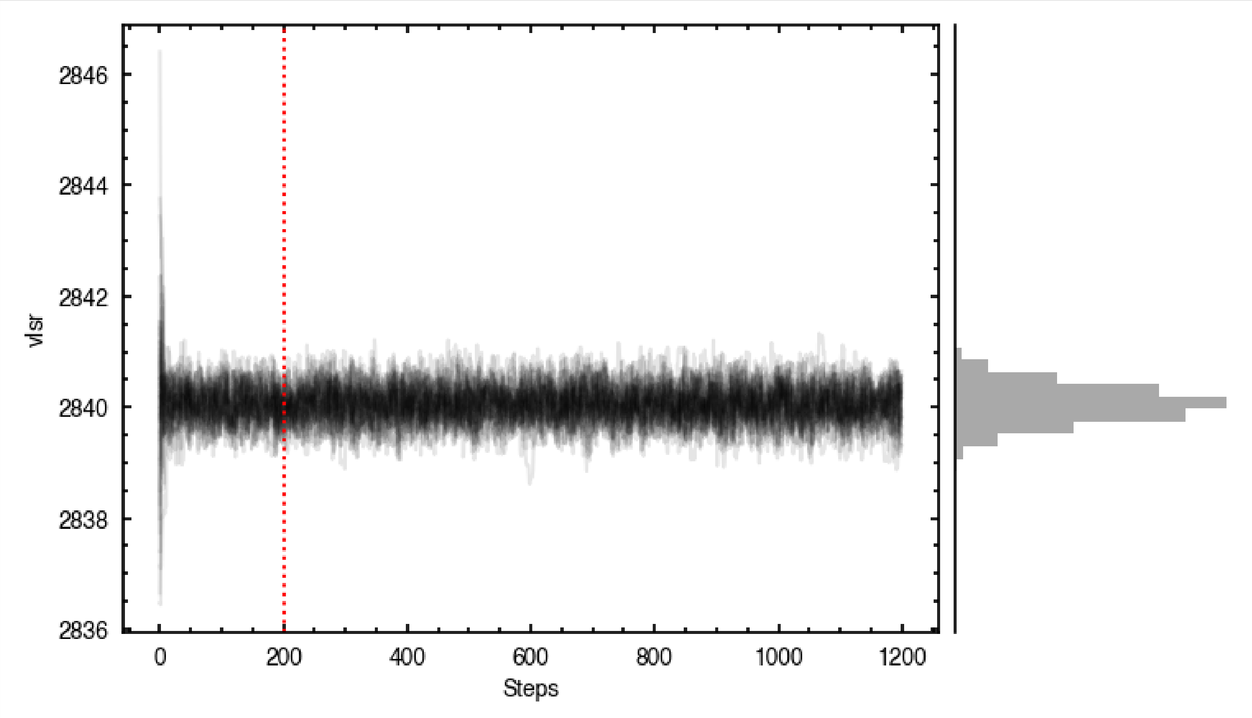
```
Initialising ensemble of 32 walkers...
Sampling progress : 100%|| 1200/1200 [00:33<00:00, 36.06it/s]
```

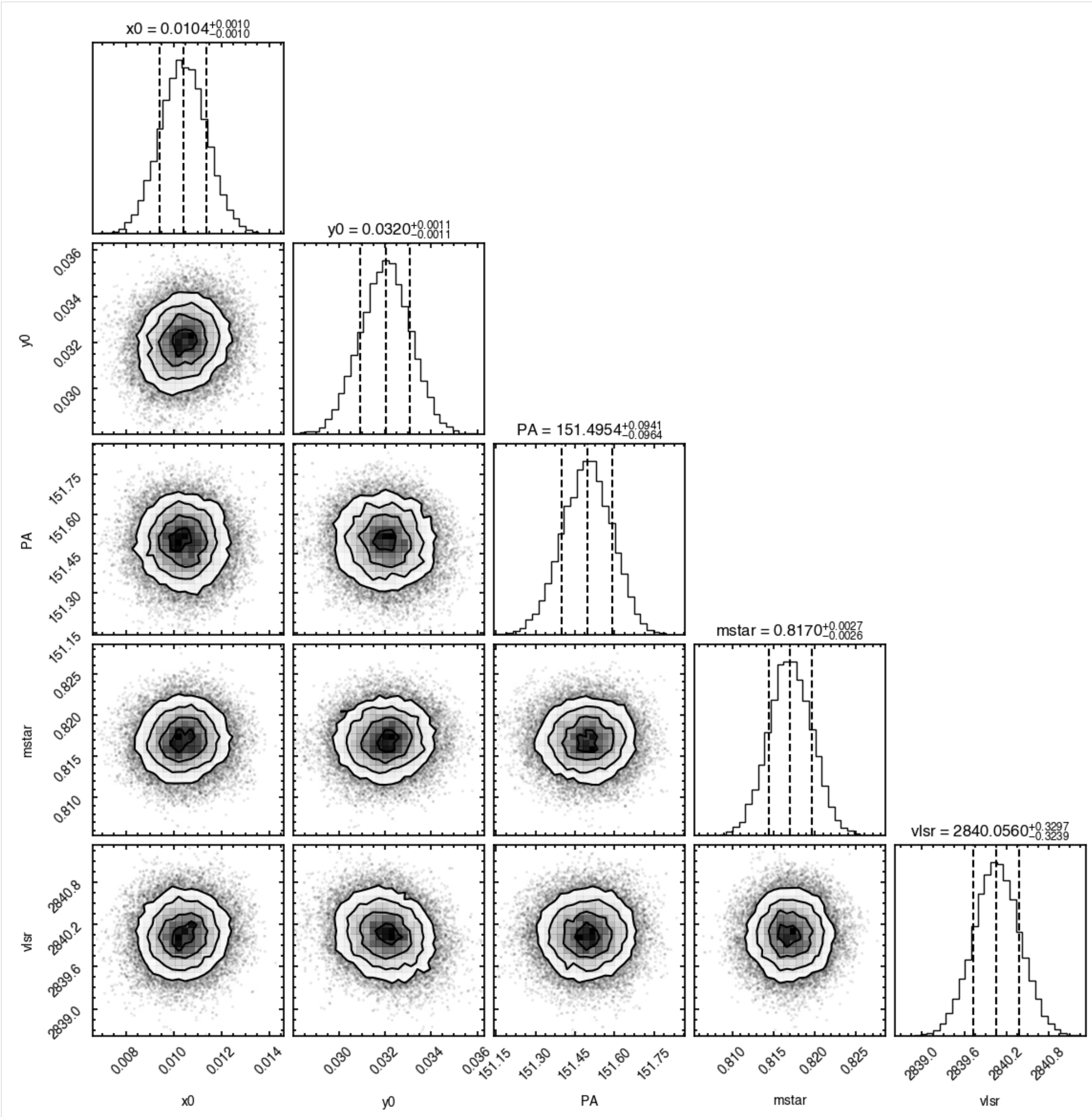

0.014
0.012
0.010
0.008
x0
0
200
400
600
800
1000
1200
Steps

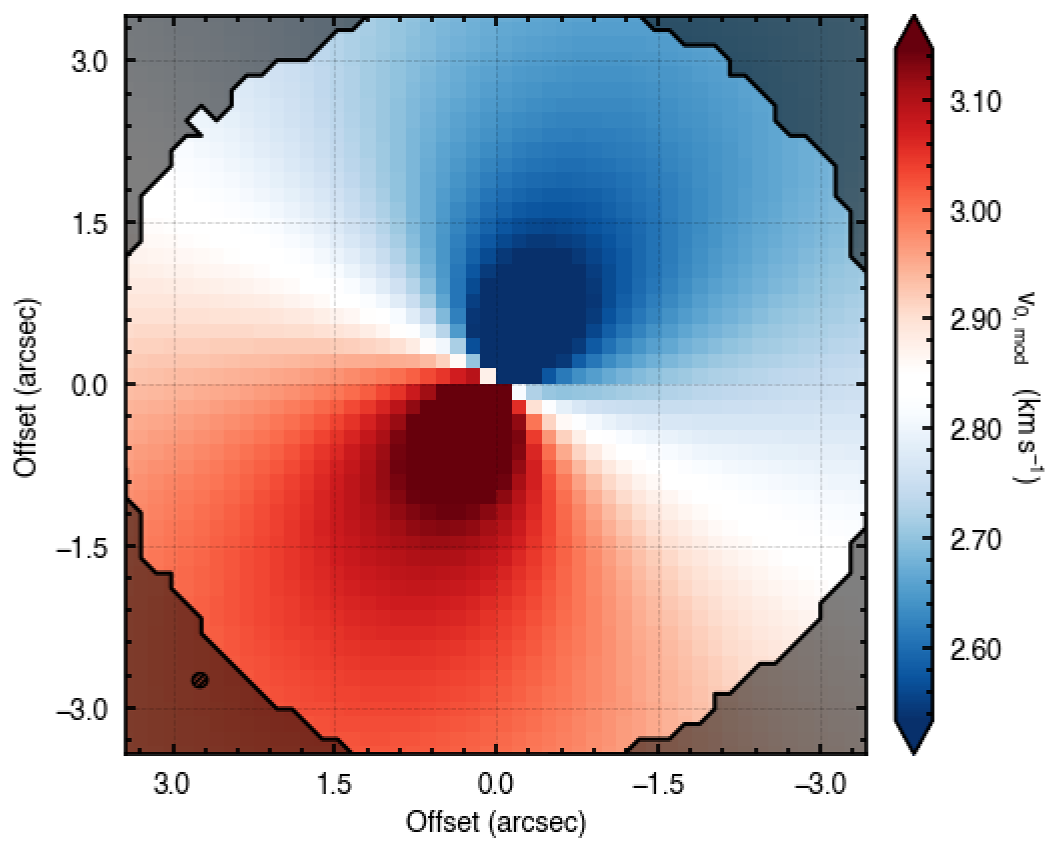

0.036
0.034
0.032
0.030
0.028
y0
0
200
400
600
800
1000
1200
Steps

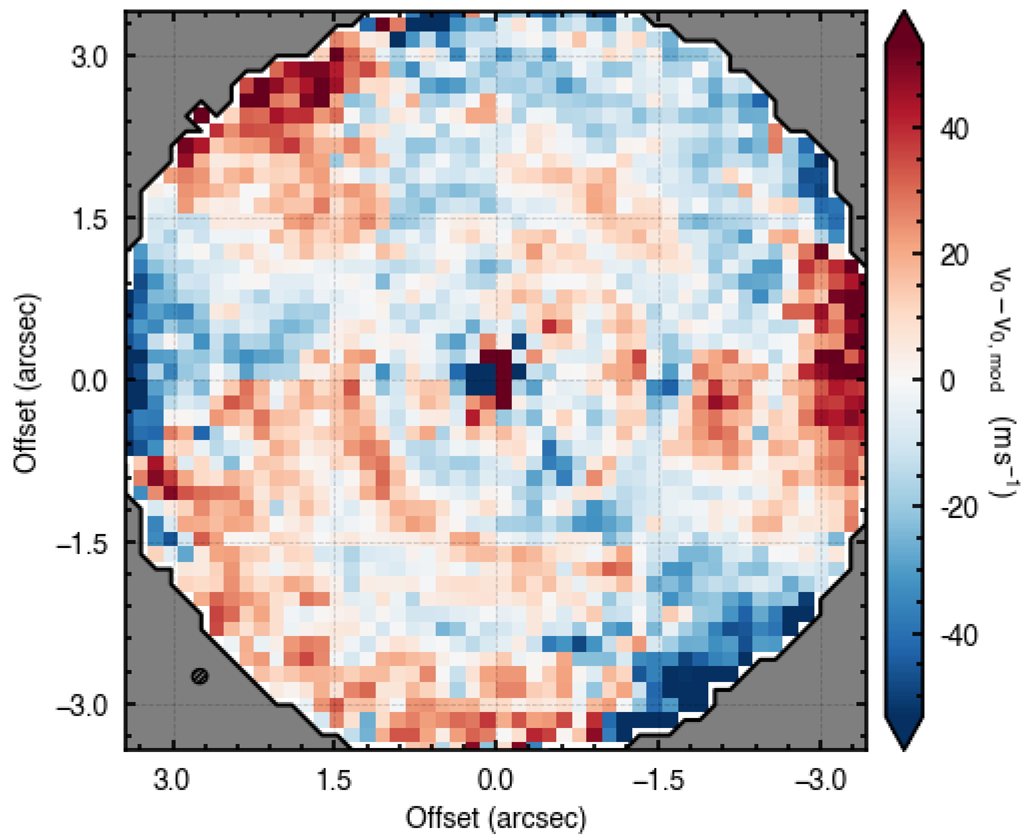

152.00
151.75
151.50
151.25
151.00
150.75
150.50
PA
0
200
400
600
800
1000
1200
Steps


0.840
0.835
0.830
0.825
0.820
0.815
0.810
0.805
0.800
mstar
0
200
400
600
800
1000
1200
Steps


2846
2844
2842
2840
2838
2836
vlsr
0
200
400
600
800
1000
1200
Steps


x0 = 0.0104$^{+0.0010}_{-0.0010}$
y0 = 0.0320$^{+0.0011}_{-0.0011}$
PA = 151.4954$^{+0.0941}_{-0.0964}$
mstar = 0.8170$^{+0.0027}_{-0.0026}$
vlsr = 2840.0560$^{+0.3297}_{-0.3239}$
x0
y0
PA
mstar
vlsr


3.0
1.5
0.0
–1.5
–3.0
Offset (arcsec)
3.0
1.5
0.0
–1.5
–3.0
Offset (arcsec)
3.10
3.00
2.90
2.80
2.70
2.60
$v_{0,\,\mathrm{mod}}$ (km s^{-1})



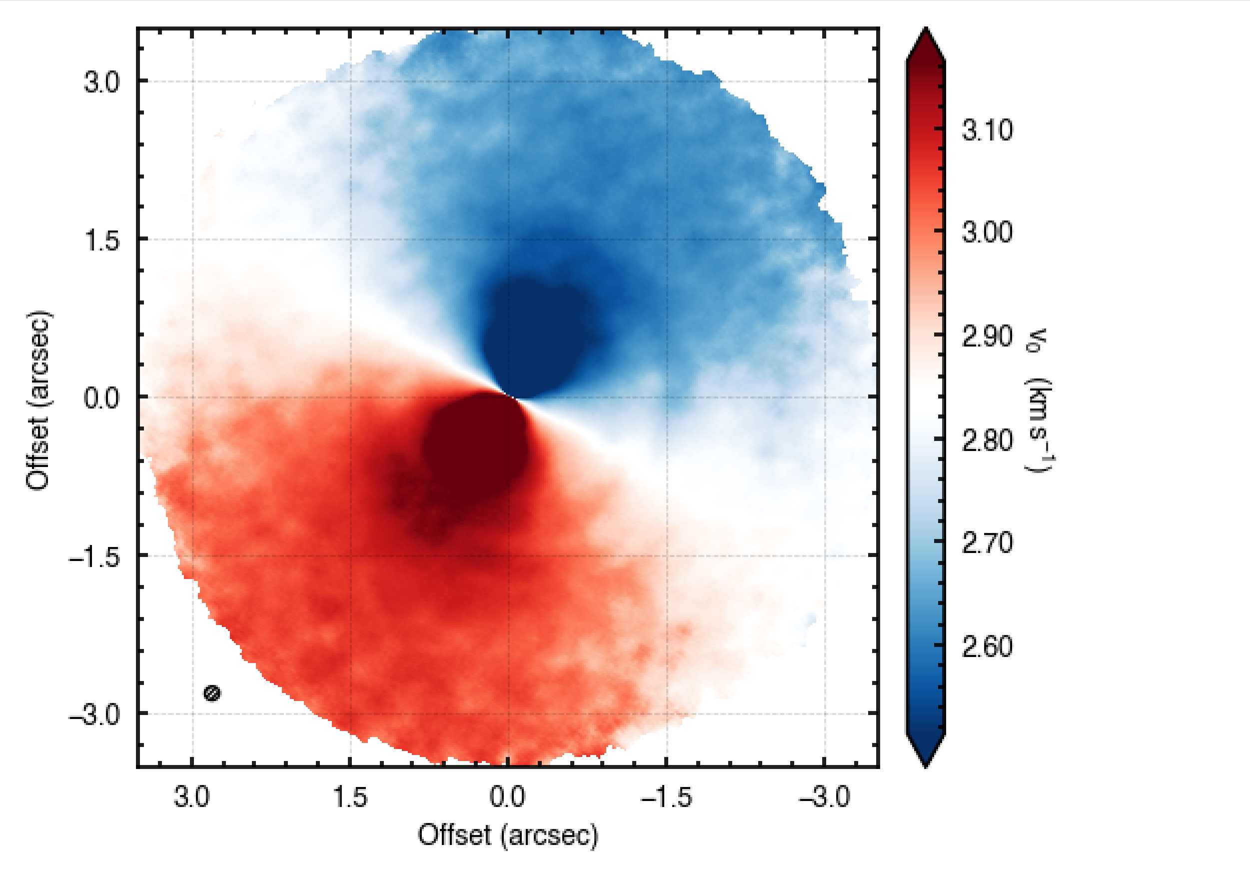
2.2.2 Working with the Model

In the previous section we downsampled the cube to perform the fitting with spatially independent pixels (and to speed up the sampling). Once we have determined the samples, we can use another instance of a `rotationmap` without any downsampling to explore the models.

First, reload the data without any downsampling.

[11]:
```
cube = rotationmap(path='TWHya_CO_cube_v0.fits', FOV=7.0)
cube.plot_data()
```

```
Assuming uncertainties in TWHya_CO_cube_dv0.fits.
```



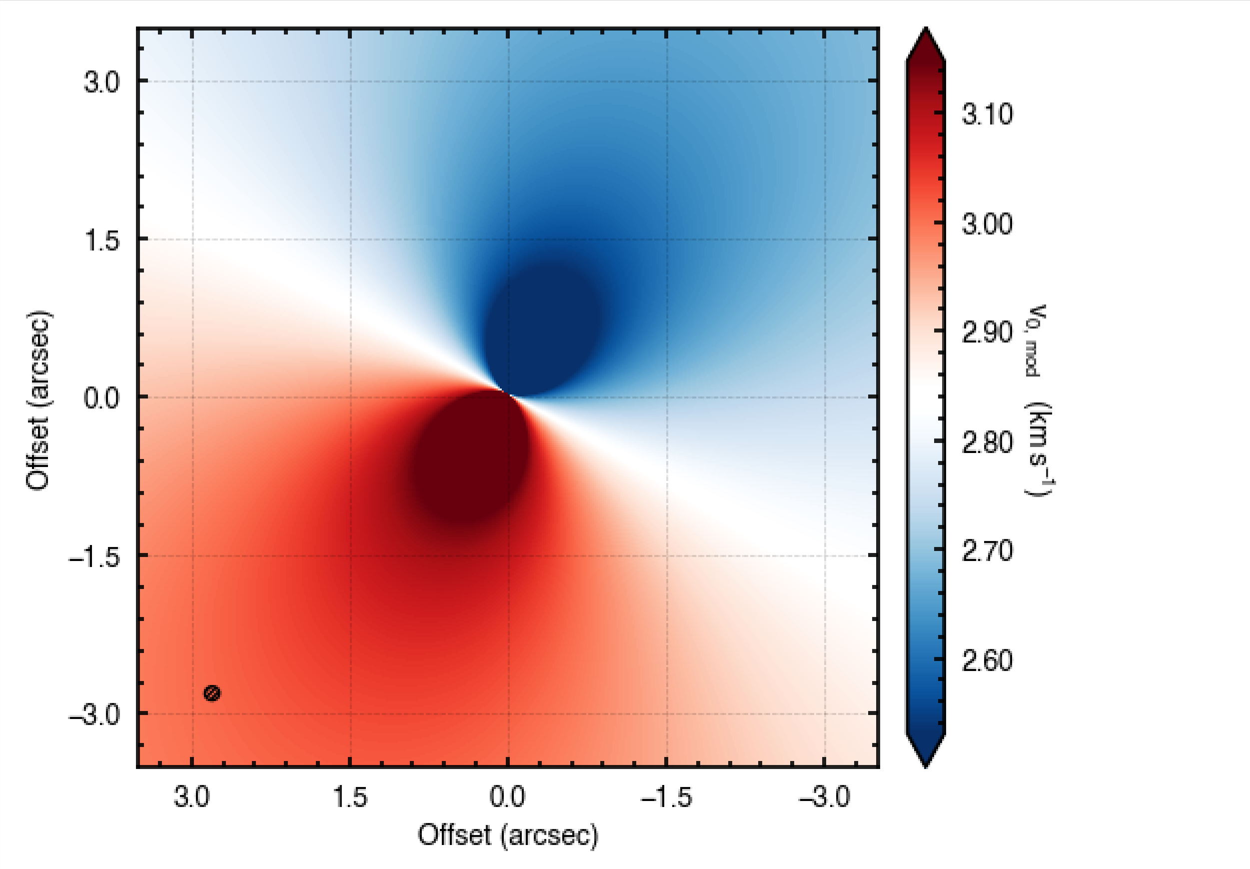
Evaluating Models

To turn the samples returned from `fit_map` into a model, we use the `evaluate_models` function.

[12]:
```
model = cube.evaluate_models(samples=samples, params=params)
```

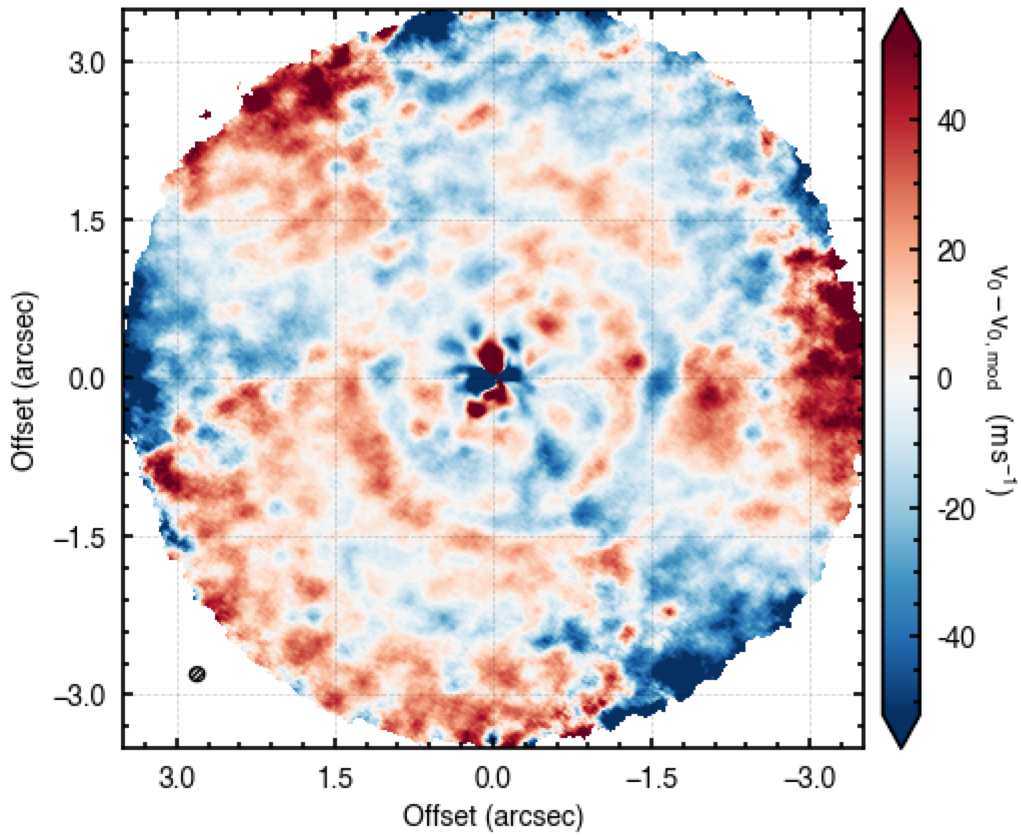
This can then be plotted through the `plot_model` convenience function.

[13]:
```
cube.plot_model(model=model)
```



Similarly, we can plot the residuals with respect to the original data through `plot_model_residuals`.

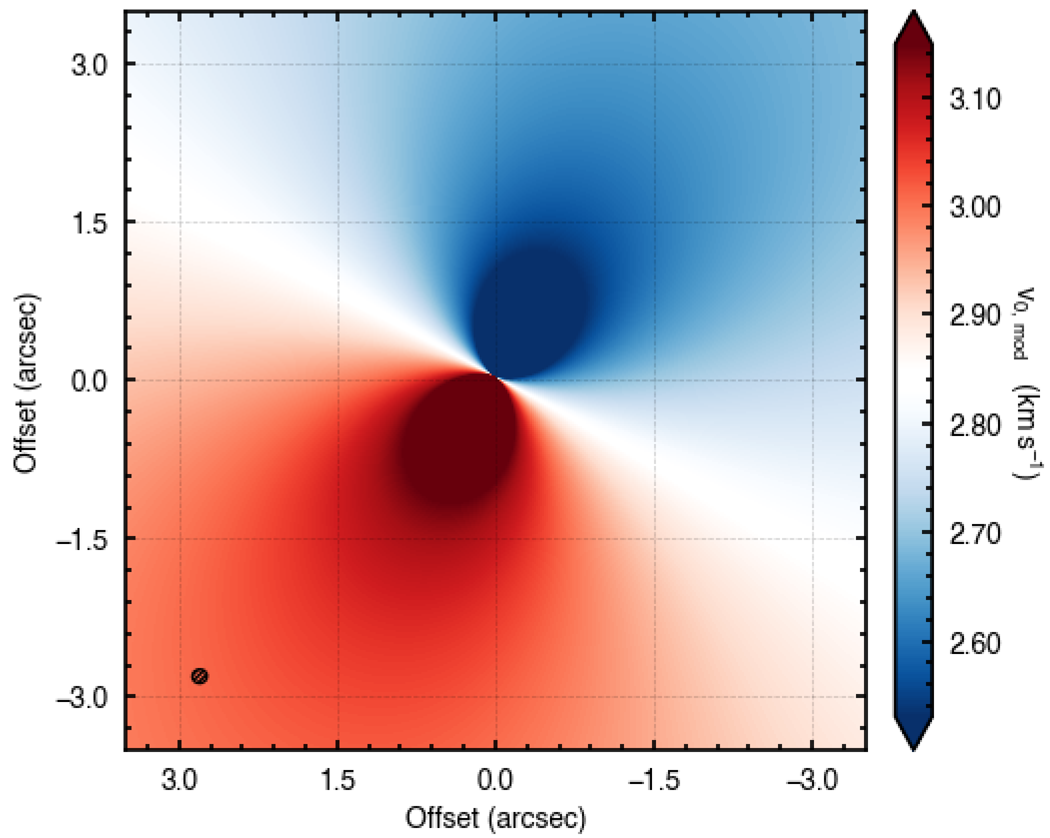
[14]:
```
cube.plot_model_residual(model=model)
```



Both these functions can also directly take `samples` and `params` to skip the `evaluate_models` step.

However, this approach, which is used to make the plots generated with `fit_map`, is that it is assumed that the model parameters are independent such that taking the median of each posterior distribution provides the 'best-fit' model. Often parameteres are correlated and it is better to generate several realizations of the model from the posterior samples and then combine them. This can be achieved with the `evaluate_models` function and the `draws` argument.

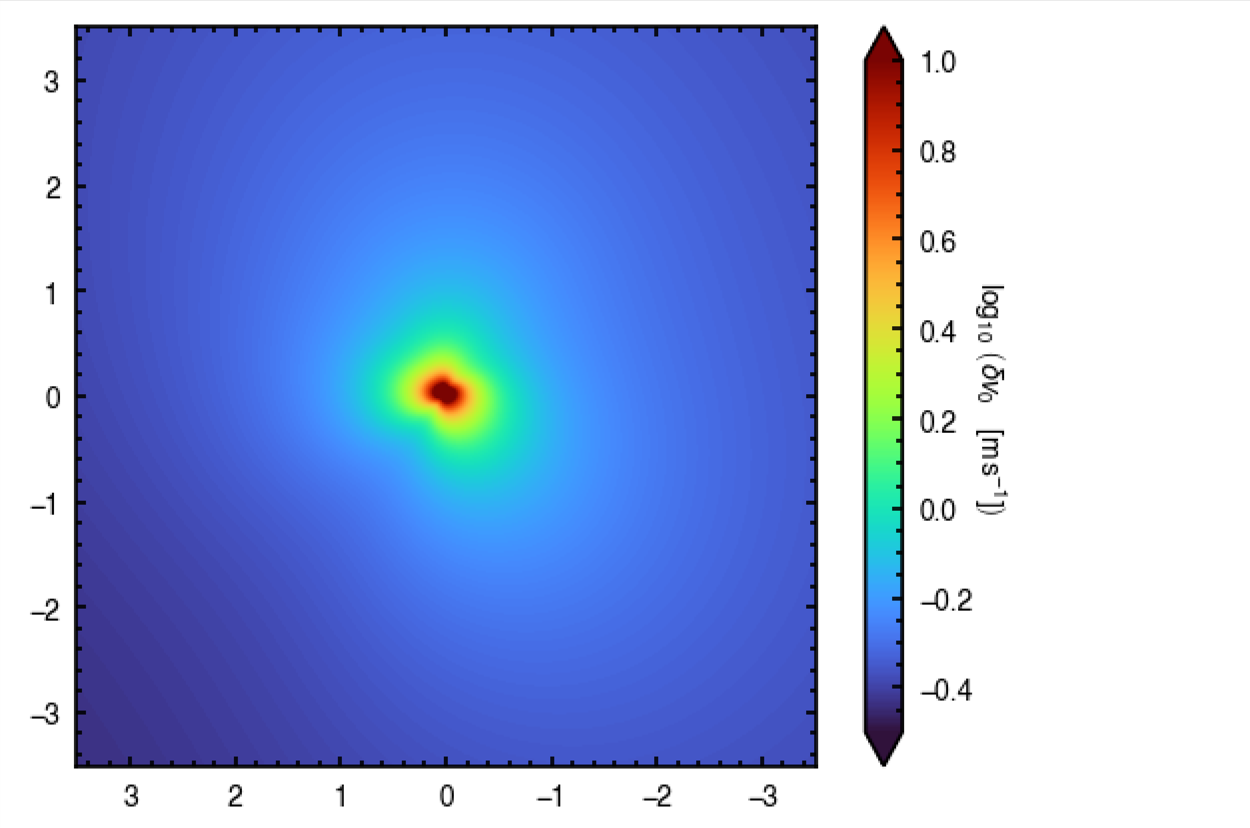
[15]:
```
model = cube.evaluate_models(samples=samples, params=params, draws=100)
cube.plot_model(model=model)
```



Here the `draws` argument tells the function how many random draws to take which are then averaged down to a single value. If `draws` is less than 1, this represents the percentile of the posteriors to use (e.g., `draws=0.5` will use the median values from all posterior distributions).

In addition, the `evaluate_models` function also allows for a user-defined function to collapse the multiple models, `collapse_func`. This is particularly useful if you want to see how much variability there is in the sample of models by looking at the standard deviation of the set. Note here how you can provide extra kwargs to the `imshow` function to display the data.

[16]:
```
scatter = cube.evaluate_models(samples=samples, params=params, draws=100, collapse_
↪func=np.std)

fig, ax = plt.subplots()
im = ax.imshow(np.log10(scatter), origin='lower', extent=cube.extent, cmap='turbo',
↪vmin=-0.5, vmax=1.0)
cb = plt.colorbar(im, extend='both')
cb.set_label(r'$\log_{10}\,\left(\delta v_0 \quad [{\rm m\,s^{-1}}] \right)$',
             rotation=270, labelpad=13)
```



From this you can clearly see that there are small variations between the models in the posterior samples, but these are typically confined to the inner region of the disk.

Saving Models

If you want to save the model for use later, you can export it as a FITS cube, copying the header information from the attached FITS cube. As with `evaluate_models`, `save_model` accepts either the `samples` and `params`, or a pre-evaluated `model`.

[17]:
```
cube.save_model(model=model)
```

NOTE If the datacube has been clipped down using the `FOV` argument when loading the cube, the model will only populate pixels in that subregion. The rest will be left as `NaN`.

2.3 2 - Disks with Elevated Emission Surfaces

This Notebook deals with disks where the vertical structure can clearly be resolved. This is fairly common now that ALMA observations of bright ^{12}CO emission are routinely achieving angular resolutions of a few hundreds of milliarcseconds.

```
[1]: from multiprocessing import Pool
     import matplotlib.pyplot as plt
     from eddy import rotationmap
```

2.3.1 HD 163296 - A Geometrically Thick Disk

For this tutorial we will look at the disk around HD 163296. We'll use data from the DSHARP project, with the data described in Isella et al. (2018) and available here. There are many different ways of collapsing a cube to a velocity map and with various different programs. For now, we'll use ones collapsed using bettermoments and are available from the eddy Dataverse.

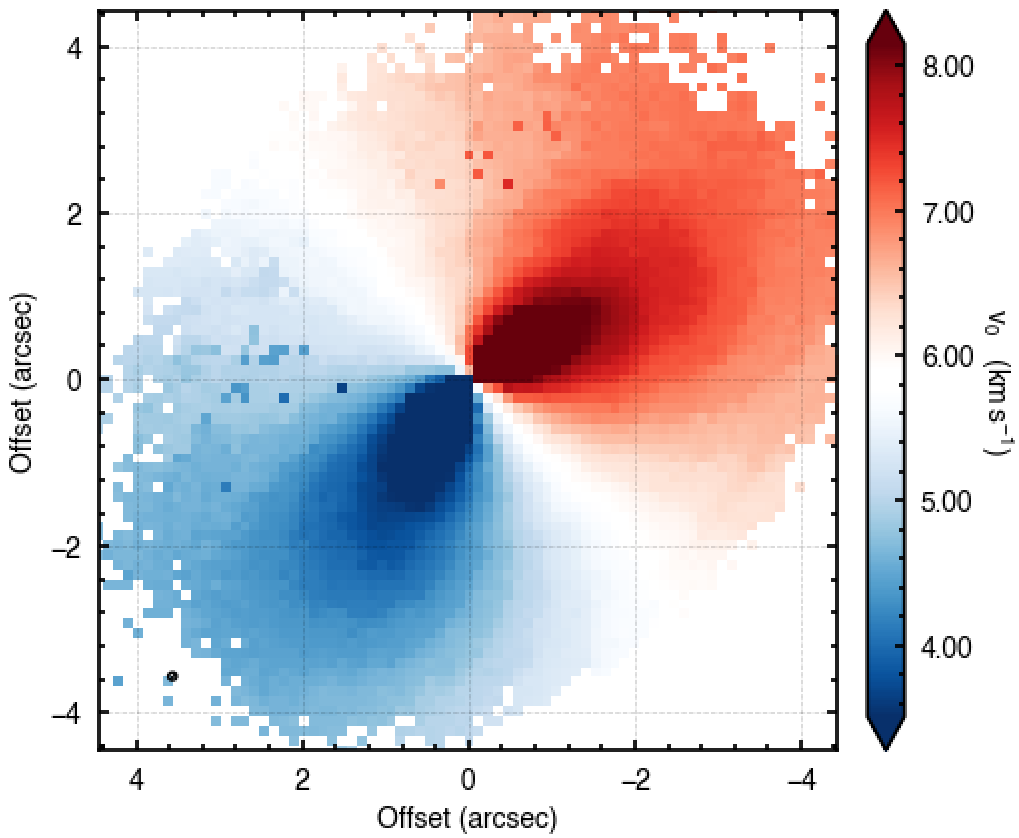
We can directly download the data if you don't already have it.

```
[2]: import os
     if not os.path.exists('HD163296_CO_v0.fits'):
         !wget -O HD163296_CO_v0.fits -q https://dataverse.harvard.edu/api/access/datafile/
     ↪:persistentId?persistentId=doi:10.7910/DVN/C2ZUNO/AWCSZR
     if not os.path.exists('HD163296_CO_dv0.fits'):
         !wget -O HD163296_CO_dv0.fits -q https://dataverse.harvard.edu/api/access/
     ↪datafile/:persistentId?persistentId=doi:10.7910/DVN/C2ZUNO/NGOW49
```

Load up the data and inspect it.

```
[3]: cube = rotationmap(path='HD163296_CO_v0.fits',
                        downsample='beam',
                        FOV=9.0)
     cube.plot_data()
```

```
Assuming uncertainties in HD163296_CO_dv0.fits.
```



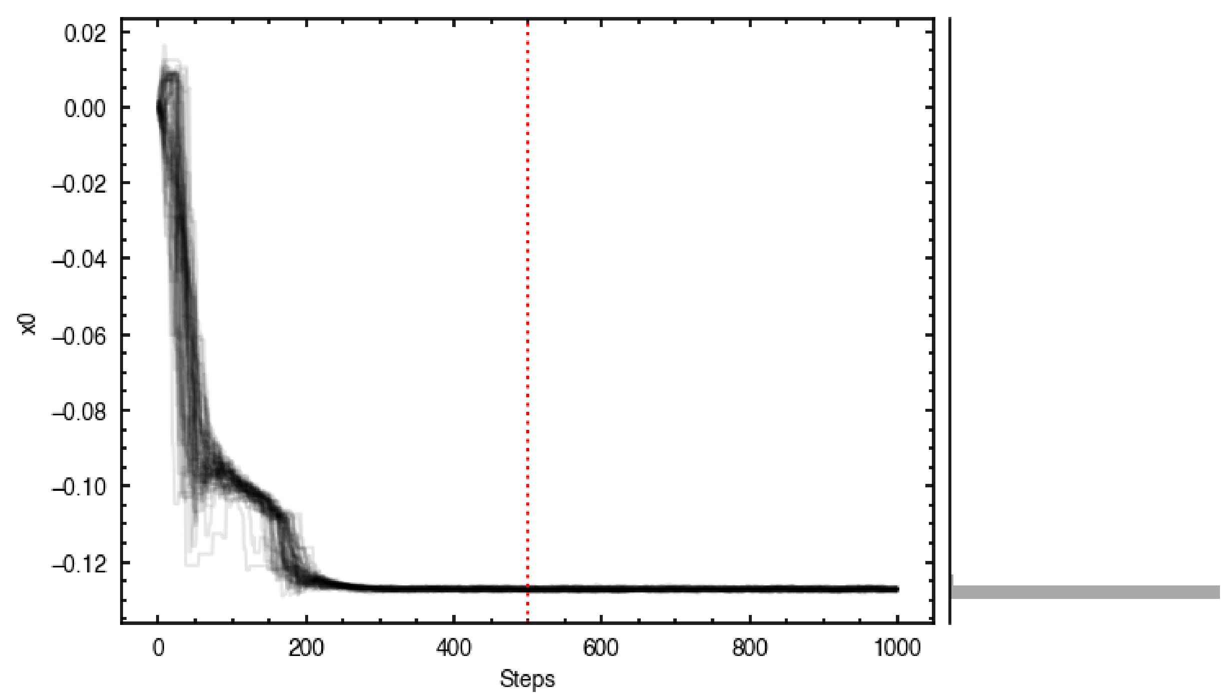
A 2D Fit

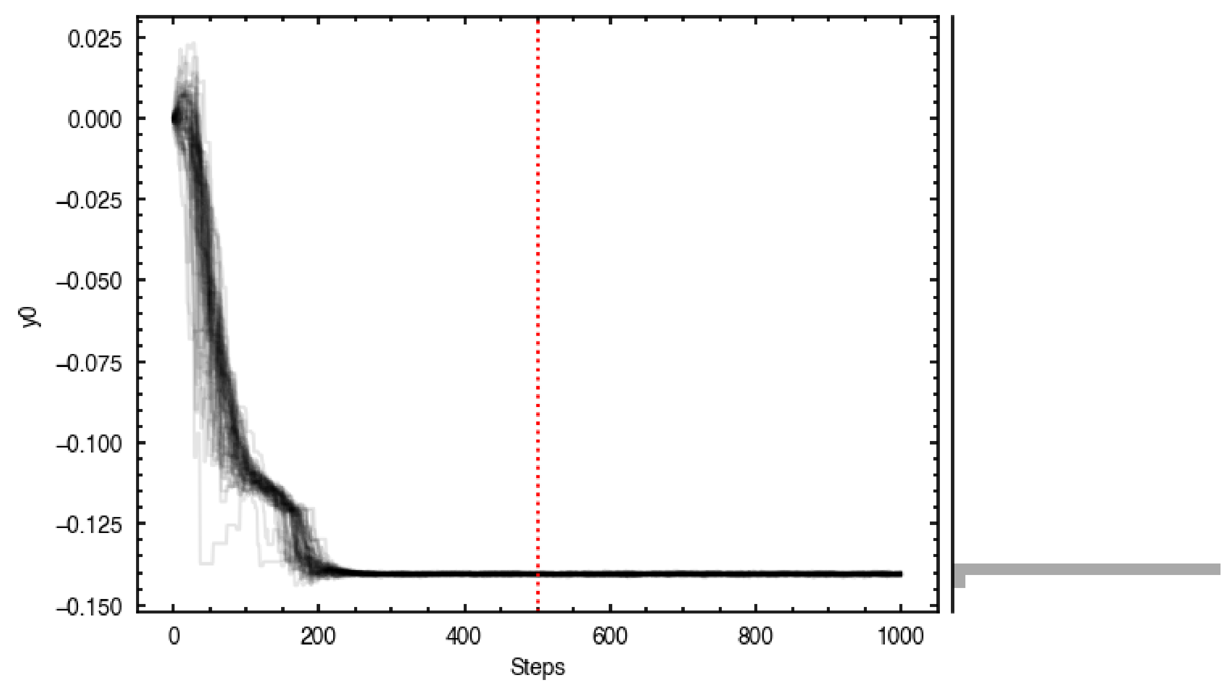
To begin, we can fit a simple 2D model to the data, just as we did for TW Hya. Note here we have used `optimize=False` to skip the initial optimization of the free parameters. As you will see, this is because the model is actually a poor description of the data and the current implementation will almost certaintly fail.

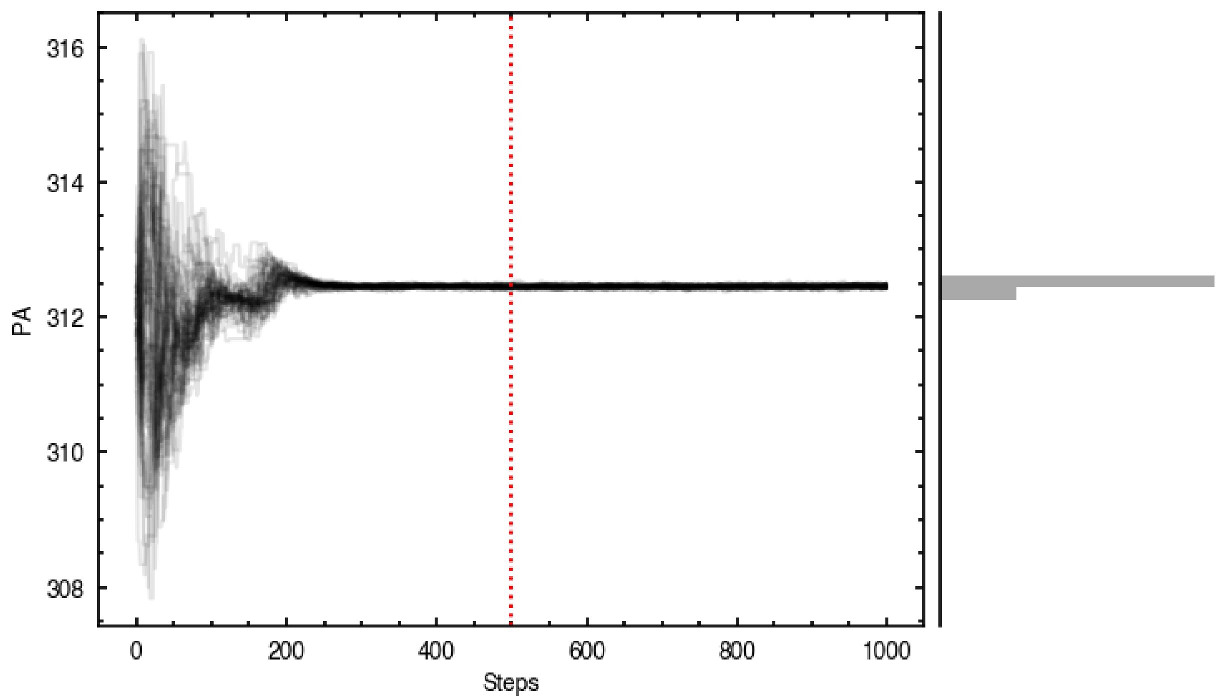
[4]:
```
params = {}

params['x0'] = 0
params['y0'] = 1
params['PA'] = 2
params['mstar'] = 3
params['vlsr'] = 4

p0 = [0.0, 0.0, 312., 2.0, 5.7e3]

params['inc'] = 46.7
params['dist'] = 101.0

samples = cube.fit_map(p0=p0, params=params, optimize=False,
                       nwalkers=64, nburnin=500, nsteps=500)
```

```
Assuming:
        p0 = [x0, y0, PA, mstar, vlsr].
```

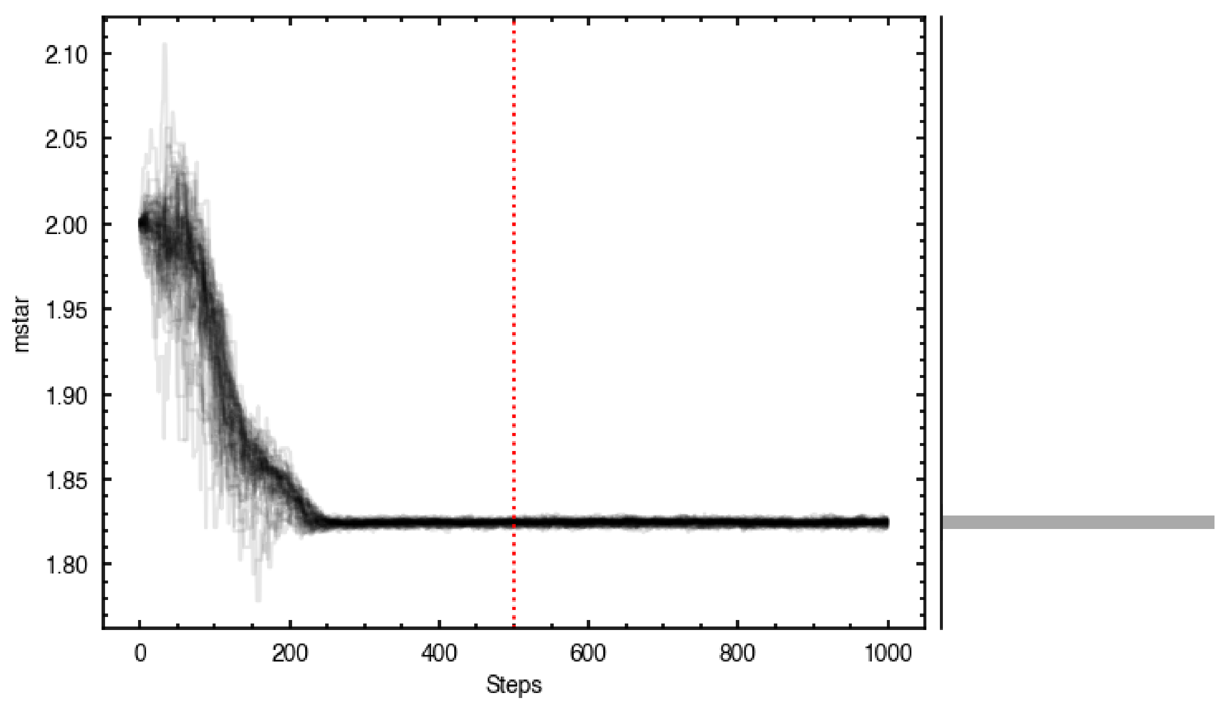
```
100%|| 1000/1000 [00:20<00:00, 49.34it/s]
```


0.02
0.00
−0.02
−0.04
−0.06
−0.08
−0.10
−0.12
x0
0
200
400
600
800
1000
Steps

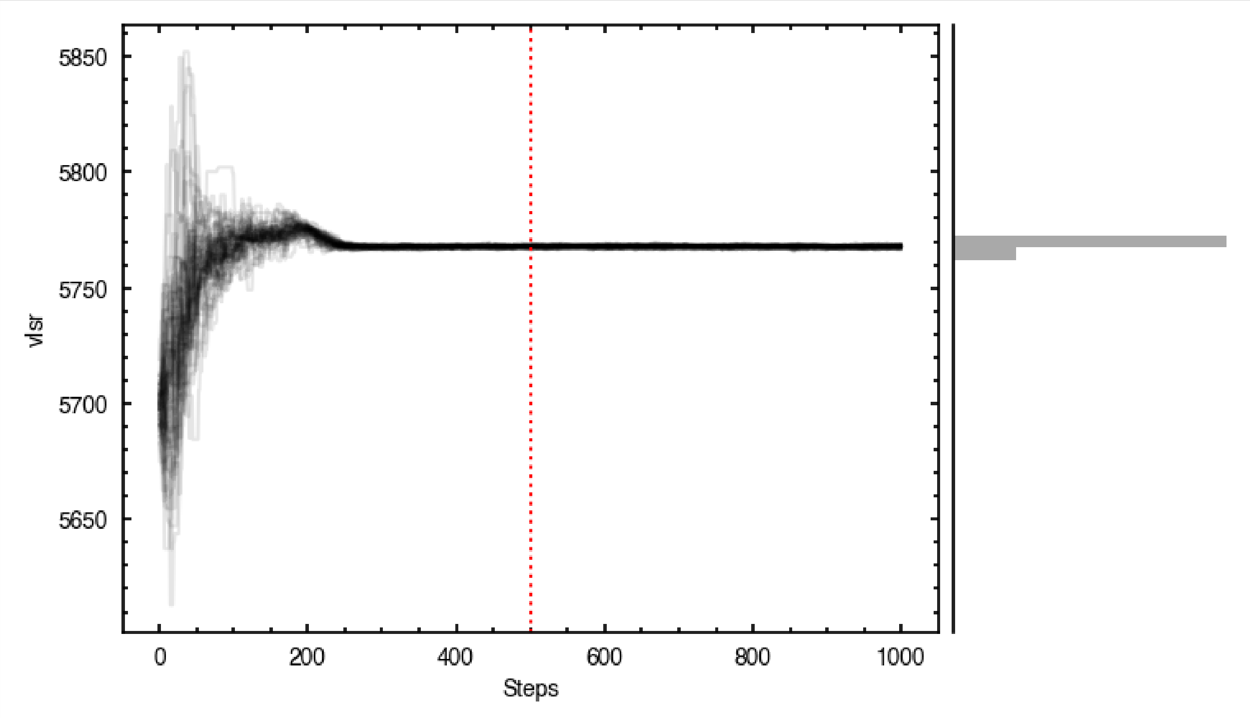

0.025
0.000
−0.025
−0.050
−0.075
−0.100
−0.125
−0.150
y0
0
200
400
600
800
1000
Steps

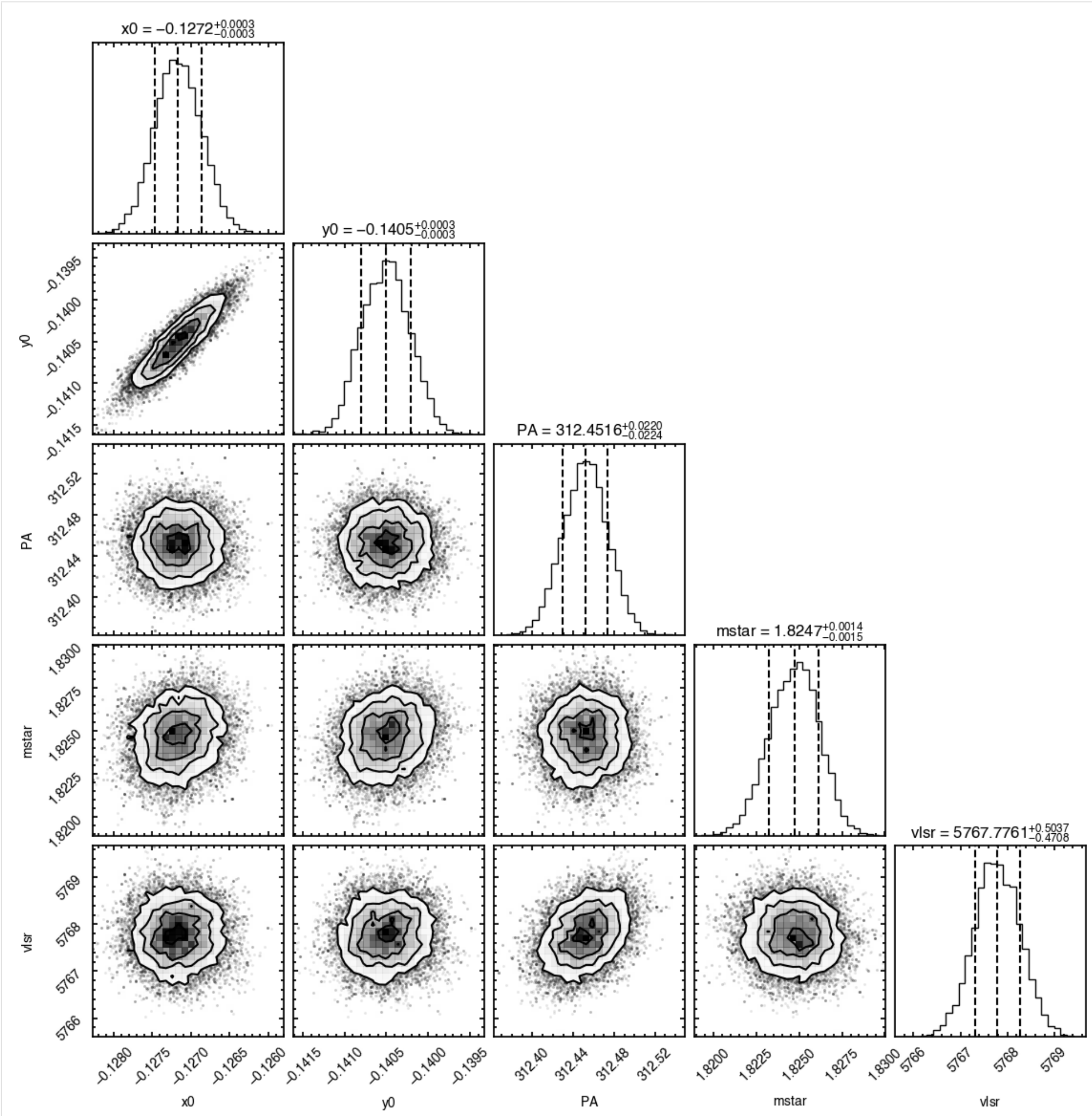


PA
316
314
312
310
308
0
200
400
600
800
1000
Steps

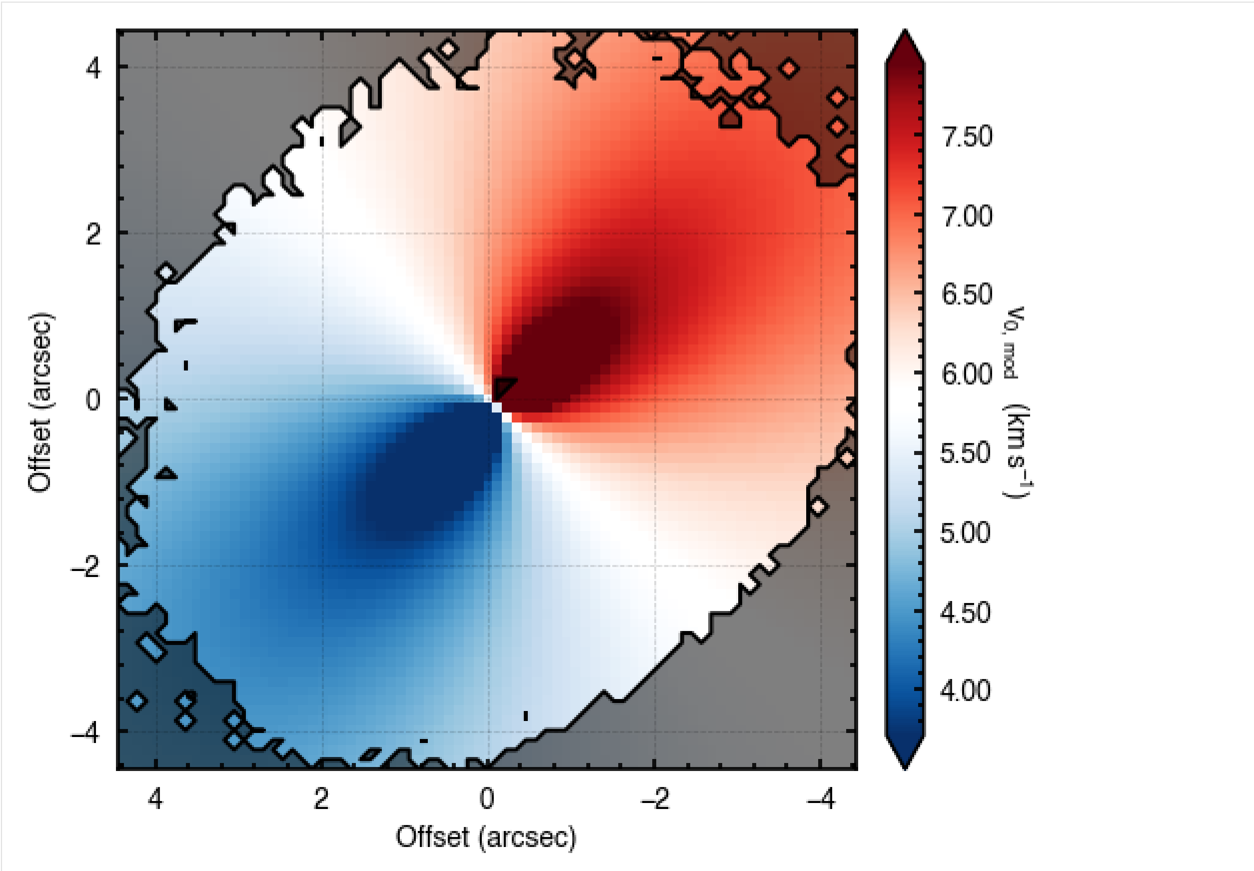


mstar
2.10
2.05
2.00
1.95
1.90
1.85
1.80
0
200
400
600
800
1000
Steps

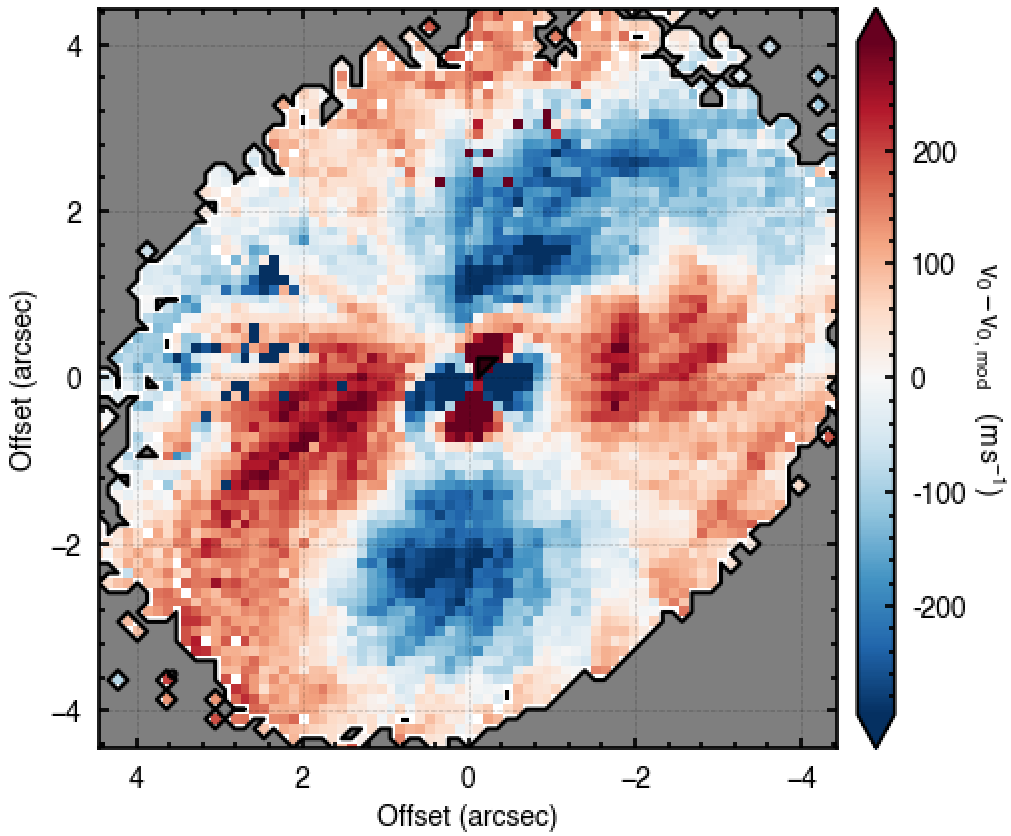

5850
5800
5750
5700
5650
vlsr
0
200
400
600
800
1000
Steps



x0 = −0.1272+0.0003 −0.0003
y0 = −0.1405+0.0003 −0.0003
PA = 312.4516+0.0220 −0.0224
mstar = 1.8247+0.0014 −0.0015
vlsr = 5767.7761+0.5037 −0.4708
x0
y0
PA
mstar
vlsr



Offset (arcsec)
Offset (arcsec)
4
2
0
–2
–4
7.50
7.00
6.50
6.00
5.50
5.00
4.50
4.00
$v_{0,\,\mathrm{mod}}$ (km s^{-1})



The walkers clearly converge to a best-fit model, but from the residual plot, it is clear this is not a good representation of the data. There are four main residual features we can identify here, and interpret with the help of the Appendices from Teague et al. (2019) and Yen & Gu (2020):

1) A quadrupole feature in the inner disk, likely because the model is not centered. Indeed, the posteriors for both x_0 and y_0 have shifted the model center along the minor axis. As we will see later, this is due to the 3D nature of the disk.

2) A ringed feature with a positive residual along the red-shifted major axis and a negative residual along the blue-shifted side. This is highly suggestive of a ring of faster rotating gas. For HD 163296, this is highly likely to be associated to the surface density perturbations traced by the gas and continuum (Teague et al., 2018).

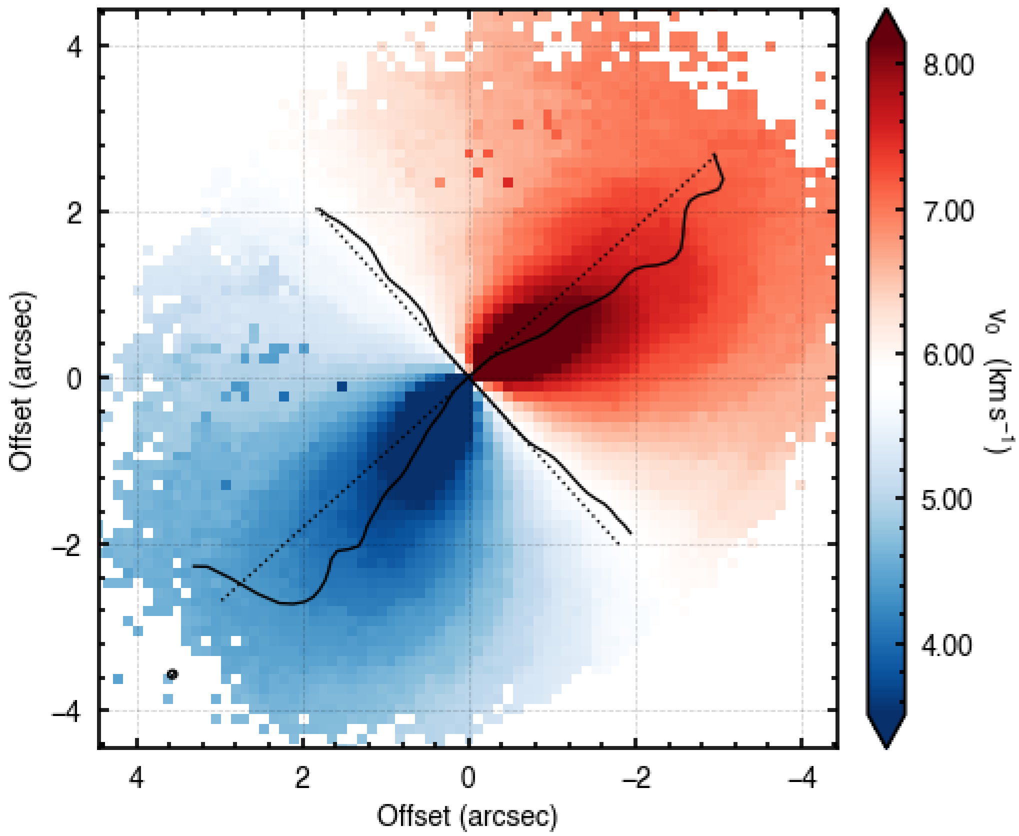
3) A larger quadrupole feature in the outer disk due to a misspecified emission surface. Again, it is clear that a geometrically thin model is not a good description of HD 163296.

4) An arc-shaped residual along the North East edge of the disk. This is contamination from the back side of the disk. Here the line emission arises from *behind* the continuum (the 'back side' of the disk), while the model assumes all the emission arises from the front side of the disk. This is a very common residual feature and can be used to determine the absolute geometry of the disk (i.e., in which direction is the disk tilted relative to the observer). Figure 1 from Teague et al. (2018b) demonstrates this geometry.

Revealing the 3D Structure

It is obvious from the residuals that a 2D model is a poor fit for HD 163296. In fact, the vertical extent of a disk can be seen in the rotation map as the lobes of the dipole rotation pattern bend away from the disk major axis. This is very clear for the case of HD 163296. We can use the `plot_maxima` function which will find the line of maximum and minimum velocities along the red-shifted and blue-shifted major axes, respectively.

```
[5]: cube.plot_maxima(inc=46.7,PA=312.0, r_max=4.0, smooth=0.3)
```



In the above figure, the dotted lines show the major and minor axes of the disk based on the `inc` and `PA` values provided. The solid lines show the line of extreme velocities, showing that the emission surface is distinctly elevated. The `smooth` argument smooths the lines to beat down the jitter due to the noise.

This approach is also good for searching for signs of warps or strong radial flows where there should be a significant variation along the minor axis, as discused in Rosenfeld et al. (2014) or Casassus et al. (2015).

Parameterising the Emission Height

We can go beyond the geometrically thin disk approximation and use a more realistic 3D structure. Here we assume an azimuthually symmetric emission surface parameterized by

$$z(r) = z_0 \times \left(\frac{r - r_{\rm cavity}}{1''}\right)^{\psi} \times \exp\left(-\left[\frac{r - r_{\rm cavity}}{r_{\rm taper}}\right]^{q_{\rm taper}}\right)$$

In this parameterization, z_0 is related to the aspect ratio of the emission surface, ψ describes the flaring of this surface, $r_{\rm cavity}$ allows for an inner cavity, such as in a transition disk, and $\{r_{\rm taper}, q_{\rm taper}\}$ descibe an exponential taper to model the drop in emission height at the edge of the disk. With this description we can recover the geometrically thin limit when $z_0 = 0$ and the conical surface discussed in Rosenfeld et al. (2013) and used with ConeRot with $z_0 > 0$, $\psi = 1$ and $r_{\rm taper} = \infty$.

NOTE: Previous versions of `eddy` used a double power law described by `z1` and `phi`. The large degeneracy between these parameters strongly favoured a move to this tapered approach (hat tip to Sean Andrews for the suggestion).

The coordinate transforms for geometrically thin disks or conical surface are purely analytical so are very quick. For the more complex surfaces this is a slower process. For monotonically increasing surface with a flaring value close to unity, this can be done in an iterative manner. This is the default in `eddy`, and the number of iterations used in the deprojection is set by the `cube.flared_niter` argument, defaulting to 5.

For the more complex surface, particularly those that are not monotonically increasing (i.e., those using the exponential tapered edge), this iterative approach fails (see the `GoFish` documentation for an example). To circumvent this, `eddy` also has a slower, but more robust, approach to deprojecting the pixels, invoked with the `'shadowed'` parameter. This builds a 3D model of the emission surface which is then rotated and projected onto the sky.

Rotation Direction

Unlike for geometrically thin disks, we can now distinguish the rotation direction of the disk. In `eddy`, this information is encoded in the *sign* of the inclination we define. We allow the inclination to run from $-90°$ to $90°$, with negative inclinations representing counter-clockwise rotation and positive inclinations described clockwise rotation. Again, the `GoFish` documentation has examples to demonstrate this.

One problem of this parameterization is that it is very hard for MCMC methods to jump between negative and positive values of inclination. As such, it is recommended to either fix the inclination, and vary stellar mass, or determine the correct sign of the inclination from the `plot_maxima` function.

Impact on Velocity Structure

The inclusion of a non-zero height will have two affects. Firstly, it will alter the deprojection which will have most significant difference along the semi-minor axis. Secondly, we have to correct $v_{\rm kep}$ for a) the additional distance to the star from an elevated location (difference in a radial polar coordinate and a radial cylindrical coordinate), and b) the projected gravitational force, resulting in,

$$v_{\rm kep} = \sqrt{\frac{GM_{\rm star}r^2}{(r^2 + z^2)^{3/2}}},$$

where r is the cylindrical (or midplane) radius. Note that for the case $z = 0$, this reduced to the same equation used in Tutorial 1.

Fitting a 3D Model

With all this in mind, we can try a fit with all these parameters. Note that with the bending of the lobes in the v_0 map towards the bottom right of the image, we can infer that the disk is tilted such that that top left edge is closer to the observer, meaning that the disk is rotating in a clockwise direction. As such, we use a *positive* inclination.

Because we are almost doubling the number of free parameters, and indeed there's a covariance between them, we will use a larger number of walkers to help the convergence of the chains. As such, we're also leveraging the `multiprocessing` package.

[6]:
```
params = {}
params['x0'] = 0
params['y0'] = 1
params['PA'] = 2
params['mstar'] = 3
params['vlsr'] = 4
```

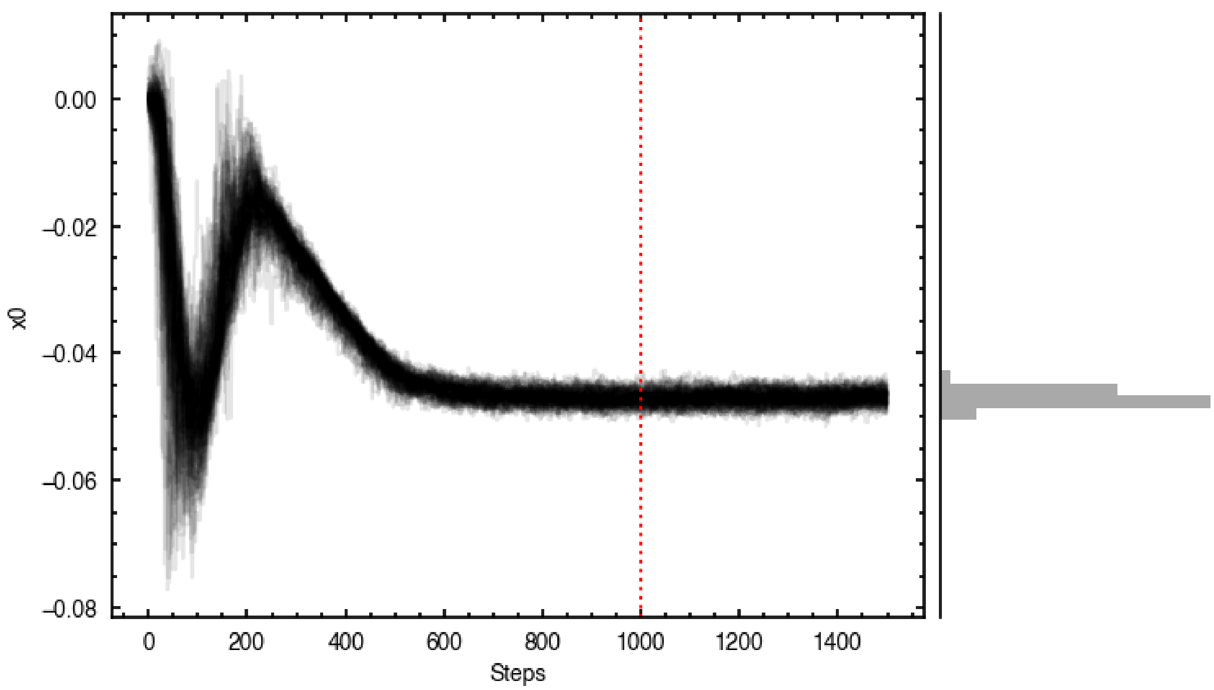
(continues on next page)

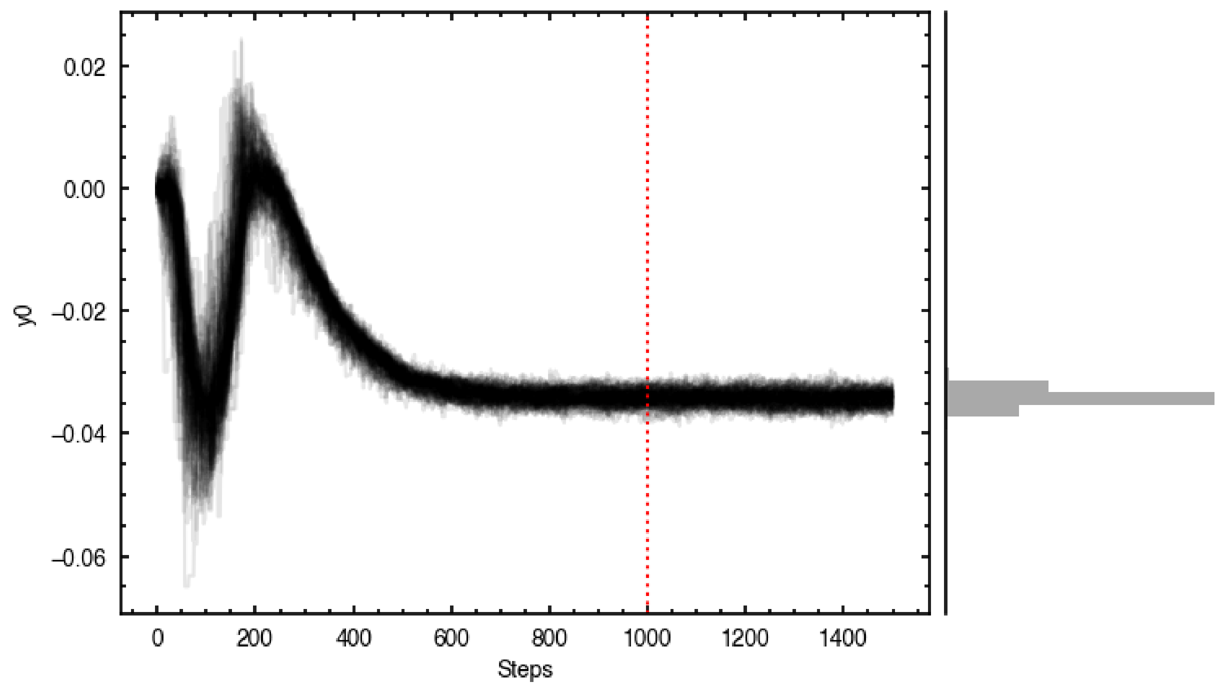
(continued from previous page)

```
p0 = [0.0, 0.0, 312., 2.0, 5.7e3]

params['inc'] = 46.7
params['dist'] = 101.0

# Include elevated emission surface parameters.

params['z0'] = 5
params['psi'] = 6
params['r_taper'] = 7
params['q_taper'] = 8

params['r_min'] = 2.0 * cube.bmaj

p0 += [0.25, 1.0, 3.0, 2.0]

with Pool() as pool:
    samples = cube.fit_map(p0=p0, params=params, optimize=False,
                           nwalkers=128, nburnin=1000, nsteps=500,
                           pool=pool)
```

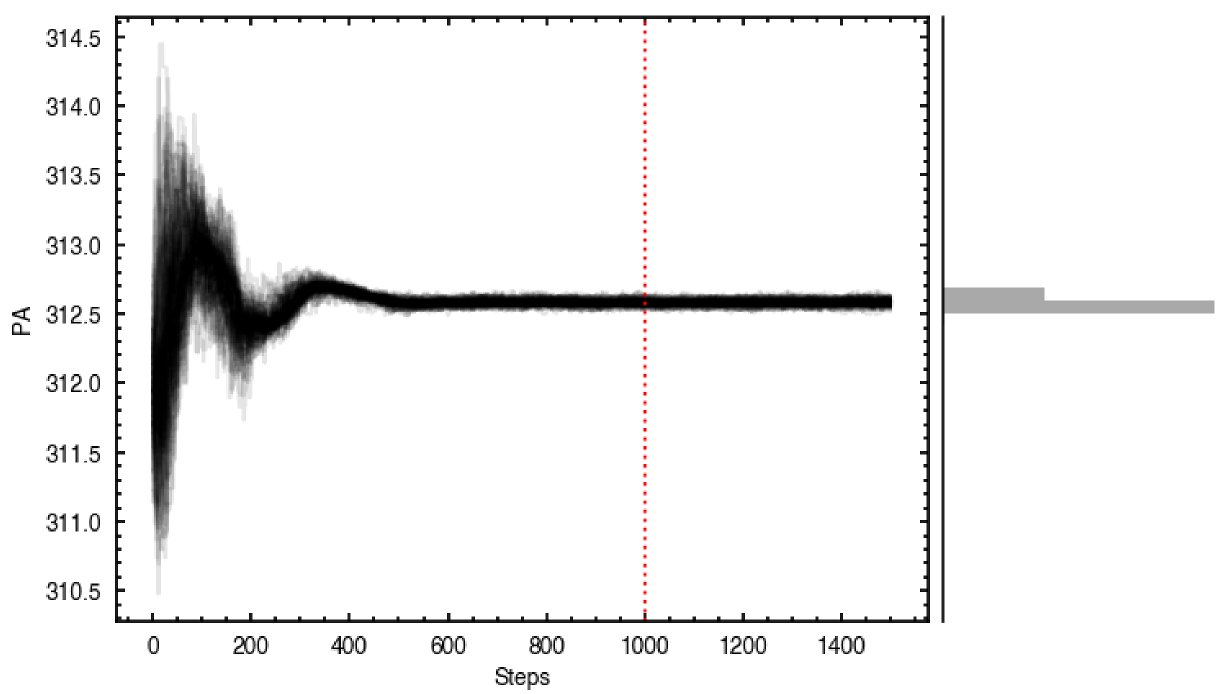
```
Assuming:
        p0 = [x0, y0, PA, mstar, vlsr, z0, psi, r_taper, q_taper].
```

```
100%|| 1500/1500 [00:47<00:00, 31.43it/s]
```

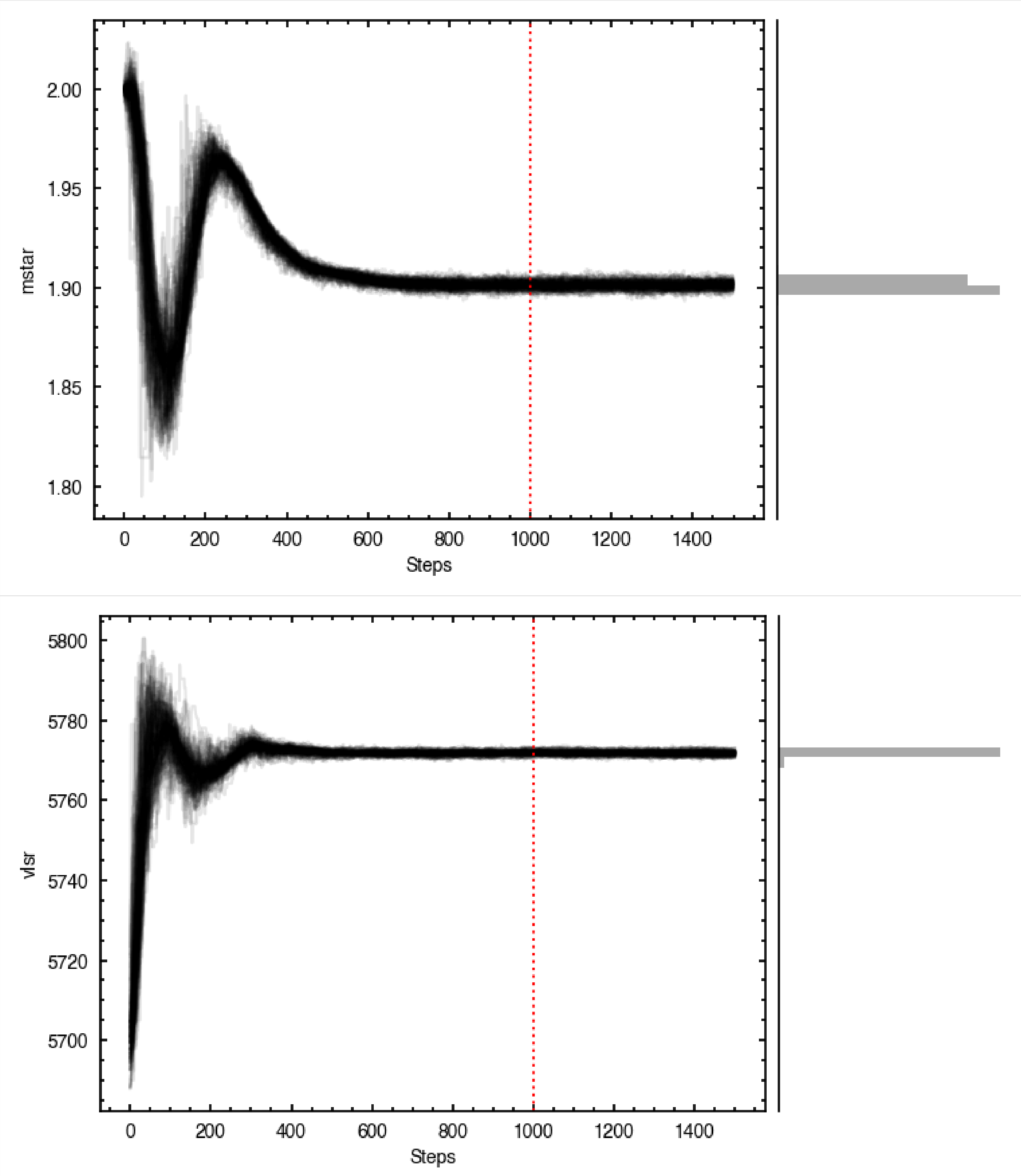


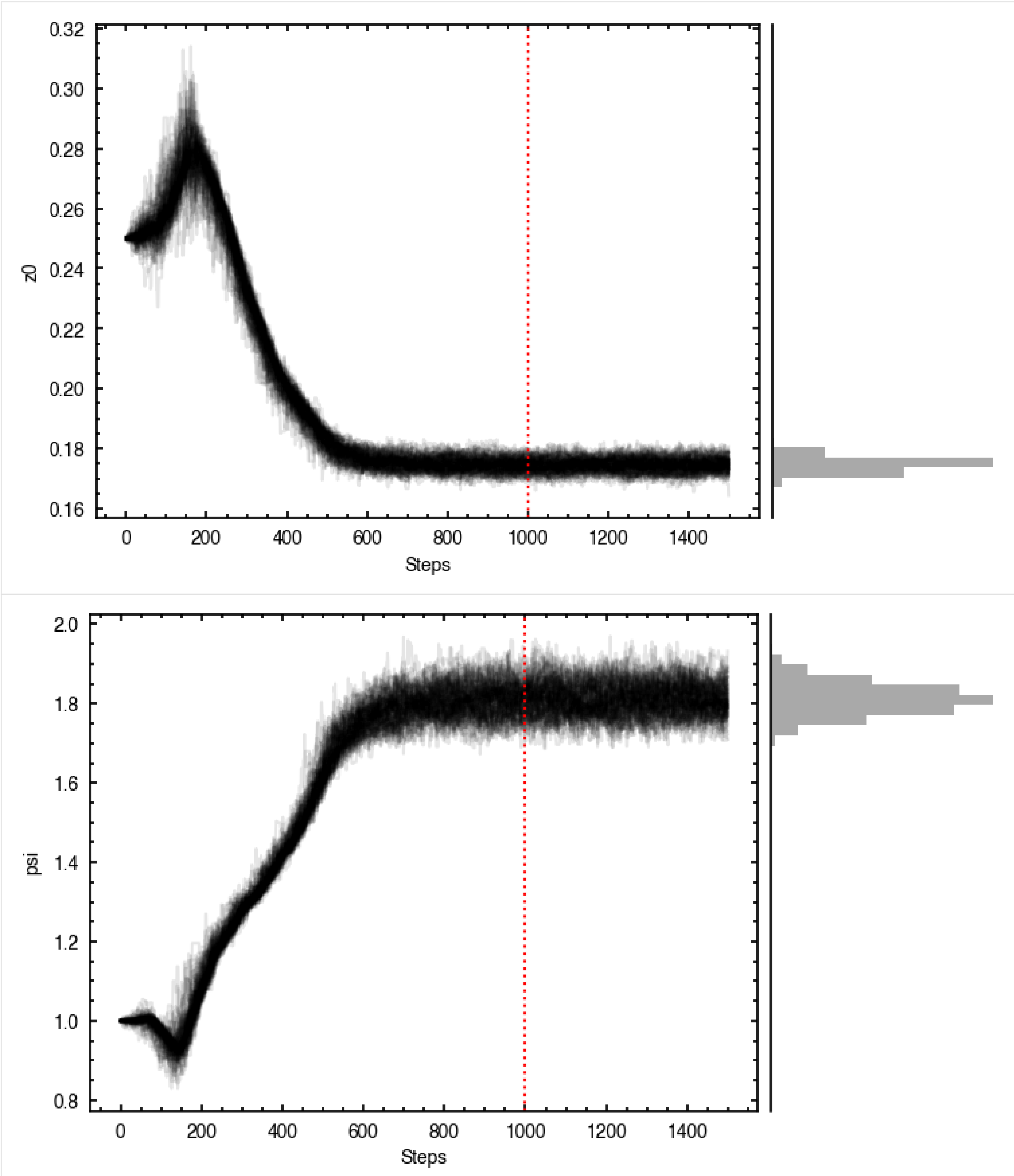


0.02
0.00
−0.02
−0.04
−0.06
y0
0
200
400
600
800
1000
1200
1400
Steps

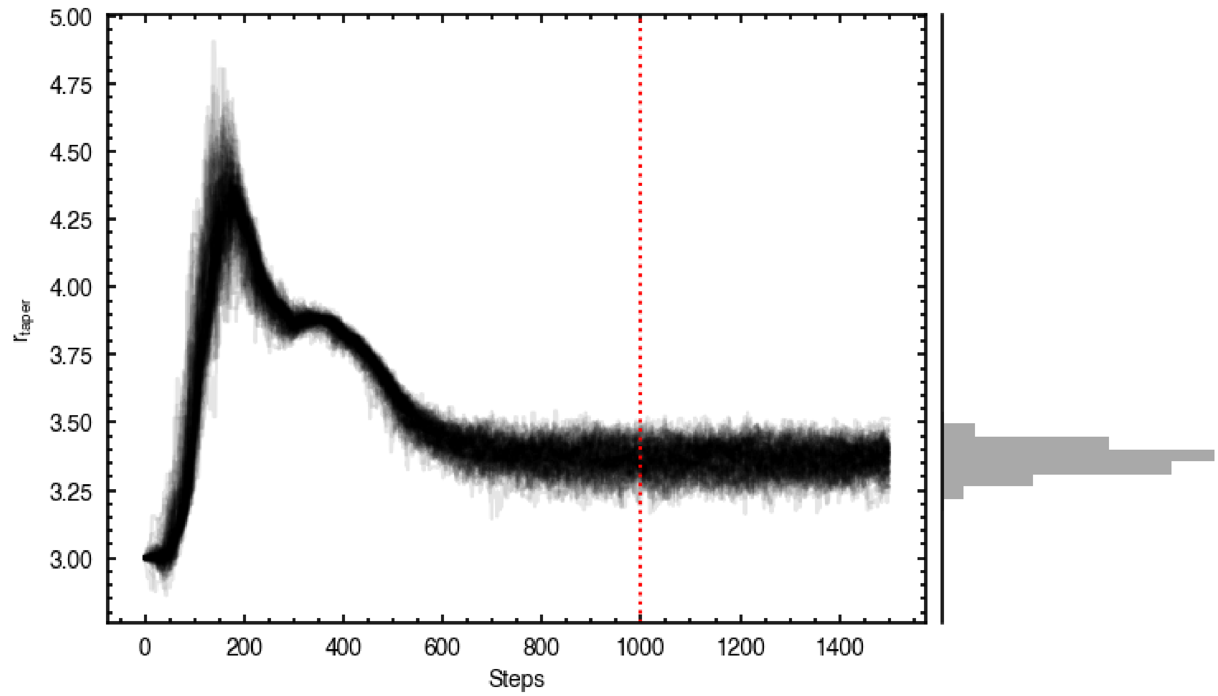


314.5
314.0
313.5
313.0
312.5
312.0
311.5
311.0
310.5
PA
0
200
400
600
800
1000
1200
1400
Steps

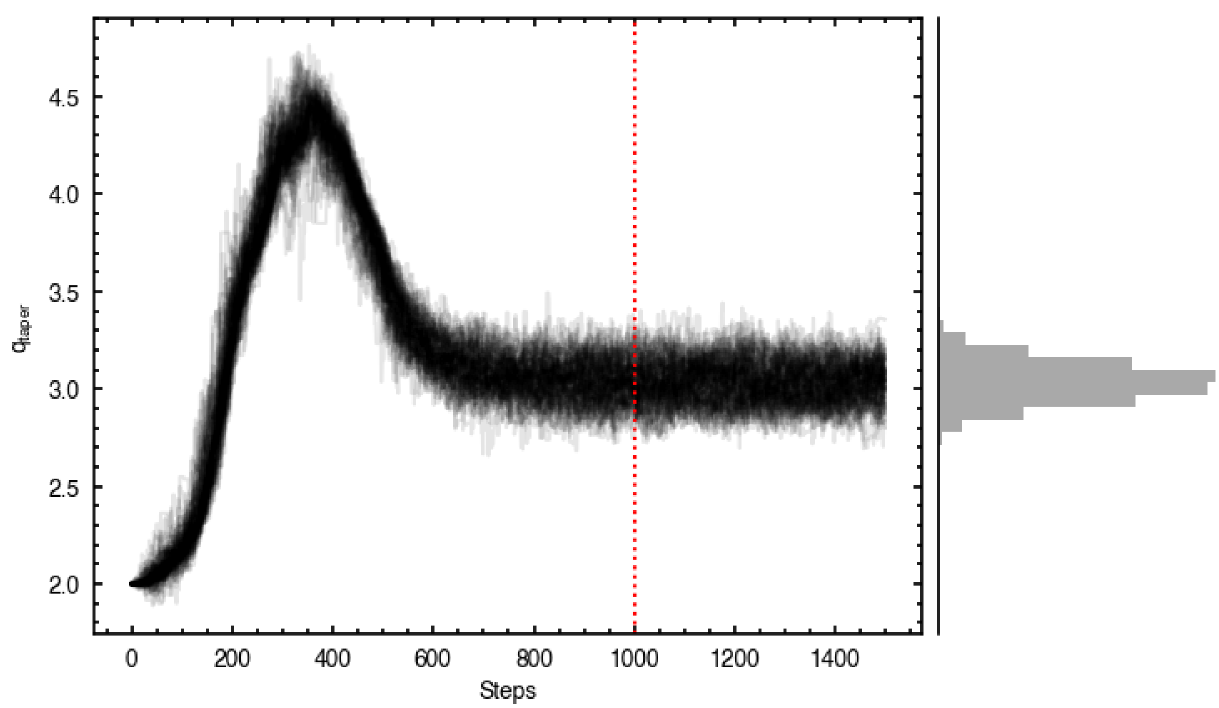

mstar
2.00
1.95
1.90
1.85
1.80
0
200
400
600
800
1000
1200
1400
Steps
vlsr
5800
5780
5760
5740
5720
5700
0
200
400
600
800
1000
1200
1400
Steps



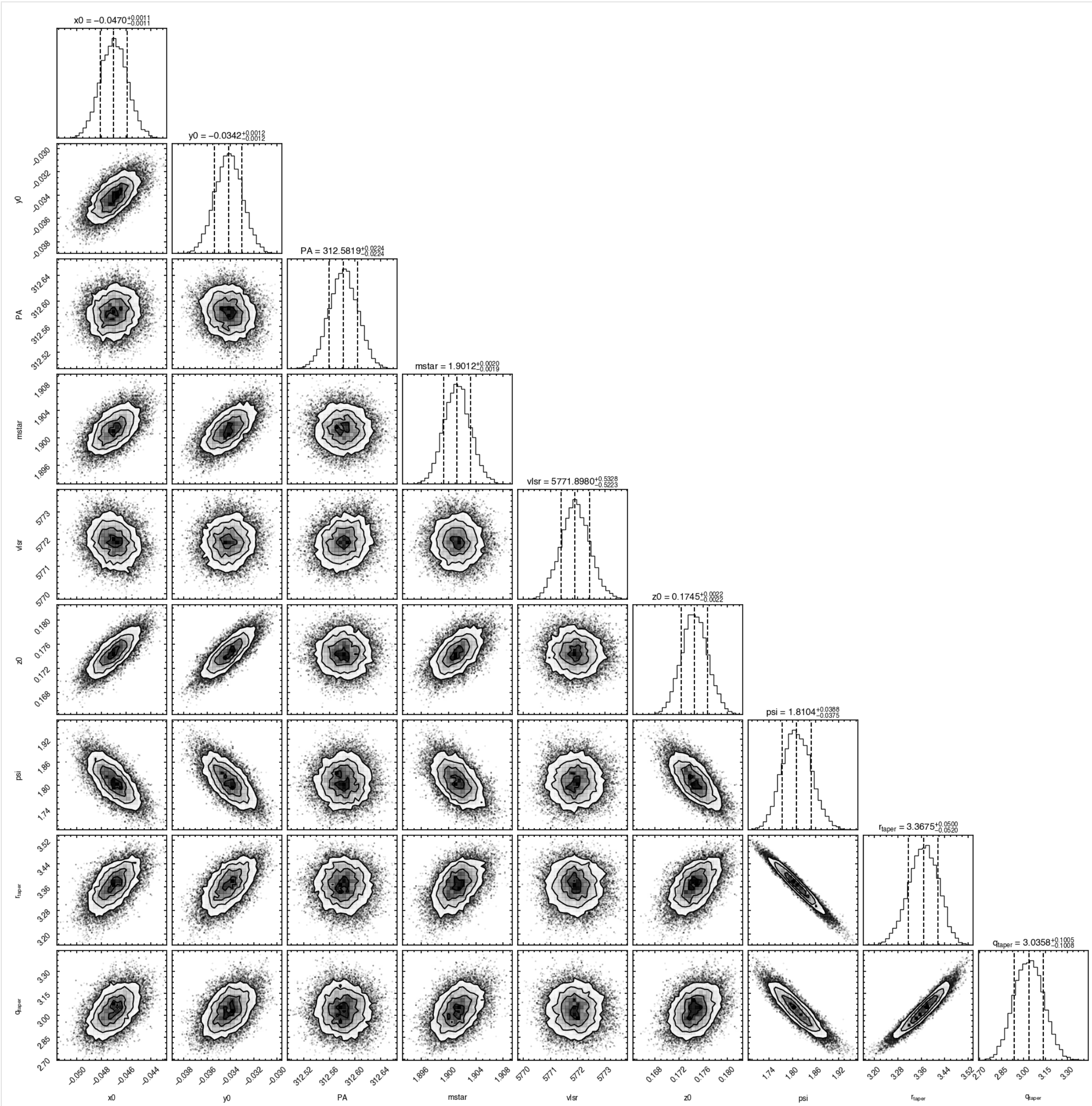
z0
0.32
0.30
0.28
0.26
0.24
0.22
0.20
0.18
0.16
0
200
400
600
800
1000
1200
1400
Steps
psi
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0
200
400
600
800
1000
1200
1400
Steps

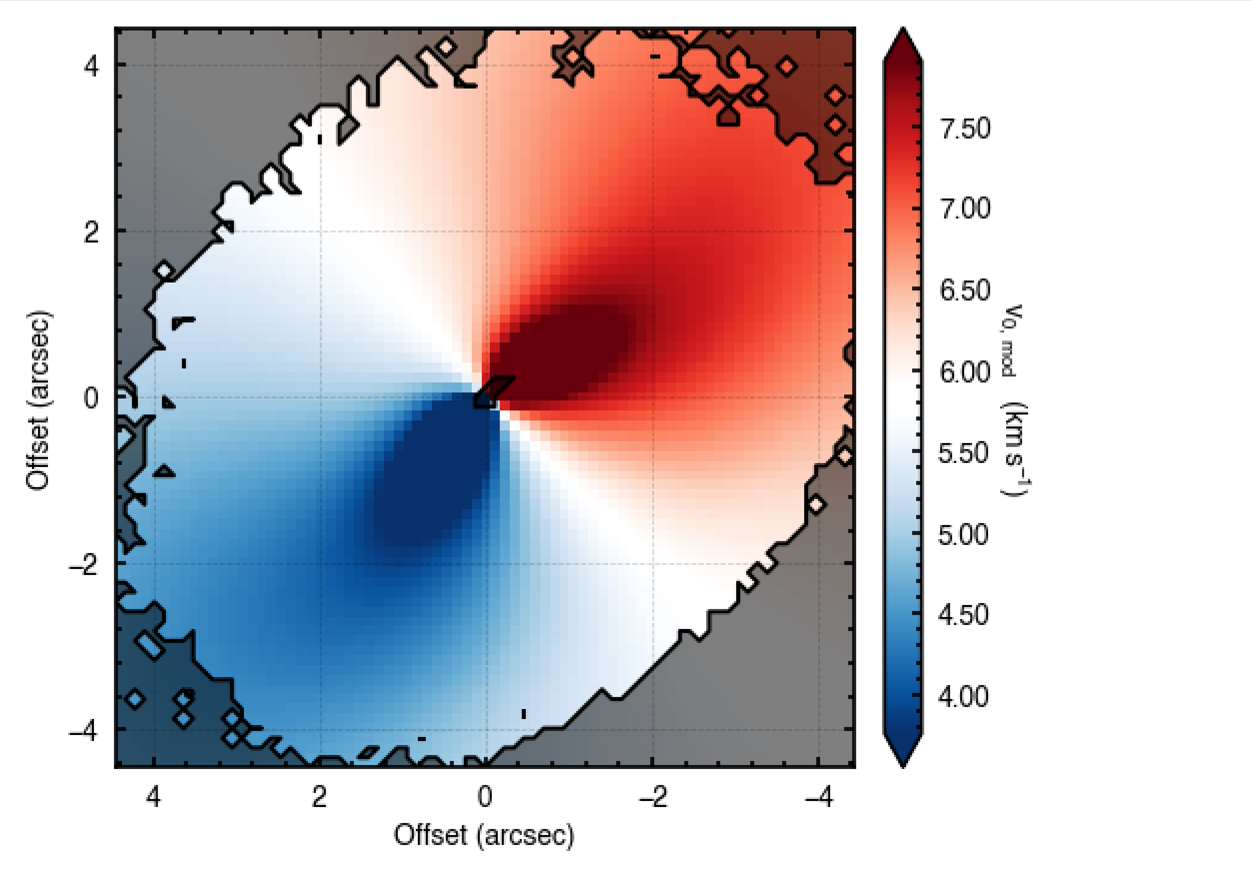


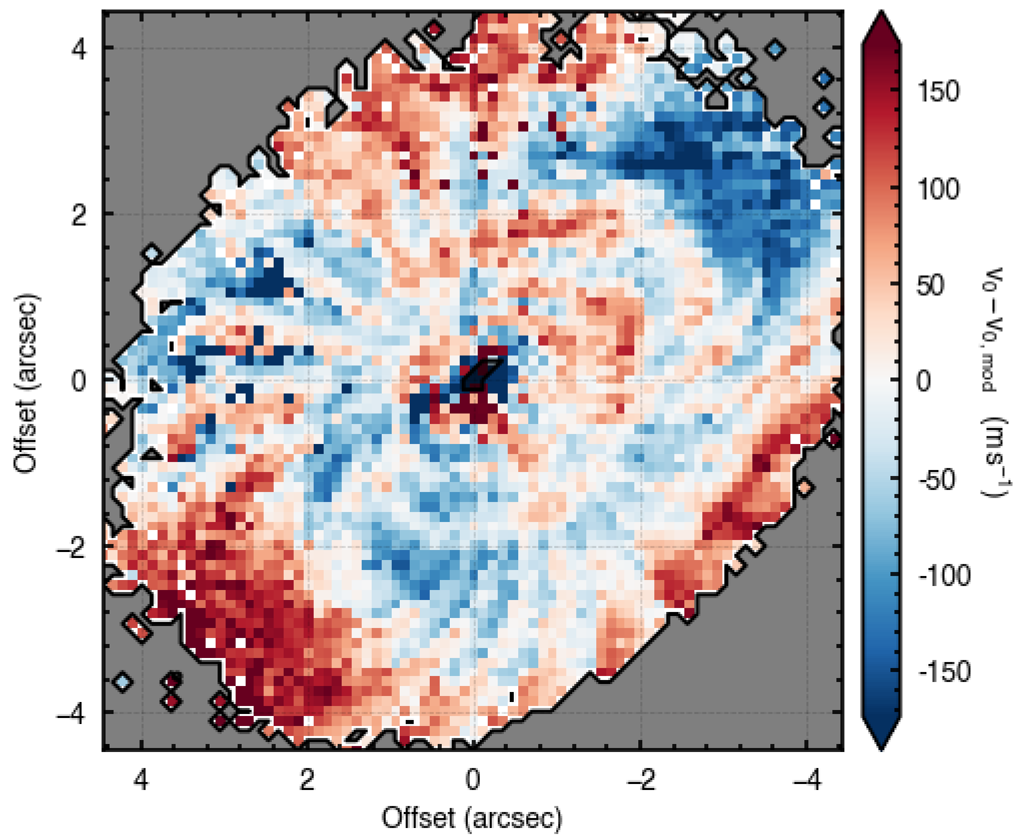
5.00
4.75
4.50
4.25
4.00
3.75
3.50
3.25
3.00
r_{taper}
0
200
400
600
800
1000
1200
1400
Steps



4.5
4.0
3.5
3.0
2.5
2.0
q_{taper}
0
200
400
600
800
1000
1200
1400
Steps




4
2
0
–2
–4
Offset (arcsec)
4
2
0
–2
–4
Offset (arcsec)
7.50
7.00
6.50
6.00
5.50
5.00
4.50
4.00
$v_{0,\,\mathrm{mod}}$ (km s^{-1})



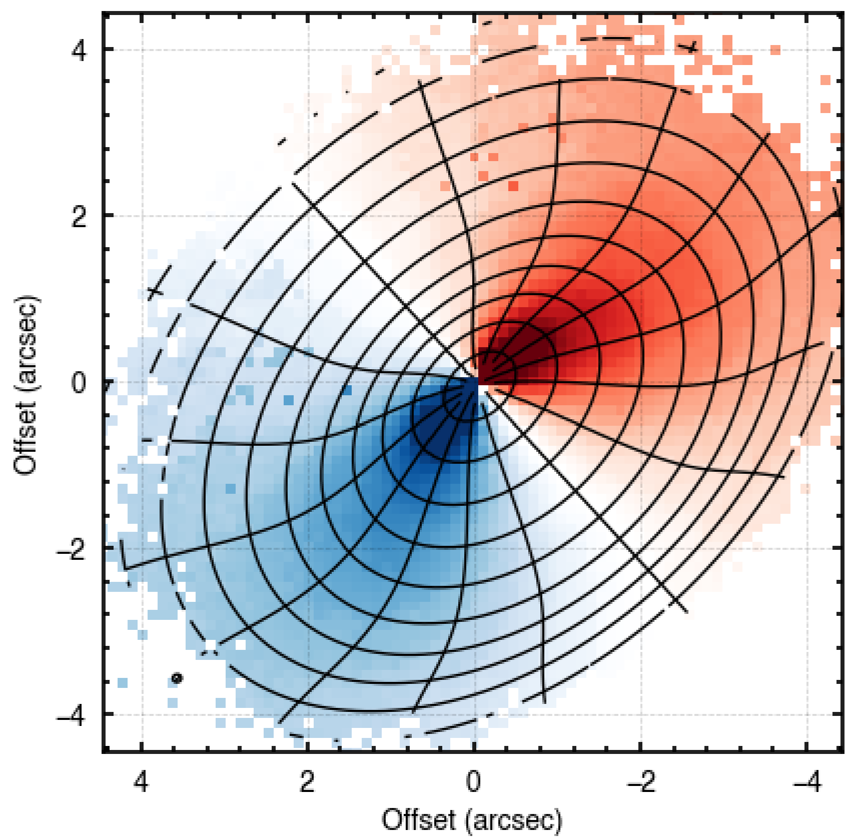
It looks like the chains have converged to a best-fit, and the residuals have definitely improved compared to the 2D fit (the the residuals are $\sim 200\ \mathrm{m\,s^{-1}}$ comapred to the $\sim 300\ \mathrm{m\,s^{-1}}$ of the 2D fit). We still see the arc-like residual along the North Eastern side, but the quadrupole residuals have been suppressed. Instead, we start to see some ordered substructure associated with the kinematic planetary signature identified by Pinte et al. (2018) and perturbations in the disk surface density profile (Teague et al., 2018).

Interestingly, we do find large positive and negative residuals at the disk edge along the blue-shifted and red-shifted major axes, respectively. This is due to slower rotating gas which Dullemond et al. (2020) argued was due to the large pressure gradienta the outer edge of the disk.

Plotting the Emission Surface

Now that we have constrained the emission surface (on the assumption that out model is a reasonable replication of the true source structure), we can overplot the emission surface to check what that looks like. Here we've used the `imshow_kwargs` and `plot_surface_kwargs` to fill the background with the attached data. The `plot_model_surface` is a wrapper for `plot_surface` which is slightly more flexible, but only accepts fixed geometrical properties rather than a (`samples`, `params`) pair.

[7]:
```
imshow_kwargs = dict(vmin=2.77, vmax=8.77, cmap=cube.cmap())
plot_surface_kwargs = dict(fill='self.data / 1e3',
                           imshow_kwargs=imshow_kwargs)

fig = cube.plot_model_surface(samples, params, plot_surface_kwargs=plot_surface_
↪kwargs)
```

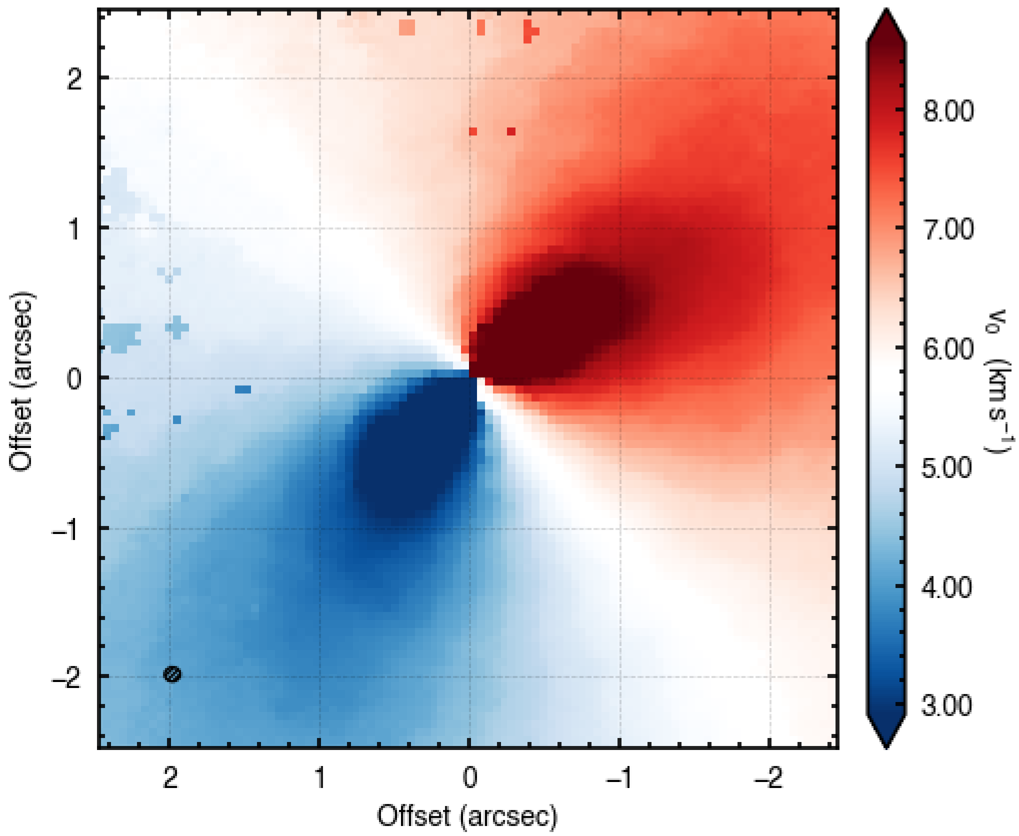


Improving the Fit

As we have discussed, a major residual we are seeing is due to the back side of the disk. We can use the masking properties to avoid this region in the fit of the data. As we are cutting down the region we are fitting, it is sensible to reload the data and use a tight FOV value.

[8]:
```
cube = rotationmap(path='HD163296_CO_v0.fits',
                   downsample=4,
                   FOV=5.0)
cube.plot_data()
```

```
Assuming uncertainties in HD163296_CO_dv0.fits.
```



[9]:
```
params = {}
params['x0'] = 0
params['y0'] = 1
params['PA'] = 2
params['mstar'] = 3
params['vlsr'] = 4

p0 = [0.0, 0.0, 312., 2.0, 5.7e3]

params['inc'] = 46.7
params['dist'] = 101.0

params['z0'] = 5
params['psi'] = 6
params['q_taper'] = 7

p0 += [0.25, 1.0, 2.0]

params['r_taper'] = 2.0

# Include a mask that cuts out the backside emisison.

params['r_min'] = 2.0 * cube.bmaj
params['r_max'] = 2.8

with Pool() as pool:
```

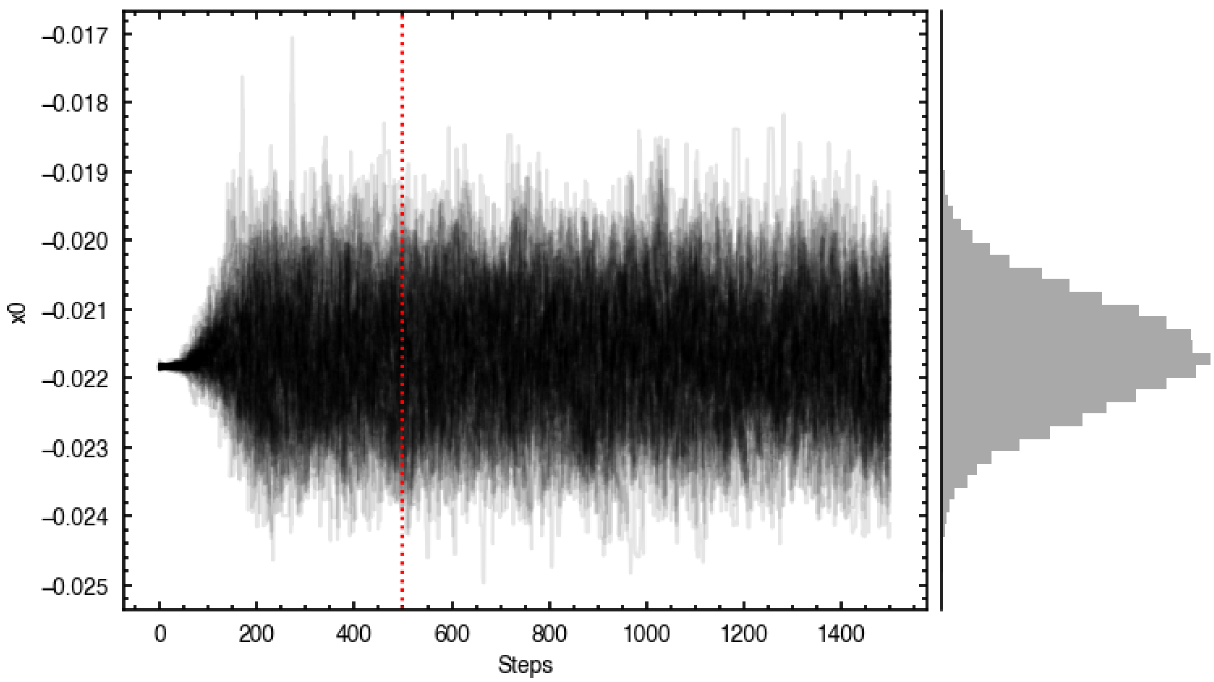
(continues on next page)

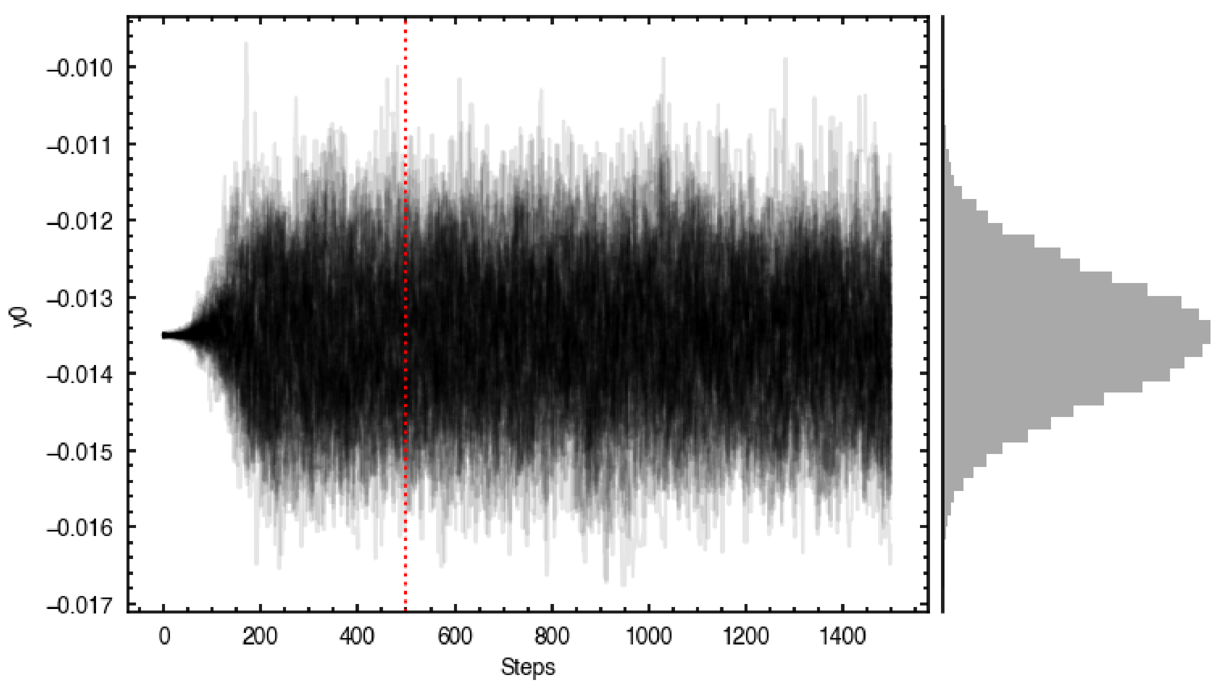
(continued from previous page)

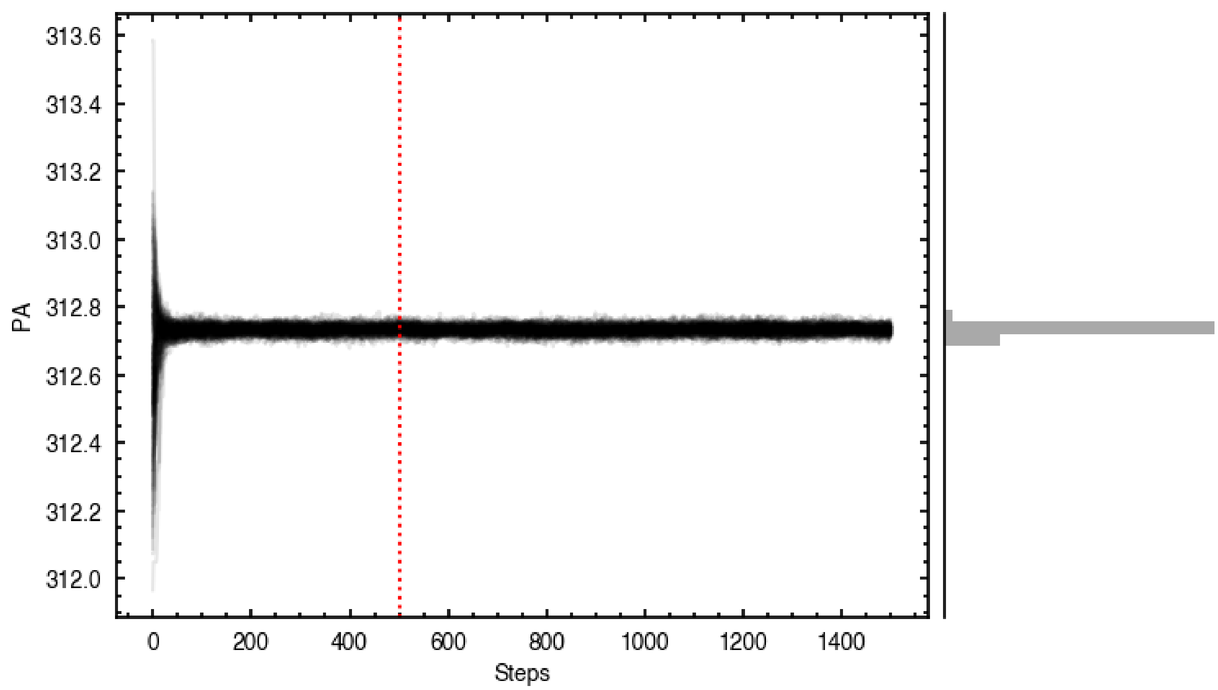
```
    samples = cube.fit_map(p0=p0, params=params, optimize=False,
                           nwalkers=128, nburnin=500, nsteps=[100, 1000],
                           pool=pool, niter=2)
```

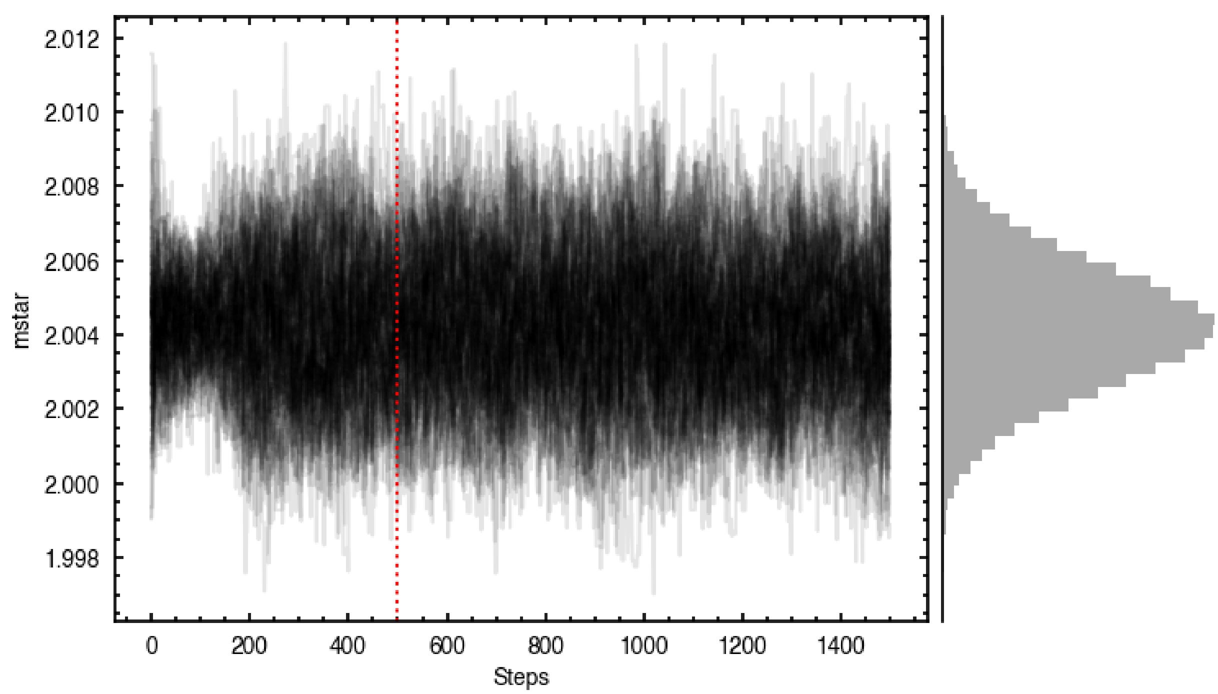
```
Assuming:
        p0 = [x0, y0, PA, mstar, vlsr, z0, psi, q_taper].
```

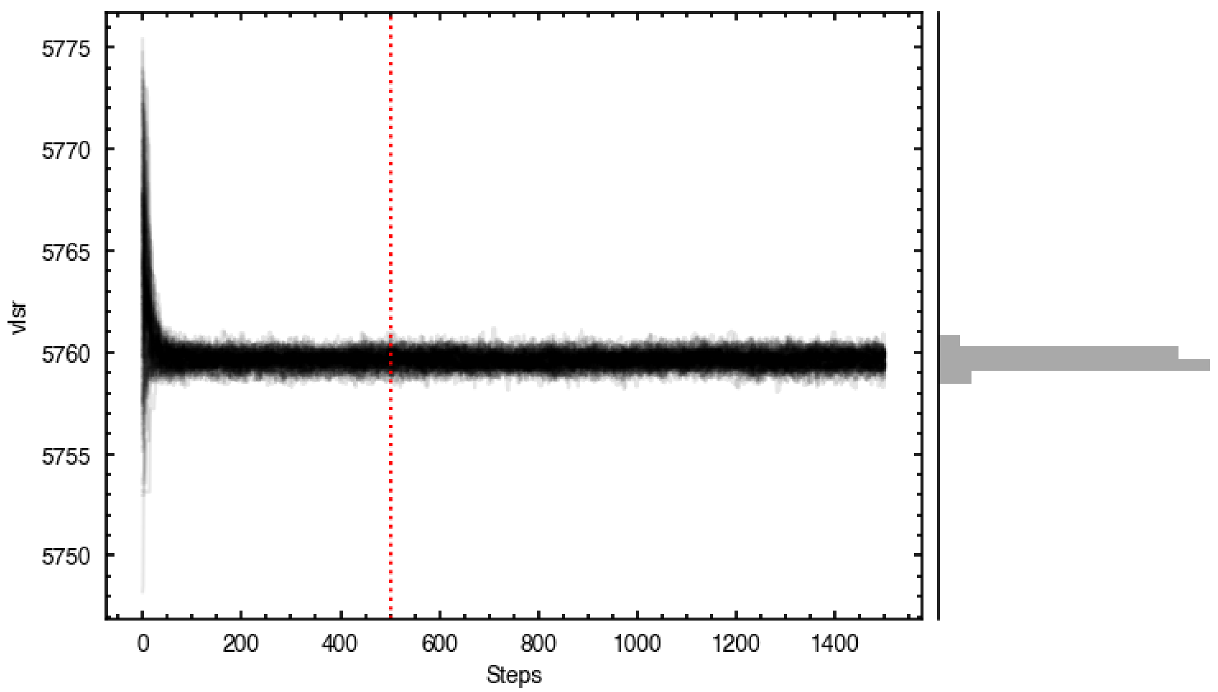
```
100%|| 600/600 [00:26<00:00, 22.86it/s]
100%|| 1500/1500 [01:07<00:00, 22.32it/s]
```

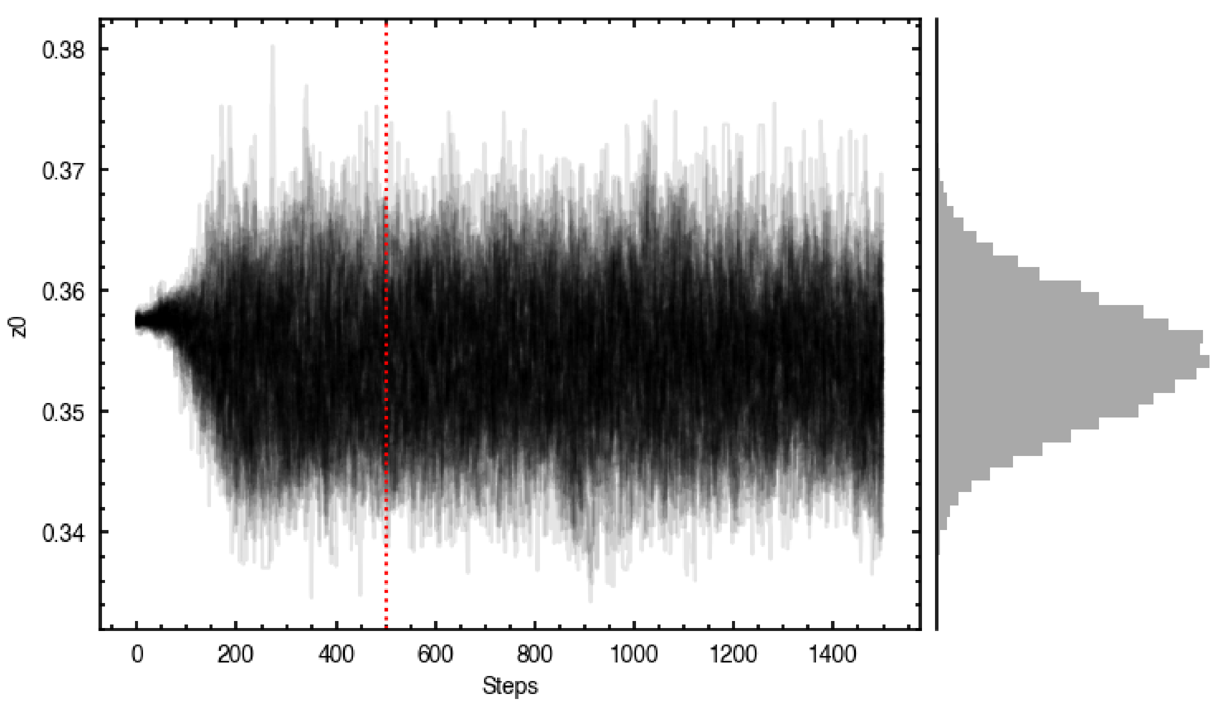


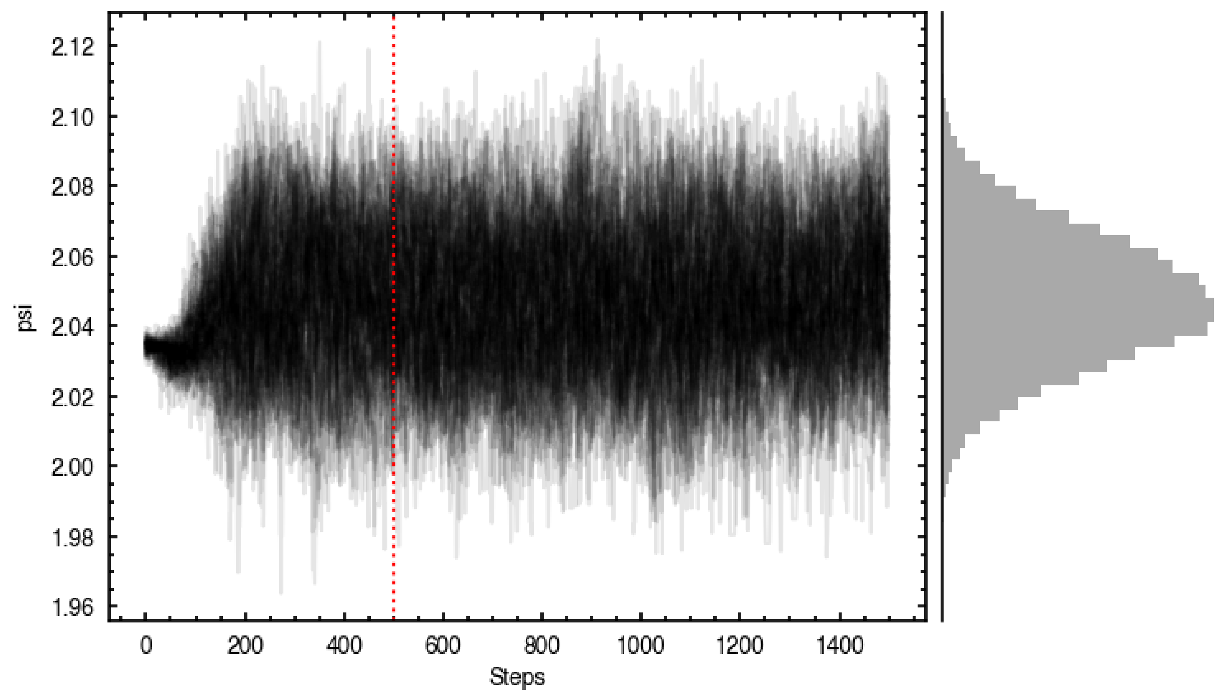



PA
313.6
313.4
313.2
313.0
312.8
312.6
312.4
312.2
312.0
0
200
400
600
800
1000
1200
1400
Steps

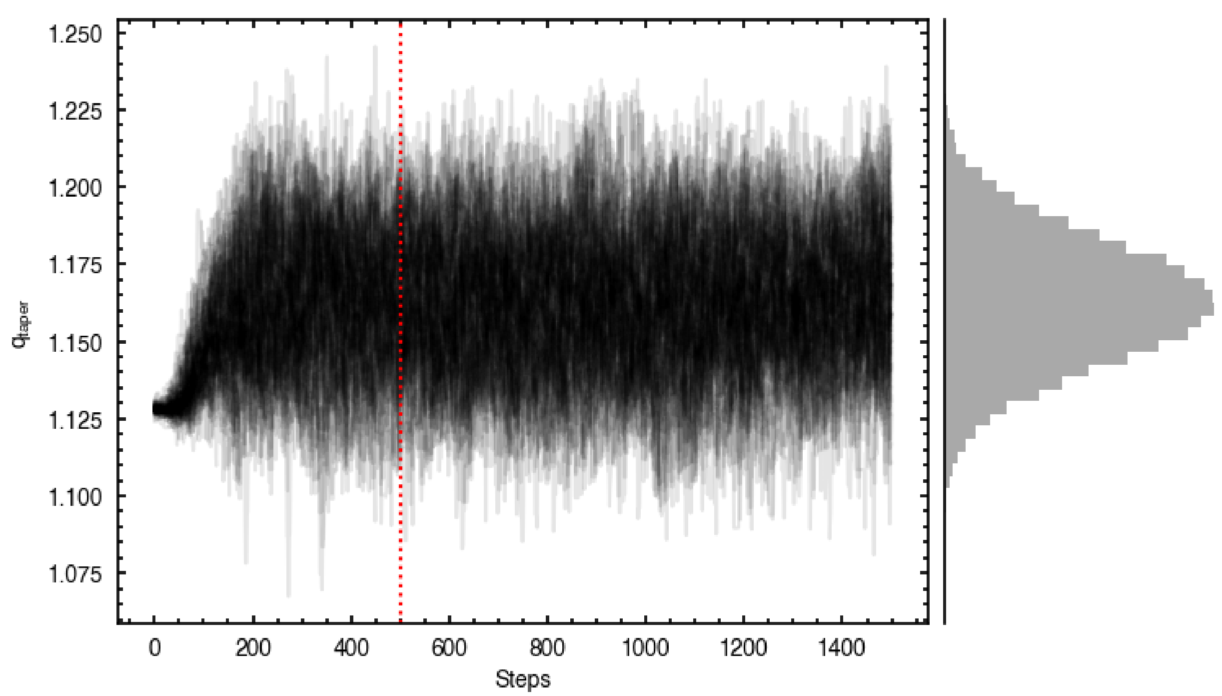

mstar
2.012
2.010
2.008
2.006
2.004
2.002
2.000
1.998
0
200
400
600
800
1000
1200
1400
Steps


5775
5770
5765
5760
5755
5750
vlsr
0
200
400
600
800
1000
1200
1400
Steps

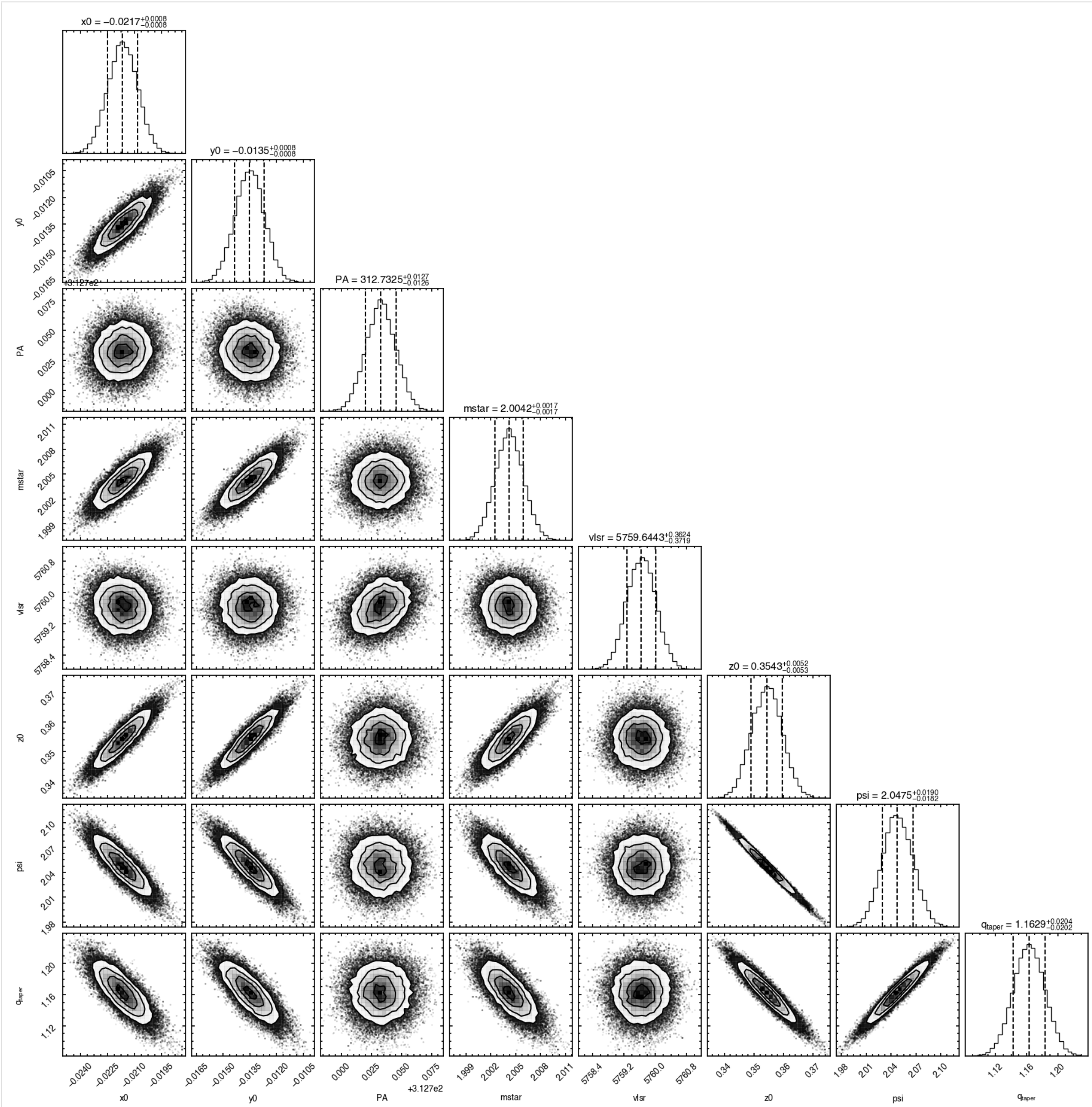

0.38
0.37
0.36
0.35
0.34
z0
0
200
400
600
800
1000
1200
1400
Steps



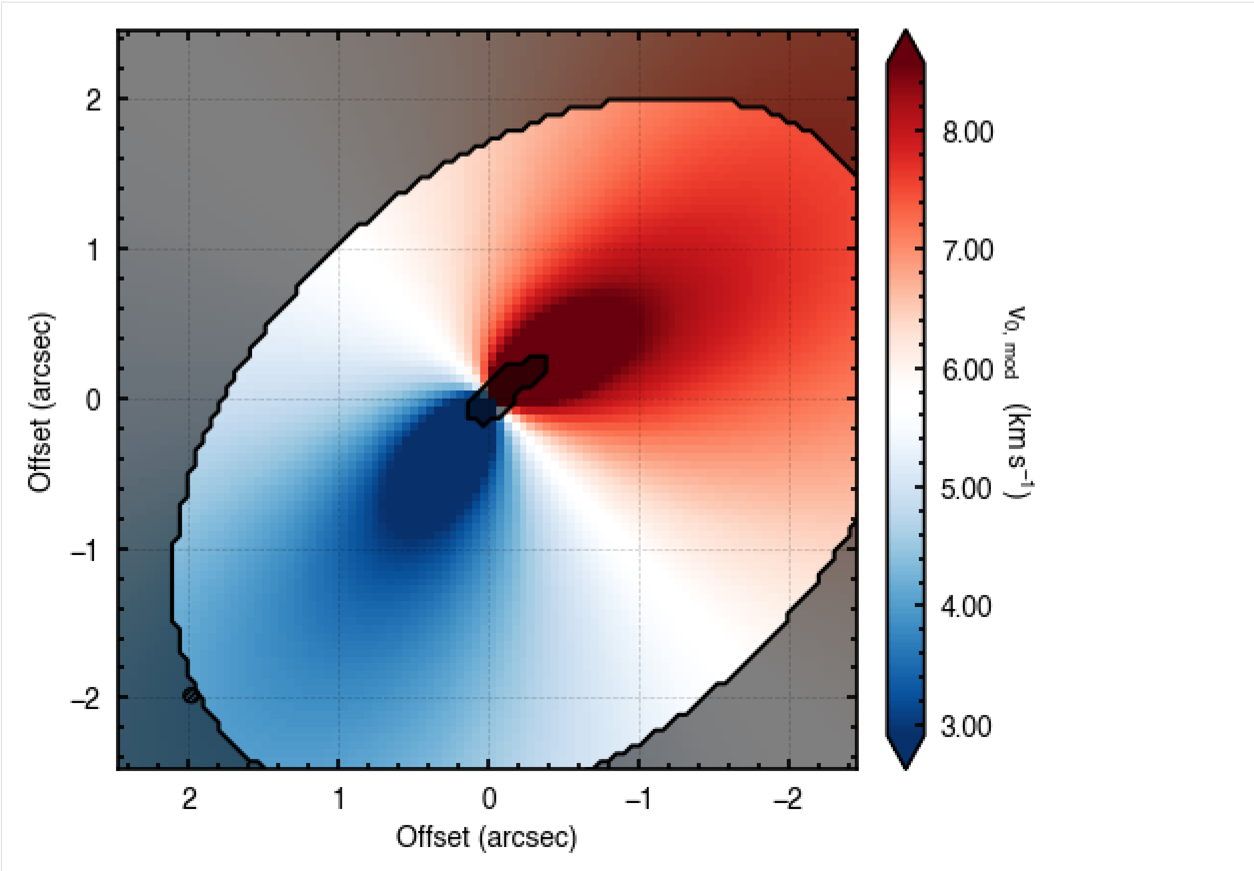
psi
2.12
2.10
2.08
2.06
2.04
2.02
2.00
1.98
1.96
0
200
400
600
800
1000
1200
1400
Steps

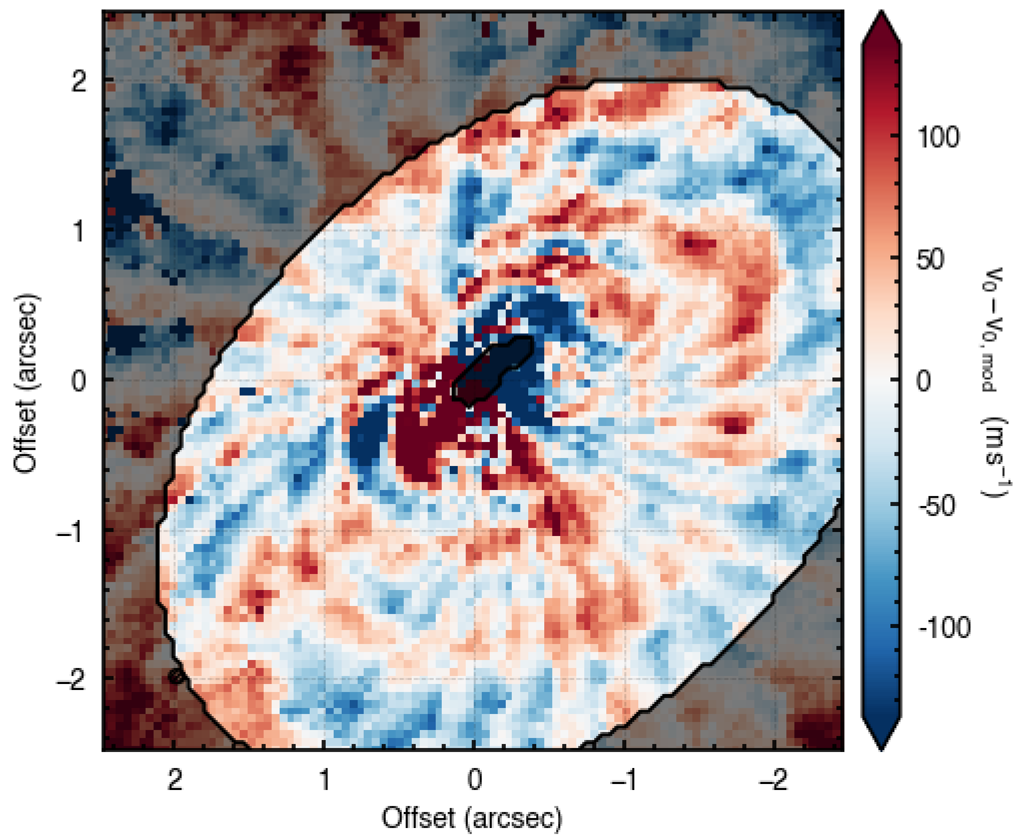


q_{taper}
1.250
1.225
1.200
1.175
1.150
1.125
1.100
1.075
0
200
400
600
800
1000
1200
1400
Steps



x0 = −0.0217$^{+0.0008}_{-0.0008}$
y0 = −0.0135$^{+0.0008}_{-0.0008}$
PA = 312.7325$^{+0.0127}_{-0.0126}$
mstar = 2.0042$^{+0.0017}_{-0.0017}$
vlsr = 5759.6443$^{+0.3624}_{-0.3719}$
z0 = 0.3543$^{+0.0052}_{-0.0053}$
psi = 2.0475$^{+0.0190}_{-0.0182}$
q_{taper} = 1.1629$^{+0.0204}_{-0.0202}$
x0
y0
PA
mstar
vlsr
z0
psi
q_{taper}
+3.127e2


Offset (arcsec)
Offset (arcsec)
$v_{0,\,\mathrm{mod}}$ (km s^{-1})
8.00
7.00
6.00
5.00
4.00
3.00
2
1
0
–1
–2



2.4 3 - An Introduction to Working with Annuli

This Notebooks works through how we can work with annuli of line emission to infer their velocity structure. Using the full line emission can be beneficial to using just the collapsed rotation map as you have more information to work with.

For this Tutorial, we will use the TW Hya data that we used in the first tutorial which is from Huang et al. (2018), and downloadable from here, or using the cell below with `wget`.

[1]:
```
import os
if not os.path.exists('TWHya_CO_cube.fits'):
    !wget -O TWHya_CO_cube.fits -q https://dataverse.harvard.edu/api/access/datafile/:
↪persistentId?persistentId=doi:10.7910/DVN/PXDKBC/QULHRK
```

2.4.1 `linecube`

This time we do not want to collapse the data to a rotation map, but keep it as a full line cube. As such, we use the `linecube` class from `eddy`, rather than the `rotationmap`.

[2]:
```
import matplotlib.pyplot as plt
from eddy import linecube
import numpy as np
```

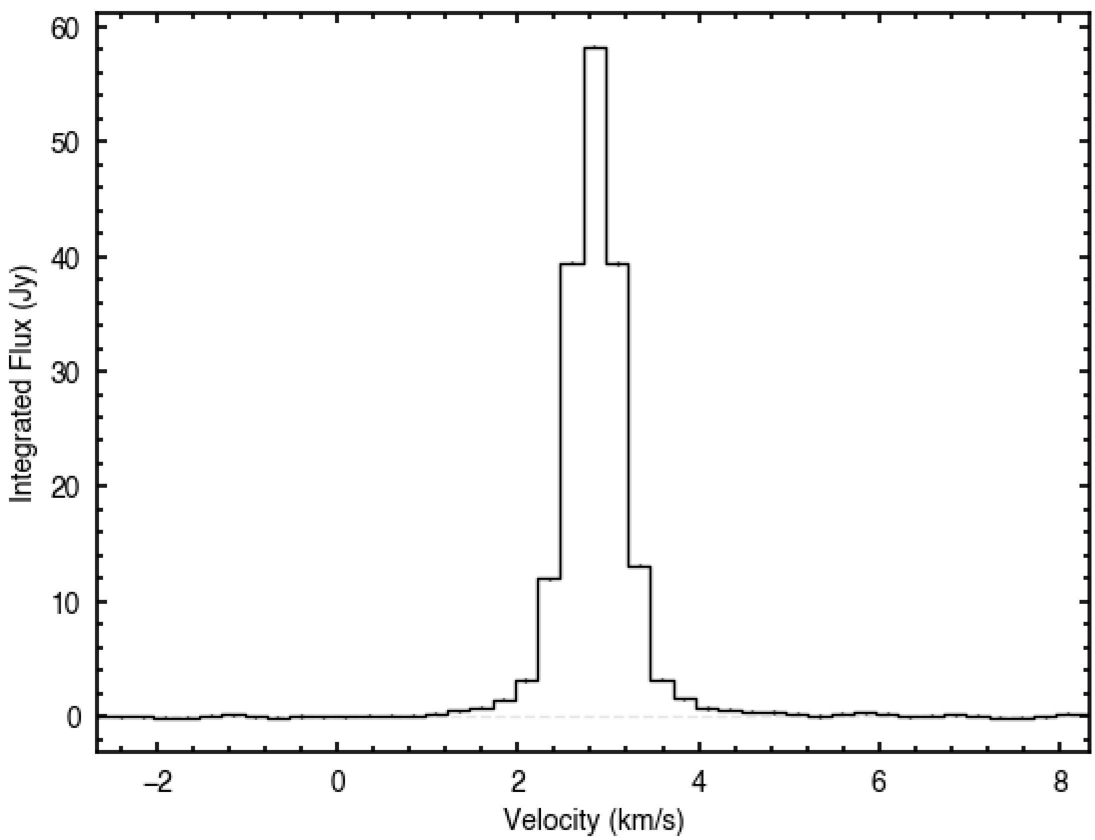
Let's load up the data. Again, we can use the same field of view argument, FOV, to cut down the field of view.

```
[3]: cube = linecube('TWHya_CO_cube.fits', FOV=8.0)
```

Inspecting the Data

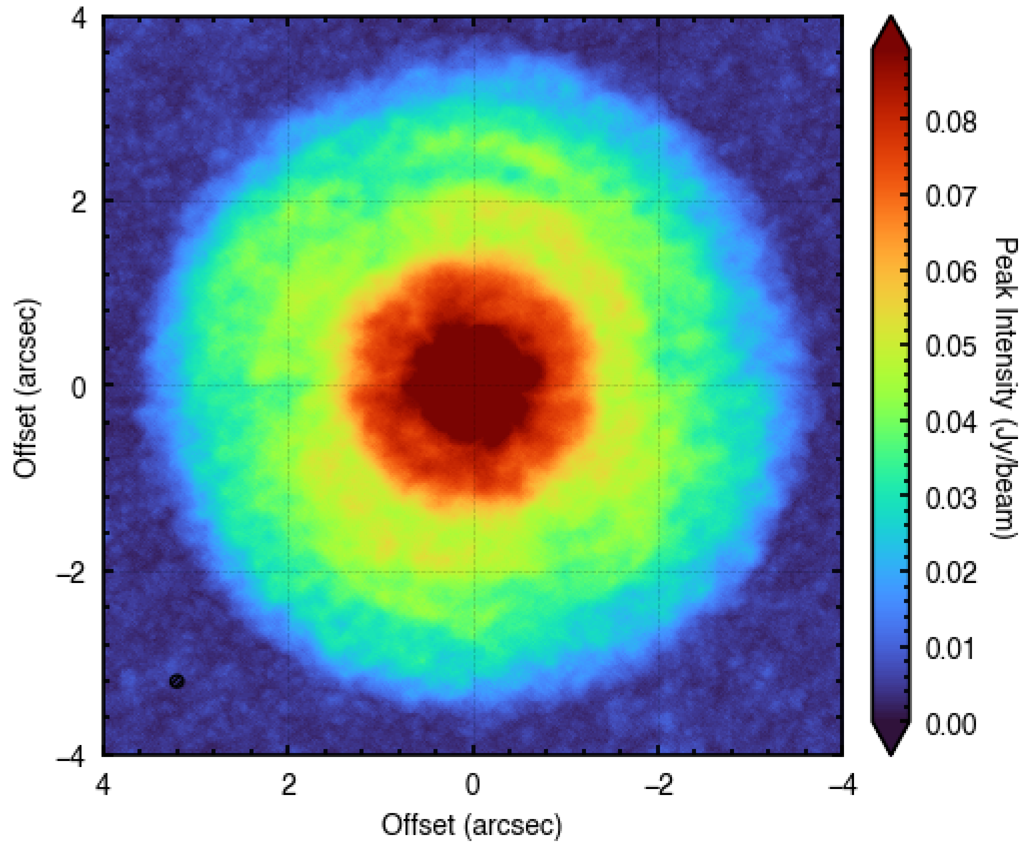
Unlike for rotationmap, a linecube instance will be 3D, with the third dimension representing the spectral dimension. This can be seen by plotting the integrated spectrum with the plot_spectrum function which basically integrating the flux in each channel.

```
[4]: cube.plot_spectrum()
```



You can clearly see the spectrum is centred on a velocity of $\sim 2.84\ \mathrm{km\,s^{-1}}$, the systemic velocity of TW Hya. Another way to inspect the data is to plot the peak intensity along every line of sight. This can be achieved with the plot_maximum function.

```
[5]: cube.plot_maximum()
```



Here it's obvious to see that the CS emission has a ring-like morphology and extends out to about $2.5''$. This is always a good check to make as you can see whether the data is well centered or not.

2.4.2 `annulus`

The main focus of this tutorial, however, is working with the `annulus` class. This contains an ensemble of spectra extracted from the `linecube` based on some geometrical cuts (usually just a small radial range). This is useful because if we expect the disk to be azimuthally symmetric, then these spectra should have the same form (i.e., peak and width), but have their line centers shifted due to the projected velocity structure of the disk. Leveraging this assumption that the line profiles should *look* the same, we can use this to infer the underlying velocity structure.

Extracting an `annulus`

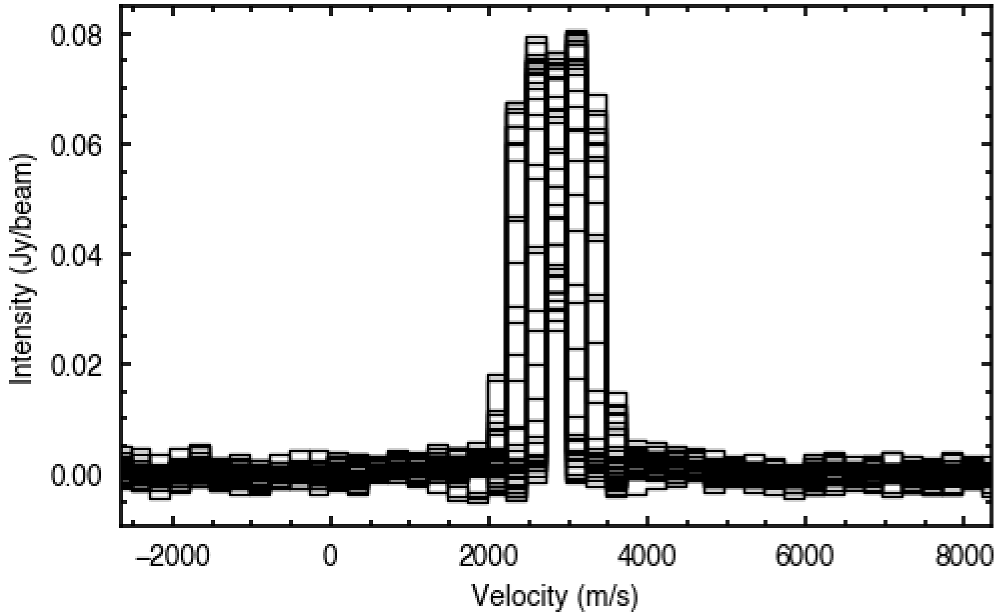
To extract an annulus, we simply use the `get_annulus` function, specifying the disk properties and the region we're interested in. By default, this will select a random sample of *spatially independent* pixels from the cube. Remember, if we want to assume that all the spectra look the same, then we only want a small radial range.

[6]:
```
annulus = cube.get_annulus(r_min=1.0, r_max=1.1, inc=6.5, PA=151.0)
```

Inspecting an `annulus`

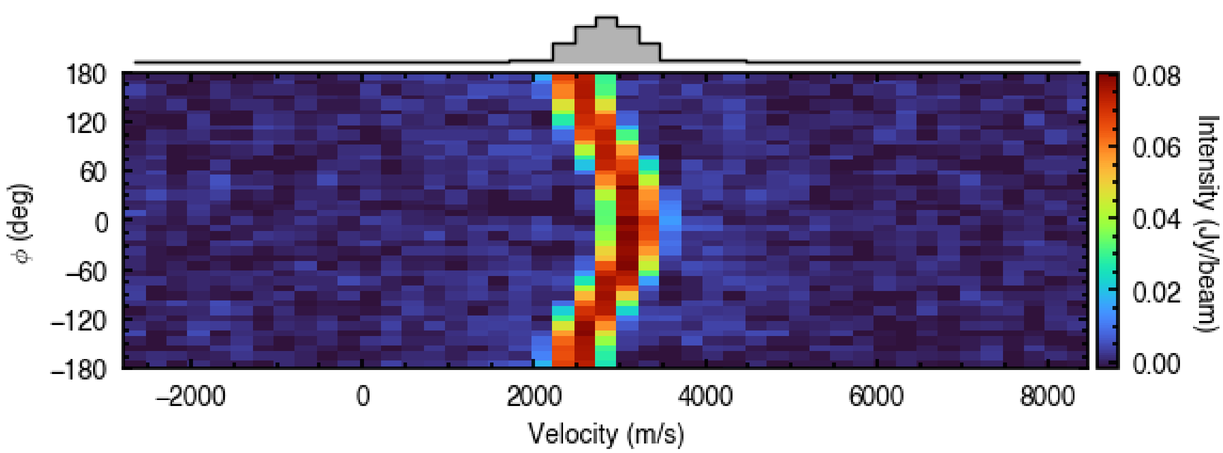
We can quickly see spectra that we've selected through `plot_spectra`:

[7]:
```
annulus.plot_spectra()
```



This figure shows that all the lines look similar, but are spread out along the velocity axis due to the Doppler shift of the lines. Another useful function is `plot_river`:

[8]:
```
annulus.plot_river()
```



What this figure shows with the colored panel are each of the spectra stacked on top of one another; each row represents a spectrum. So you can see at $\phi = 0^{\circ}$, the red-shifted axis (remember in `eddy` the PA is always measured to the red-shifted axis, and the deprojected polar coordinate ϕ is measured from this axis in an east-of-north direction) has the peak at around $3.2\ \mathrm{km\,s^{-1}}$, while the blue-shifted axis, $\phi = \pm 180^{\circ}$, has the peak around $2.4\ \mathrm{km\,s^{-1}}$.

The top panel shows the azimuthally averaged spectrum. This is exceptionally broad as we're averaging over spectra with a large range of line centroids.

Inferring Rotation Velocity

The most basic approach to accounting for this velocity shift is to model the line centroid as a very simple harmonic oscillator:

$$v_0(\phi) = v_\phi \cos(\phi) \sin(i) + v_{\rm LSR}$$

This can be easily done with the `get_vlos_SHO` function. In short, this determins the line centroid for each spectrum in the `annulus`, then fits v_ϕ and $v_{\rm LSR}$ to best recover the observation. By default, the function will use the `quadratic` method described in Teague & Foreman-Mackey (2018) to fit the line centroids, and will return both v_ϕ and $v_{\rm LSR}$ and their associated uncertainties.

```
[9]: annulus.get_vlos_SHO()
```

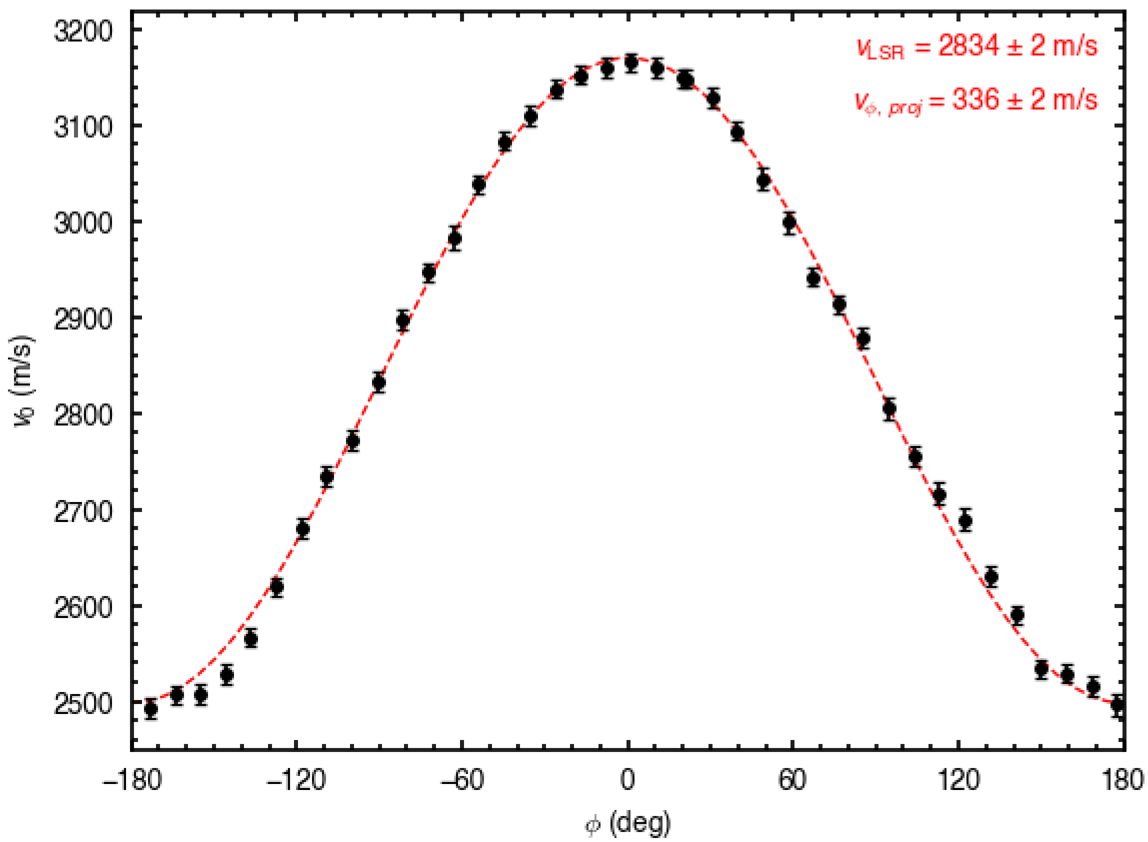
```
[9]: (array([2968.83394979, 2834.2600223 ]), array([18.90417071,  1.56812085]))
```

There are three methods implemented in `eddy` to determine the line centroid:

- `'quadratic'` - The method described in Teague & Foreman-Mackey (2018). This is only good when the line is only partially resolved.
- `'max'` - Assumes the line center is the velocity of the channel with the peak intensity in. This is the fastest, but is limited by the spectral resolution of the data and the noise.
- `'gaussian'` - Finds the line center by fitting a Gaussian profile to it.
- `'gaussthick'` - Finds the line center by fitting an optically thick Gaussian profile to it.
- `'doublegauss'` - Fits two Gaussian components to each spectrum, taking the larger of the two as the 'main' component. This is very chaotic and should be used with caution.

To understand the quality of the fit, we can use the convenience function, `plot_centroids` to show our data, and overplot the fit. Note this function also takes the `centroid_method` arugment to change between the different methods described above.

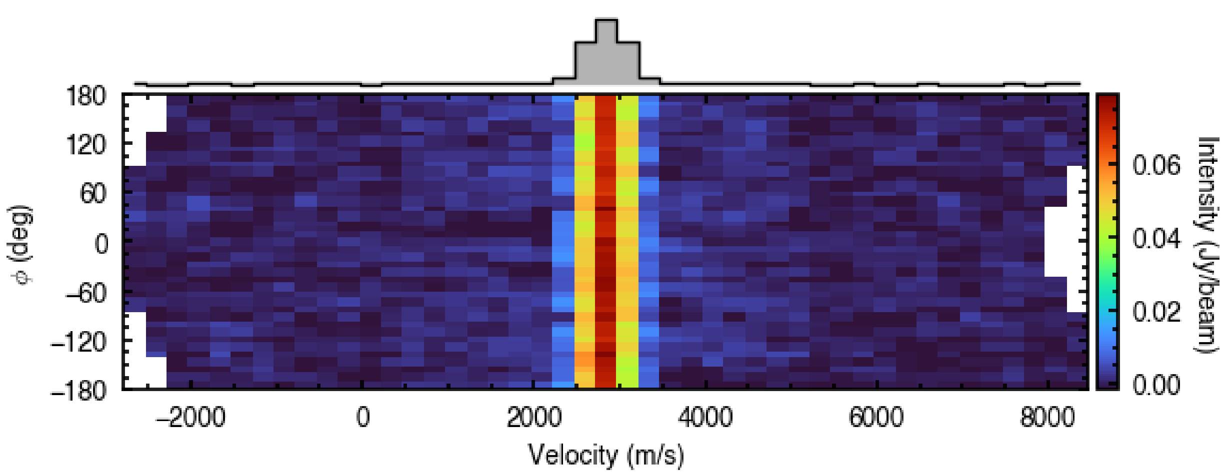
```
[10]: annulus.plot_centroids(plot_fit=True, centroid_method='quadratic')
```



Note that here what's annotated is $v_{\phi,\,\mathrm{proj}} = v_\phi \sin(i)$. In general, any symbol with the proj subscript means taking into account the projection from the disk inclination.

As another check that this is the correct velocity, we can use this to 'straighten out' the river plot from above by providing it the projected rotational velocity.

[11]:
```
annulus.plot_river(vrot=annulus.get_vlos_SHO()[0][0])
```



We can easily see that this correction has straightened out the river and tightened up the azimuthally averaged spectrum. It is this approach that we use with GoFish to tease out weak emission lines.

Radial and Vertical Velocities

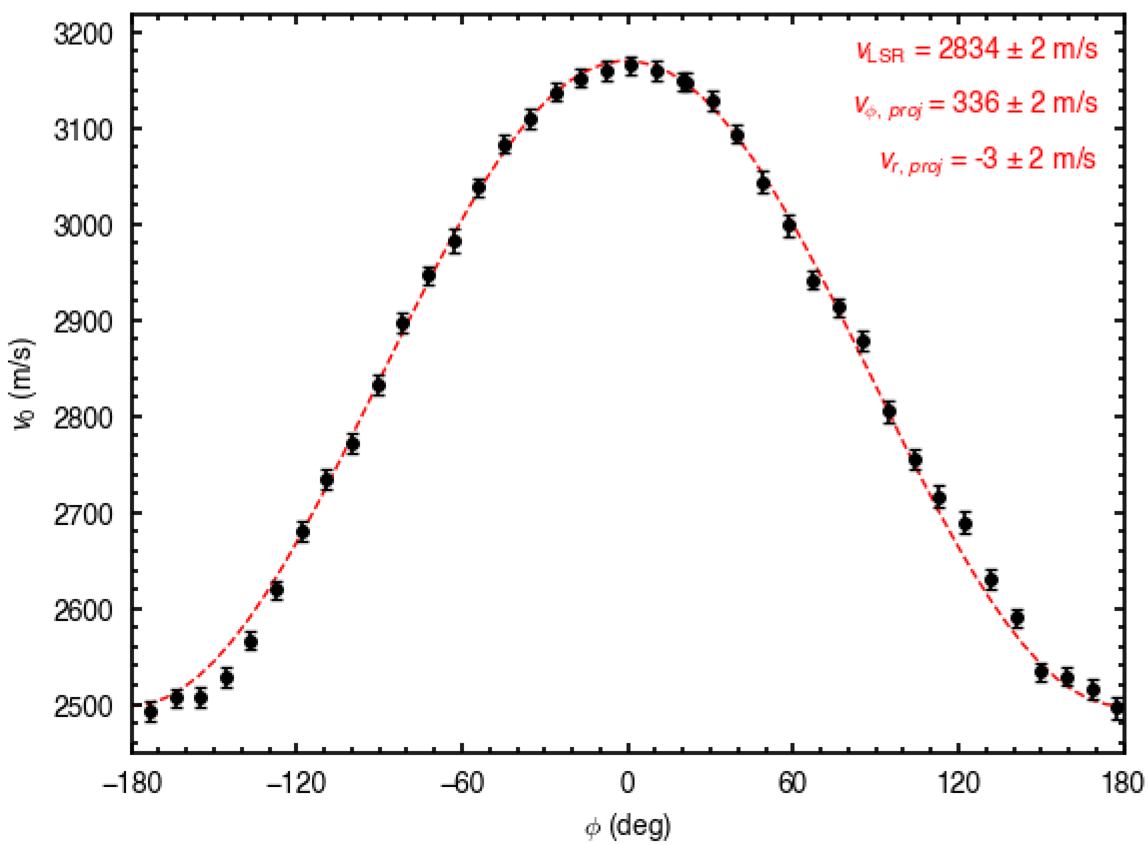
Radial and vertical velocities may also be present. We can extend the simple SHO model above to account for this:

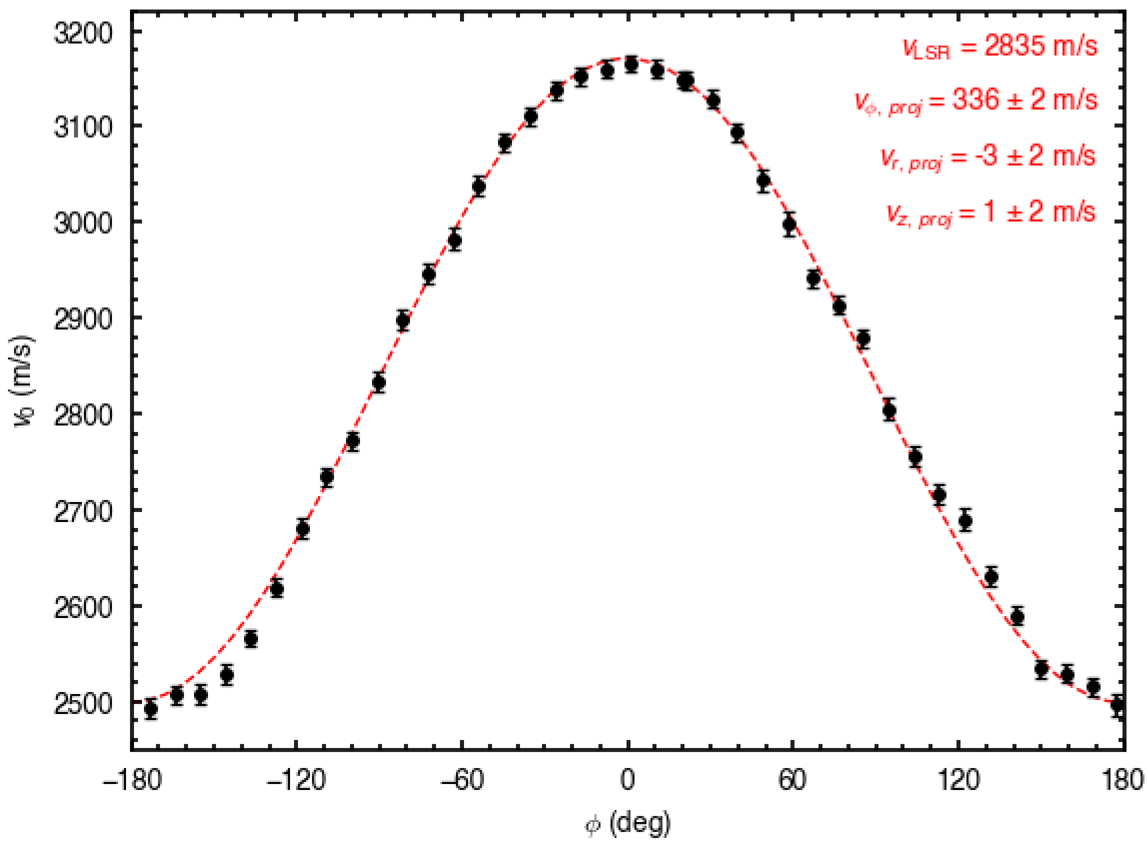
$$v_0(\phi) = v_\phi \cos(\phi) \sin(i) - v_{\rm r} \sin(\phi) \sin(i) - v_{\rm z} \cos(i) + v_{\rm LSR}.$$

Here we use the sign of the disk inclination, i, to encode the direction of rotation for the disk: a positive inclination means clockwise rotation, while a negative inclination means an anti-clockwise rotation. This rotation direction is inherited when using `get_annulus`. In this form, positive $v_{\rm r}$ values are moving *away* from the star and positive $v_{\rm z}$ values are moving away from the midplane.

To include the radial component, most functions allow for a `fit_vrad` argument. Similarly, we also provide a `fix_vlsr` value which specifies the systemic velocity of the disk. If this is not provided, what is returned is $v_{\rm LSR} - v_{\rm z} \cos(i)$. When `fix_vlsr` is provided, you will see that a $v_{\rm z,\,proj}$ is returned, while without it, only a $v_{\rm LSR}$ is returned

[12]:
```
annulus.plot_centroids(plot_fit=True, fit_vrad=True, centroid_method='quadratic')

annulus.plot_centroids(plot_fit=True, fit_vrad=True, fix_vlsr=2.835e3, centroid_
↪method='quadratic')
```





2.4.3 Velocity Profiles

The `linecube` class provides a wrapper for splitting the disk into concentric annuli assuming a source geometry, and then calculating the rotational and, if requested, radial velocity profiles.

NOTE: This approach is different to the one implemented in ConeRot which allows each annulus to be described by a different set of geometrical parameters.

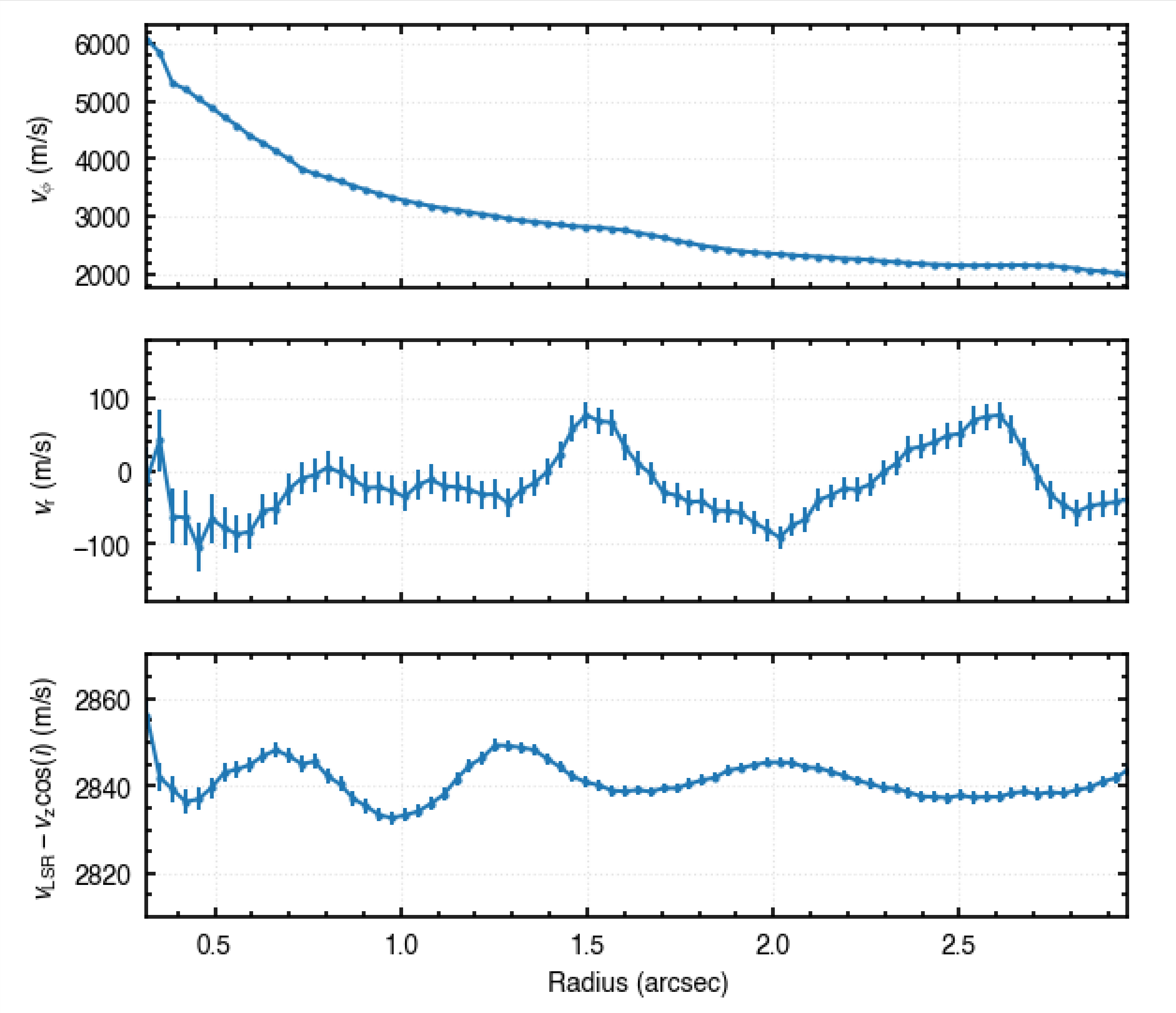
This is the `get_velocity_profile` function, as demonstrated below. By default it will calculate the profile for the whole image with bin annuli of 1/4 the beam major axis size, however for this we trim down the region to speed things up. For this Tutorial, we will stick with the `fit_method='SHO'`. Other fit methods are discussed in a second Tutorial.

```
[13]: r, v, dv = cube.get_velocity_profile(x0=0.0, y0=0.0, inc=6.0, PA=151.0,
                                          fit_vrad=True, fit_method='SHO',
                                          get_vlos_kwargs=dict(centroid_method='quadratic
      ↪'),
                                          rbins=np.arange(0.3, 3.0, 0.25 * cube.bmaj))
```

This function will return three arrays: the bin centers, the velocity profiles and the uncertainties on the velocity profiles. Note that this will return *deprojected* velocities, taking into account the disk projection. Here positive v_r are velocities moving *away* from the star, while positive v_z values are moving *away* from the disk midplane.

In this example above we didn't set the `fix_vlsr` value, such that the third velocity component is the sum of the systemic velocity and the projection of the vertical component (this should be clear as the values are typically measured in the km/s rather than the m/s).

[14]:
```
fig, axs = plt.subplots(figsize=(6.75, 6.17), nrows=3)

axs[0].grid(ls=':', color='0.9')
axs[0].errorbar(r, v[0], dv[0], marker='o', ms=2)
axs[0].set_xticklabels([])
axs[0].set_ylabel(r'$v_{\rm \phi}$' + ' (m/s)')

axs[1].grid(ls=':', color='0.9')
axs[1].errorbar(r, v[1], dv[1], marker='o', ms=2)
axs[1].set_xticklabels([])
axs[1].set_ylabel(r'$v_{\rm r}$' + ' (m/s)')
axs[1].set_ylim(-180, 180)

axs[2].grid(ls=':', color='0.9')
axs[2].errorbar(r, v[2], dv[2], marker='o', ms=2)
axs[2].set_xlabel('Radius (arcsec)')
axs[2].set_ylabel(r'$v_{\rm LSR} - v_{\rm z} \cos(i)$' + ' (m/s)')
axs[2].set_ylim(2810, 2870)

for ax in axs:
    ax.set_xlim(r[0], r[-1])
fig.align_labels(axs)
```



Repeating the above, but now specifying `fix_vlsr`, we can see that the third velocity component is now v_z.

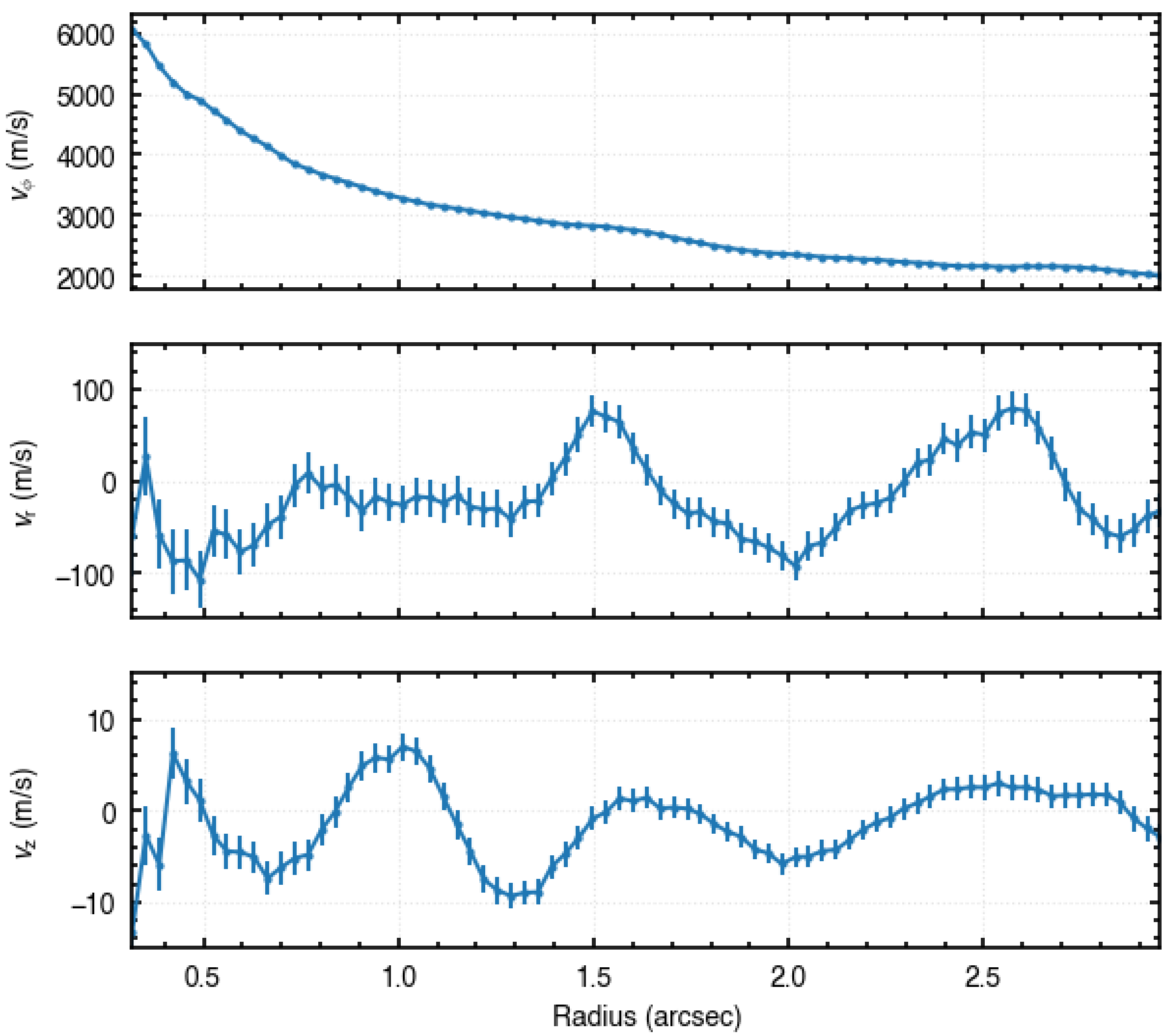
[15]:
```
r, v, dv = cube.get_velocity_profile(x0=0.0, y0=0.0, inc=6.0, PA=151.0,
                                     fit_vrad=True, fix_vlsr=2.84e3, fit_method='SHO',
                                     get_vlos_kwargs=dict(centroid_method='quadratic
↪'),
                                     rbins=np.arange(0.3, 3.0, 0.25 * cube.bmaj))
```

[16]:
```
fig, axs = plt.subplots(figsize=(6.75, 6.17), nrows=3)

axs[0].grid(ls=':', color='0.9')
axs[0].errorbar(r, v[0], dv[0], marker='o', ms=2)
axs[0].set_xticklabels([])
axs[0].set_ylabel(r'$v_{\rm \phi}$' + ' (m/s)')

axs[1].grid(ls=':', color='0.9')
axs[1].errorbar(r, v[1], dv[1], marker='o', ms=2)
axs[1].set_xticklabels([])
axs[1].set_ylabel(r'$v_{\rm r}$' + ' (m/s)')
```

(continues on next page)

(continued from previous page)

```
axs[1].set_ylim(-150, 150)

axs[2].grid(ls=':', color='0.9')
axs[2].errorbar(r, v[2], dv[2], marker='o', ms=2)
axs[2].set_xlabel('Radius (arcsec)')
axs[2].set_ylabel(r'$v_{\rm z}$' + ' (m/s)')
axs[2].set_ylim(-15, 15)

for ax in axs:
    ax.set_xlim(r[0], r[-1])
fig.align_labels(axs)
```

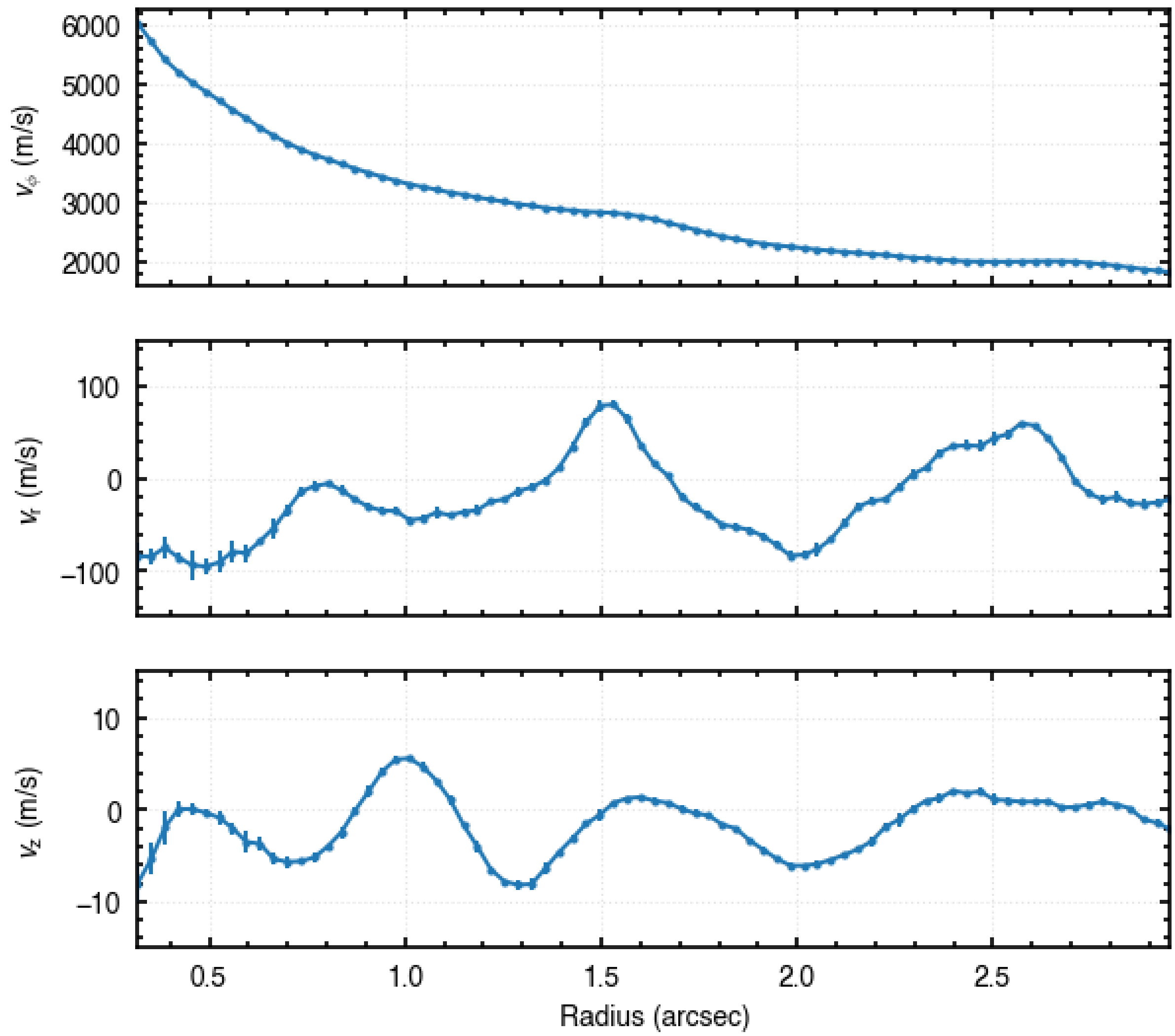


Multiple Iterations

This approch often yields uncertainties (if they do at all!) that are implausibly small and more than likely reflect the inflexibility in the model. One approach to circumvent this is use the `niter` argument to calculate several different velocity profiles, each using annuli with different pixels (at least statistically, each annulus is taken with a random draw of pixels), and then taking a weighted average over the samples.

[17]:
```
r, v, dv = cube.get_velocity_profile(x0=0.0, y0=0.0, inc=6.0, PA=151.0,
                                     fit_vrad=True, fix_vlsr=2.84e3, fit_method='SHO',
                                     get_vlos_kwargs=dict(centroid_method='gaussian'),
                                     rbins=np.arange(0.3, 3.0, 0.25 * cube.bmaj),
                                     niter=5)
```

[18]:
```
fig, axs = plt.subplots(figsize=(6.75, 6.17), nrows=3)

axs[0].grid(ls=':', color='0.9')
axs[0].errorbar(r, v[0], dv[0], marker='o', ms=2)
axs[0].set_xticklabels([])
axs[0].set_ylabel(r'$v_{\rm \phi}$' + ' (m/s)')

axs[1].grid(ls=':', color='0.9')
axs[1].errorbar(r, v[1], dv[1], marker='o', ms=2)
axs[1].set_xticklabels([])
axs[1].set_ylabel(r'$v_{\rm r}$' + ' (m/s)')
axs[1].set_ylim(-150, 150)

axs[2].grid(ls=':', color='0.9')
axs[2].errorbar(r, v[2], dv[2], marker='o', ms=2)
axs[2].set_xlabel('Radius (arcsec)')
axs[2].set_ylabel(r'$v_{\rm z}$' + ' (m/s)')
axs[2].set_ylim(-15, 15)

for ax in axs:
    ax.set_xlim(r[0], r[-1])
fig.align_labels(axs)
```



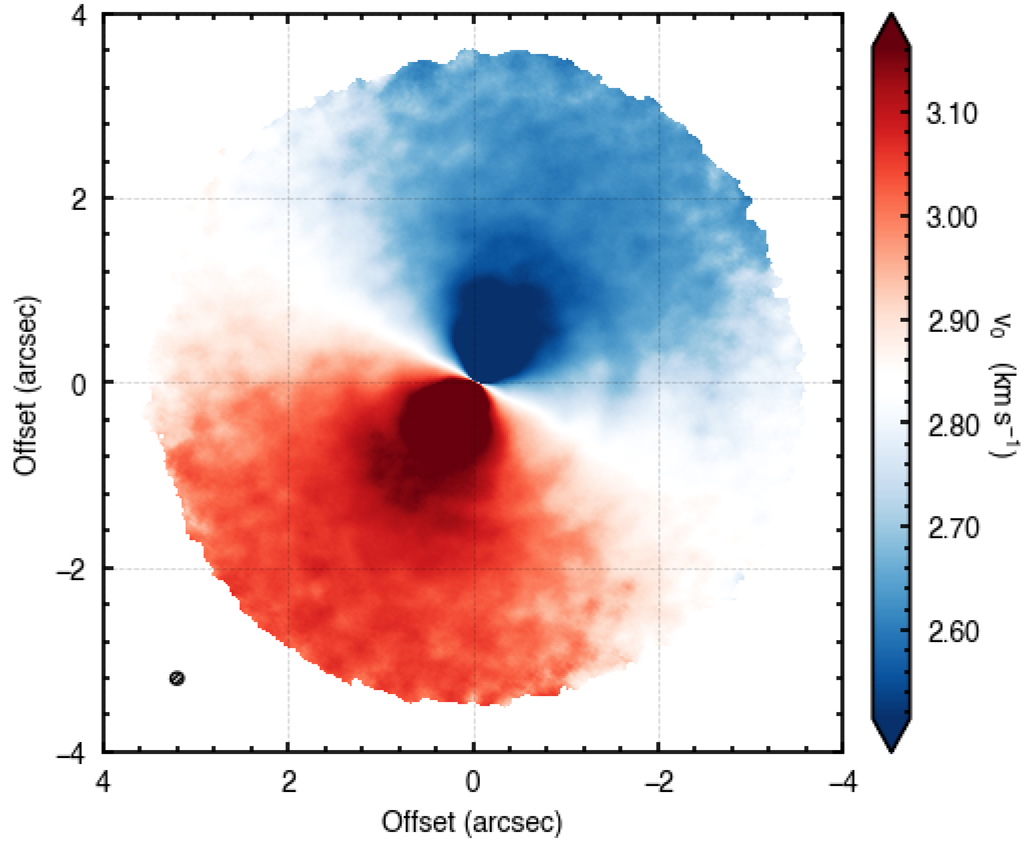
A `rotationmap` Wrapper

You may have noticed that this approach of splitting the data into annuli, calculating the centroids of each spectrum within each annulus and then fitting a SHO model can be accelerated if we already have a map of the line centroids, as we worked with in the previous tutorial.

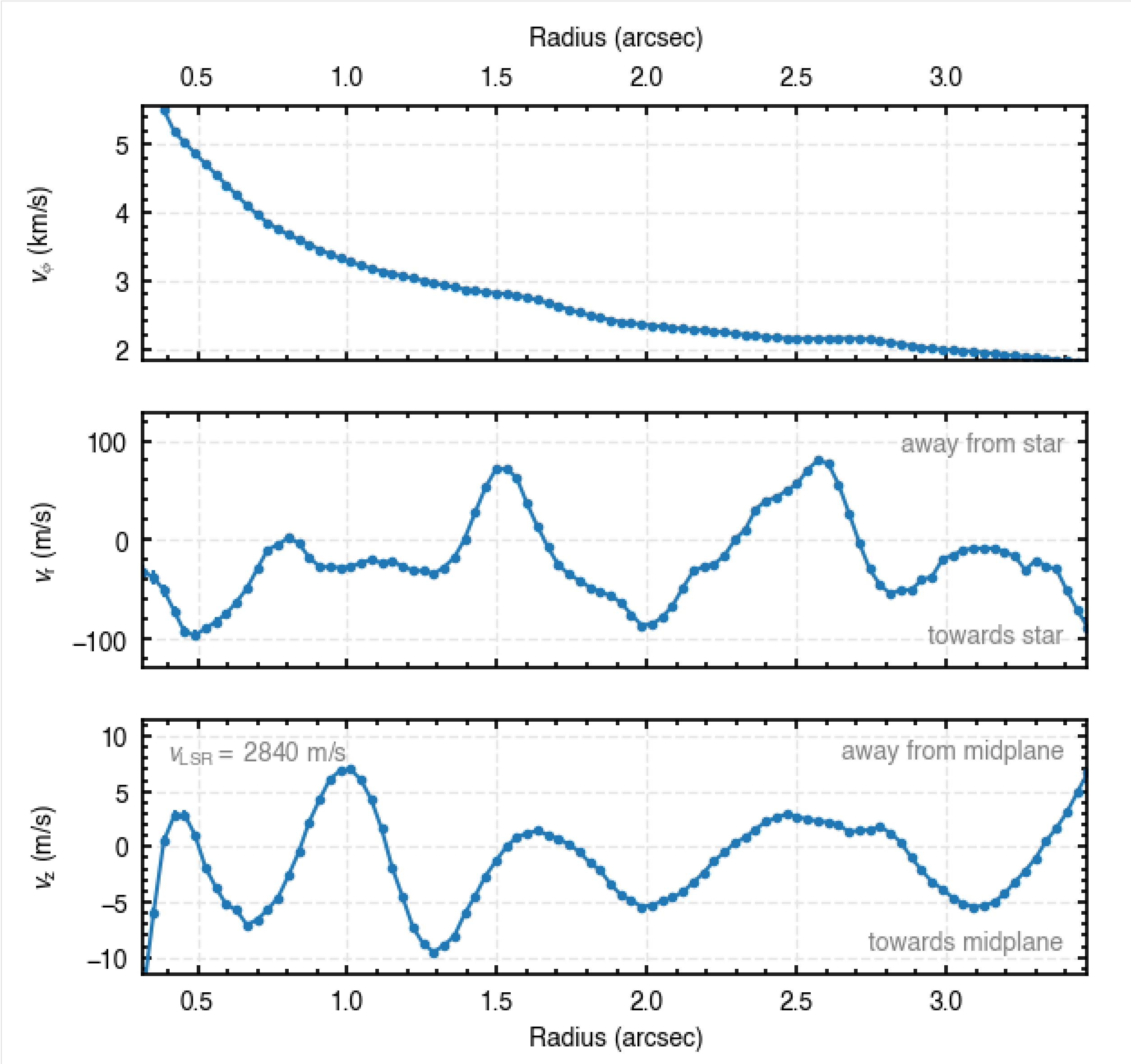
In fact, `rotationmap` has a similar functionailty, `fit_annuli`, which performs the same process, but without having to calculate the line centroids each time. This also has the option to return the linearly interpolated model and residuals using the same `returns` argument as found for `fit_map`. There are many more options for this function, and we encourage the reader to read the documentation for find more.

[19]:
```
from eddy import rotationmap
import numpy as np
cube = rotationmap('TWHya_CO_cube_v0.fits', FOV=8.0)
cube.plot_data()
```

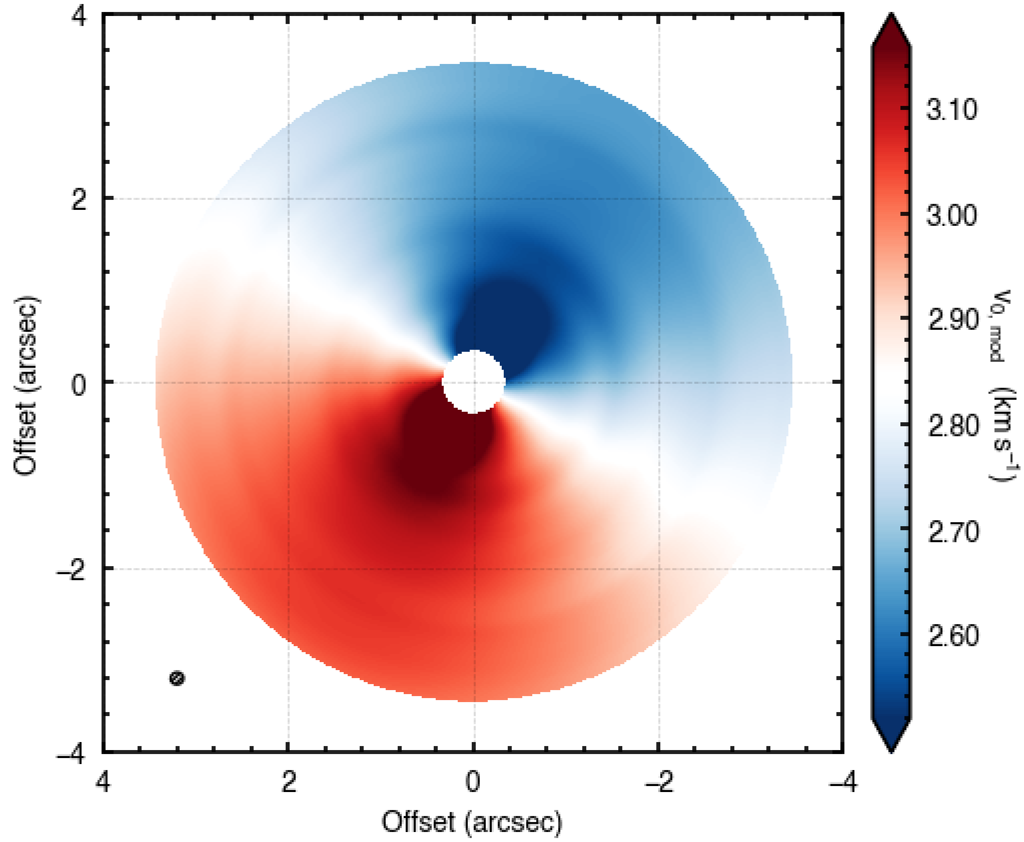
```
Assuming uncertainties in TWHya_CO_cube_dv0.fits.
```

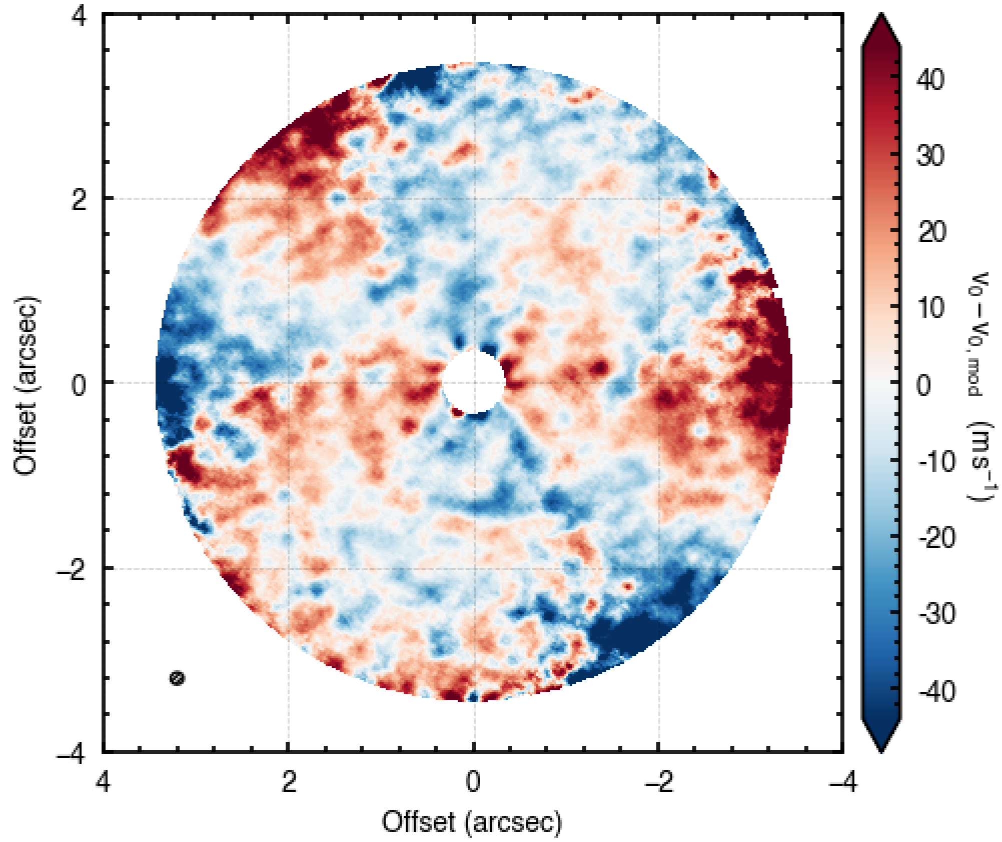



```
[20]: r, v, dv = cube.fit_annuli(x0=0.0, y0=0.0, inc=6.0, PA=151.0,
                                 fit_vrad=True, fix_vlsr=2.84e3,
                                 rbins=np.arange(0.3, 3.5, 0.25 * cube.bmaj))
```



Radius (arcsec)
0.5
1.0
1.5
2.0
2.5
3.0
v_ϕ (km/s)
5
4
3
2
v_r (m/s)
100
0
−100
away from star
towards star
v_z (m/s)
10
5
0
−5
−10
$v_{\rm LSR}$ = 2840 m/s
away from midplane
towards midplane


Offset (arcsec)
Offset (arcsec)
4
2
0
–2
–4
3.10
3.00
2.90
2.80
2.70
2.60
$v_{0,\,\mathrm{mod}}$ (km s^{-1})



2.5 4 - Different Methods for Inferring Velocities

In the previous tutorial, we saw how modeling the line center of an annulus of spectra as a simple harmonic oscillator provided a good way to extract radial profiles of v_ϕ and v_r. While this was a quick approach, we essentially threw data away by collapsing each spectrum into a single value: the line center. In this tutorial, we will look at alternative methods to infer the velocity structure by leveraging as much information as possible.

2.5.1 Background

To demonstrate the power of using a full spectrum in this endeavor, let's look at a river plot again. The figure below shows a river plot for an annulus between $1.5''$ to $1.6''$ in TW Hya. It cycles through different v_ϕ values, shown in the top left of the bottom panel.

As can be clearly seen, when v_ϕ approaches the correct value, the river straightens out and the azimuthally averaged spectrum, shown along the top, aproaches a high SNR Gaussian profile. There have been several approaches which have been described in the literature that leverage this fact. We'll explore them below.

2.5.2 Deprojection Basics

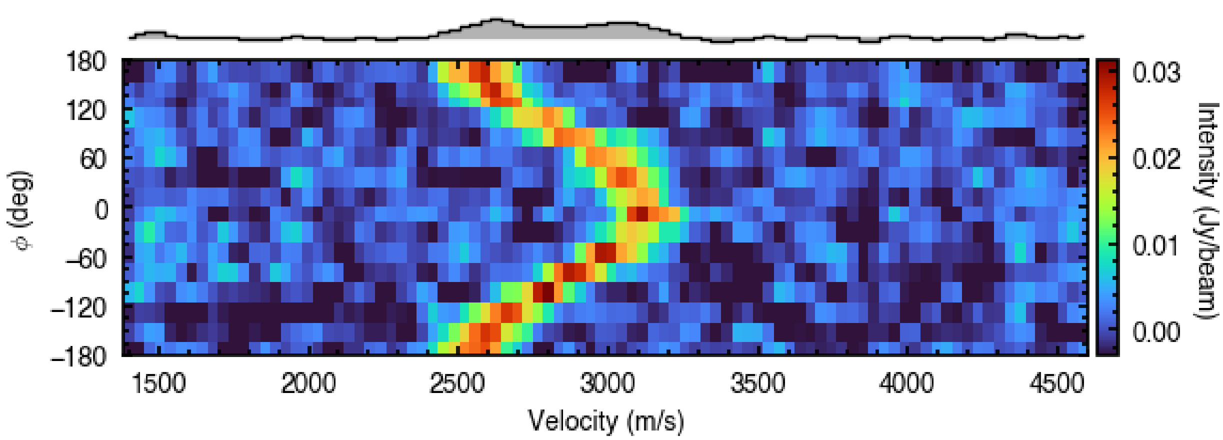
First of all, we need to explore some of the features of the `annulus` class. Let's load up a cube and grab an annulus. Here we're using a slightly different example cube that has CS emission from Teague et al. (2018b) and is available for download from the `GoFish` Dataverse.

[1]:
```
import os
if not os.path.exists('TWHya_CS_cube.fits'):
    !wget -O TWHya_CS_cube.fits -q https://dataverse.harvard.edu/api/access/datafile/:
↪persistentId?persistentId=doi:10.7910/DVN/LO2QZM/KYZFUQ
```

[2]:
```
import matplotlib.pyplot as plt
from eddy import linecube
import numpy as np
```

[3]:
```
cube = linecube('TWHya_CS_cube.fits', FOV=8.0)
```

[4]:
```
annulus = cube.get_annulus(r_min=1.5, r_max=1.6, inc=6.0, PA=151.0)
annulus.plot_river()
```



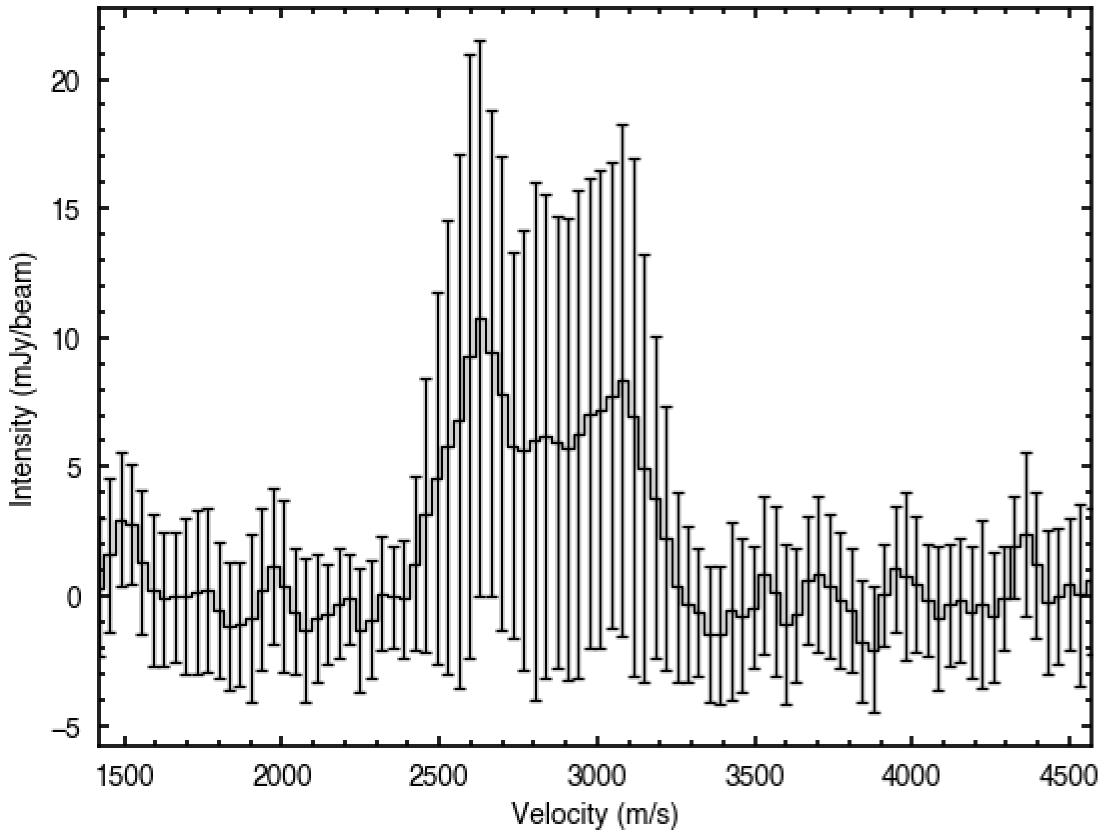
Deprojected Spectrum

One of the fundamental functions of the `annulus` class is the `deprojected_spectrum` function. This shifts each spectrum by the projected velocity component decribed by $(v_\phi,\, v_\mathrm{r})$ and then averages of all spectra. This is what produces the spectrum on the top of the river plot.

[5]:
```
x, y, dy = annulus.deprojected_spectrum(vrot=0.0)
```

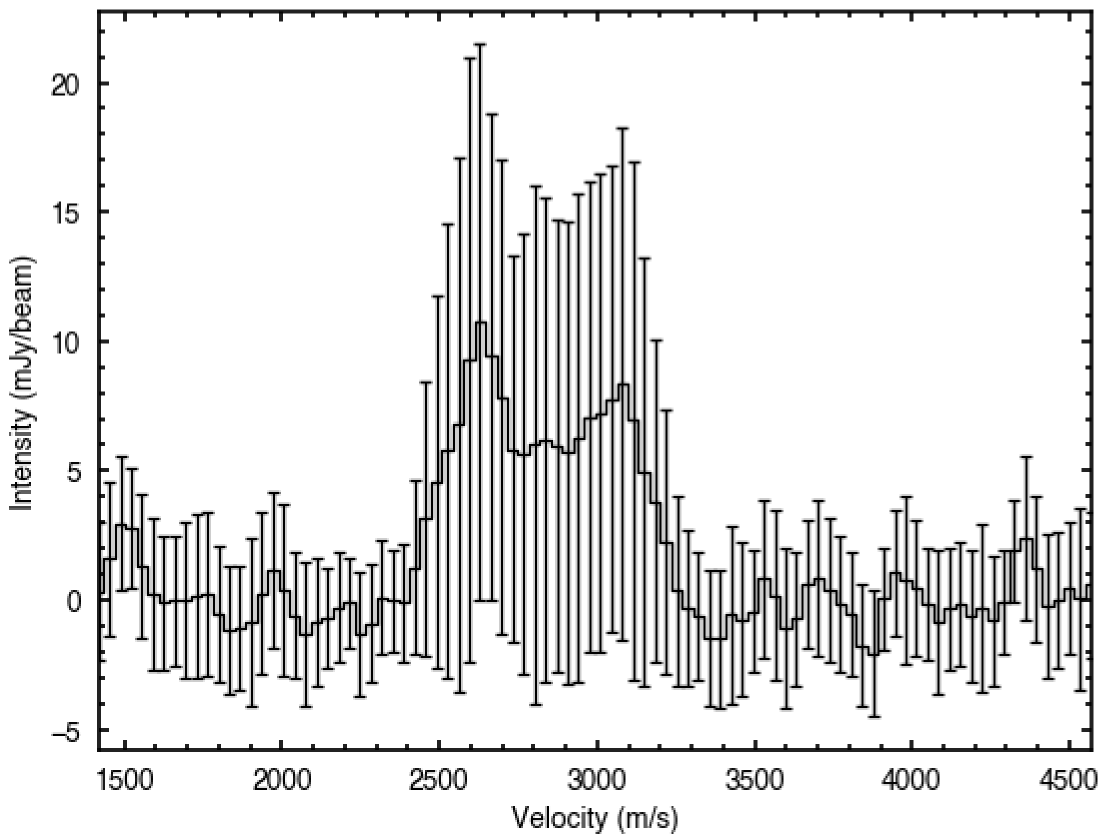
[6]:
```
fig, ax = plt.subplots()
ax.errorbar(x, y * 1e3, dy * 1e3, fmt=' ', capsize=2.0, lw=1.0, color='k')
ax.step(x, y * 1e3, where='mid', lw=1.0, color='k')
ax.set_ylabel('Intensity (mJy/beam)')
ax.set_xlabel('Velocity (m/s)')
ax.set_xlim(x.min(), x.max())
```

[6]:
```
(1417.3097826192488, 4567.690222234542)
```



The `annulus` class has a convenience function, `plot_spectrum` which will align, stack and plot the spectra for you!

[7]:
```
annulus.plot_spectrum(vrot=0.0)
```

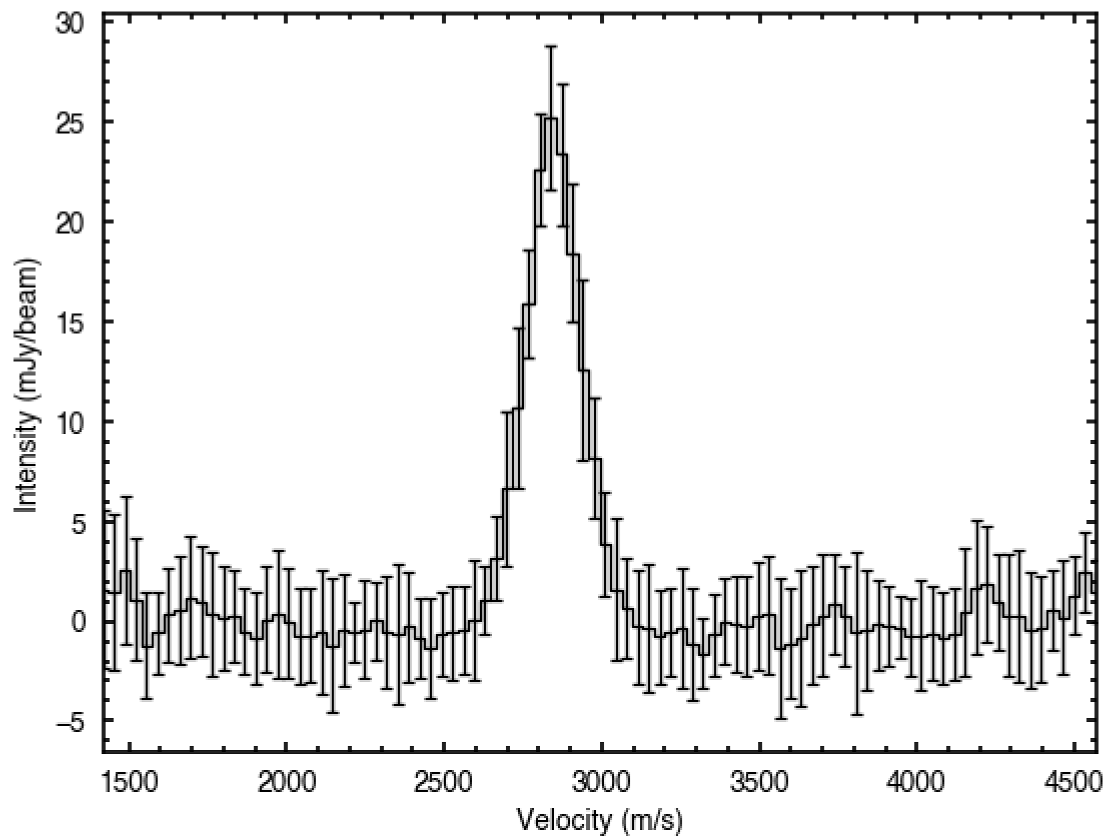


Just as we did for the `plot_river` function, we can provide the correct rotation velocity to get a nicer looking spectrum. For now, let's use the `fit_SHO` approach to infer the velocity profile.

[8]:
```
v, dv = annulus.get_vlos(fit_method='SHO')
v_phi, dv_phi = v[0], dv[0]
print('v_phi = {:.0f} +\- {:.0f} m/s'.format(v_phi, dv_phi))
```

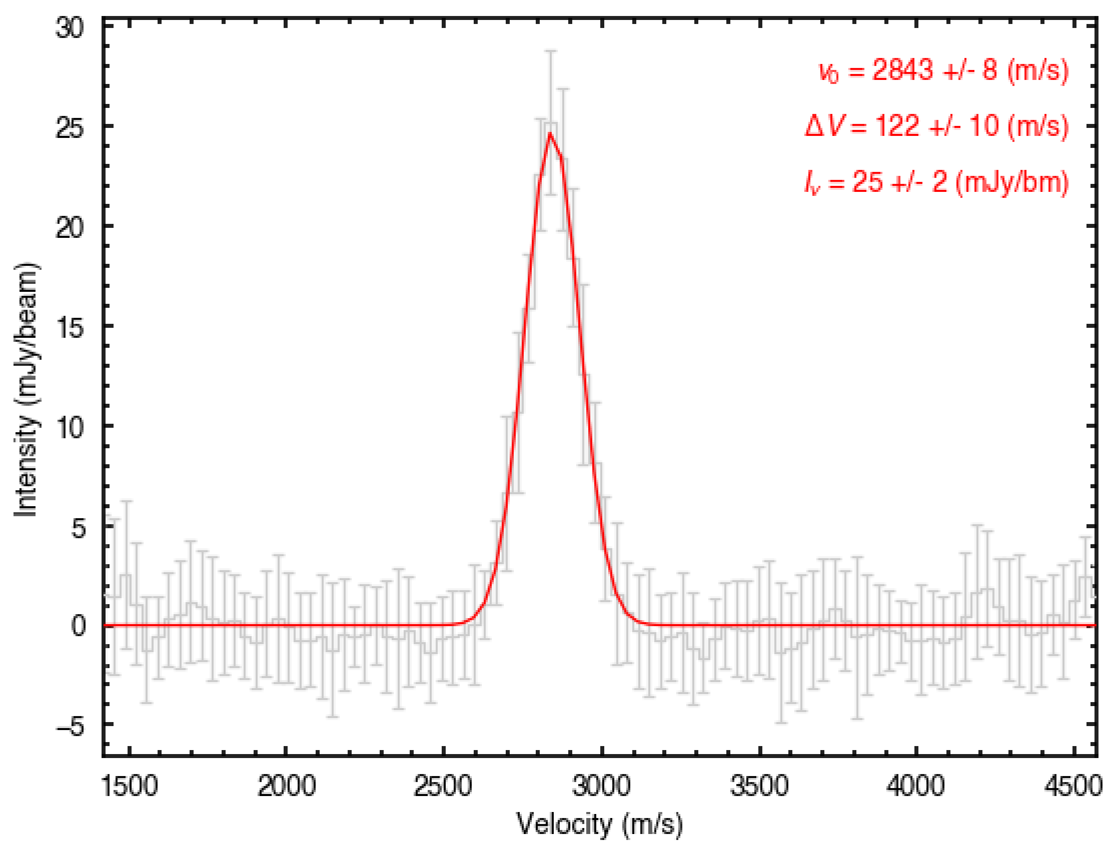
```
v_phi = 2613 +\- 65 m/s
```

[9]:
```
annulus.plot_spectrum(vrot=v_phi)
```



The `deprojected_spectrum` also takes a `vrad` argument in case you have an idea of what $v_{\rm r}$ should be. In addition, we can overplot a Gaussian fit to the data to check how well the aligning has done.

[10]:
```
annulus.plot_spectrum(vrot=v_phi, plot_fit=True)
```



Resampled Spectra

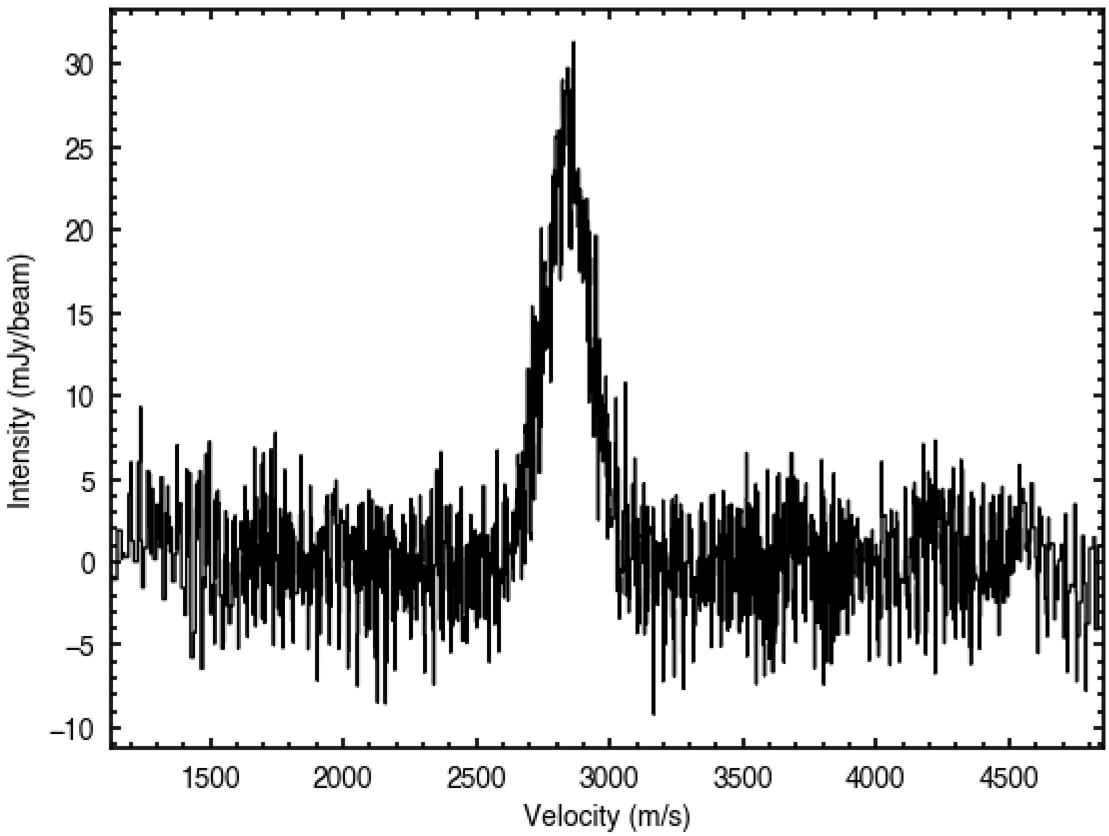
Another powerful aspect of the `deprojected_spectrum` function is that you can resample your averaged spectrum to a finer channel spacing, as done in Teague & Loomis (2020). This works because for each spectrum at a polar angle ϕ, the Doppler shift is not quantized into channels meaning that we sample the underlying spectrum at a much higher rate. This can be seen when we use `resample=False`.

[11]:
```
x, y, dy = annulus.deprojected_spectrum(vrot=v_phi, resample=False)
```

[12]:
```
print('mean spacing = {:.1f} m/s'.format(np.diff(x).mean()))
```

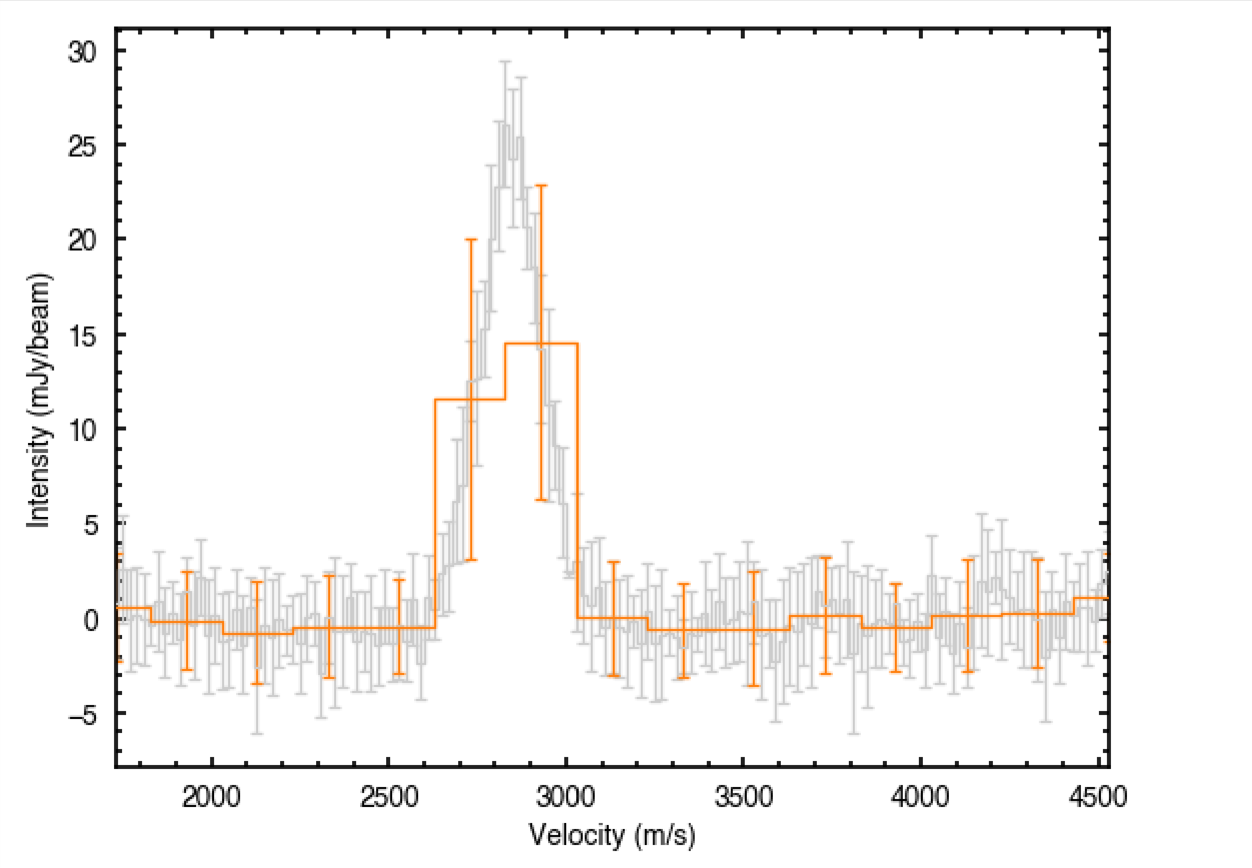
```
mean spacing = 2.5 m/s
```

[13]:
```
annulus.plot_spectrum(vrot=v_phi, resample=False)
```



The `resample` argument is actually quite versatile. If we provide it a float, this specifies the new channel spacing (e.g., `resample=10.0` will return the spectrum binned down to 10 m/s channels), while an integer specifies the factor that the current spacing is increased by (e.g., `resample=4` will return a spectrum with a sampling rate 4 times higher than the native data.

[14]:
```
# resample at 20 m/s
fig = annulus.plot_spectrum(vrot=v_phi, resample=20.0,
                            plot_kwargs=dict(color='0.8'), return_fig=True)

# resample at 200 m/s
annulus.plot_spectrum(vrot=v_phi, resample=200.0,
                      ax=fig.axes[0], plot_kwargs=dict(color='C1'))
```



With this functionality in mind, we can look at how we can used these deprojected spectra to infer the correct velocity.

2.5.3 Maximizing Signal to Noise

Yen et al. (2018) proposed that the correct velocity should maximize the signal-to-noise of the averaged spectrum. This because of two main aspects:

1) As the specrta are aligned, they will average coherently and so will result in the largest peak intensity (modulo background noise). You can see this by comparing the peak of the deprojected spectra above with and without the correct alignment.

2) Averaging over many samples beats down the noise. For independent spectra we would expect a $\sim\sqrt{N}$ improvement in the noise measured in the spectrum when averaging over N spectra. We have the added advantage with the velocity shifting in that some spatial correlations are decorrelated due to the shifting of the spectra (see Yen et al., 2016, for a discussion about this).

This approach can get used in the `get_vlos` function using `fit_method='SNR'`.

```
[15]: print('v_phi = {:.0f} m/s'.format(annulus.get_vlos(fit_method='SNR')[0][0]))
```

```
v_phi = 2654 m/s
```

This approach (as it is currently coded) has two issues. Firstly, it does not return an uncertainty on v_ϕ as it simply uses `scipy.optimize.minimize` to minimize the negative SNR of the deprojected spectrum.

The other is that the calculation of the SNR involves several steps: measuring the 'signal' of the spectrum and measuring the 'noise' of the spectrum. The signal is calculated either by taking the integrated intensity of the deprojected spectrum (`signal='int'`, the default), or the peak intensity (`signal='max'`). As both of these quantities are very noise as a function of v_ϕ, the `annulus` class will fit a Gaussian profile to the deprojected spectrum to calculate these quantities.

Yen et al. (2018) also discuss this issue, and advocate for using a Gaussian weighting when calculating the SNR, such that high intensity parts of the spectrum count more to this statistic. This can be used with `signal='weighted'`.

2.5.4 Minimizing Line Width

To circumvent noisiness of the SNR statistic, Teague et al. (2018a) advocated for using the line width as a proxy for alignment. When the spectra are aligned correctly, the width of the averaged spectrum should be minimized. This can be used with the `fit_method='dV'` argument.

[16]:
```
print('v_phi = {:.0f} m/s'.format(annulus.get_vlos(fit_method='dV')[0][0]))
```

```
v_phi = 2605 m/s
```

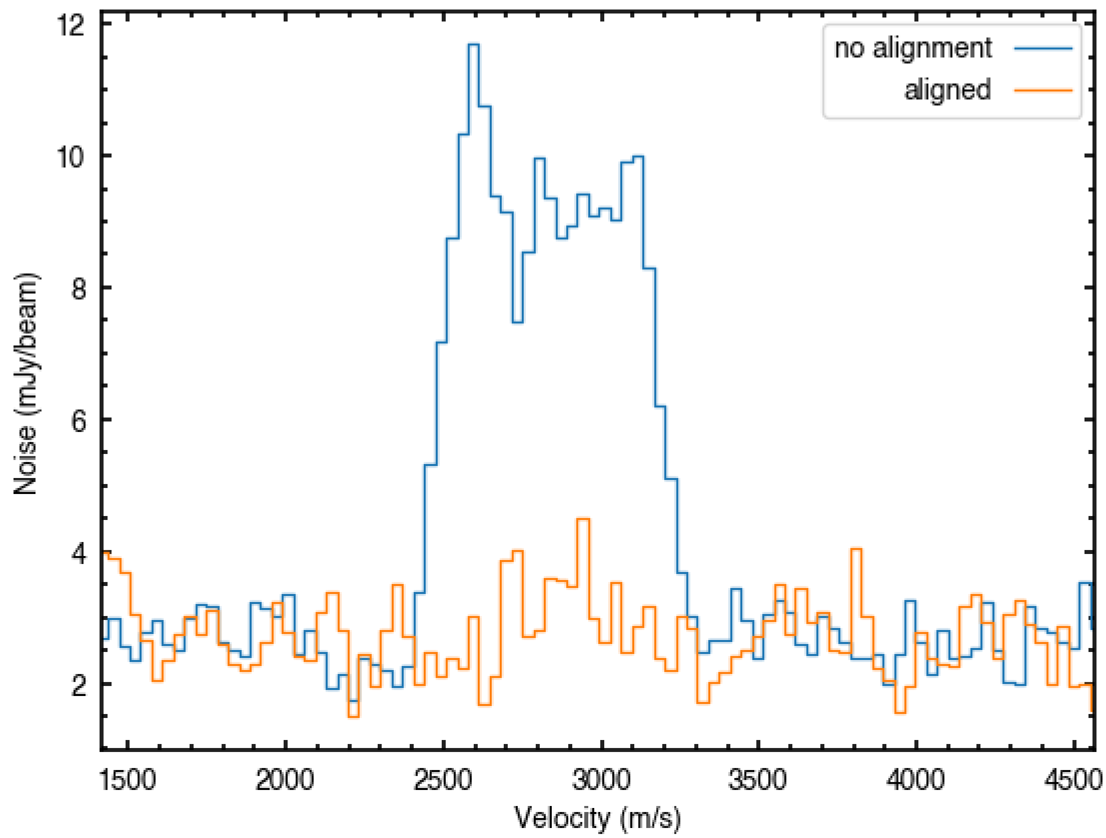
Again, this method (as it is currently coded) does not return an uncertainly. In addition, this approach requires the fitting of a Gaussian profile to the averaged spectrum which may fail for noisy spectra or poor starting conditions. One particular worry is that if the underlying profile is *not* a Gaussian, then both these methods will fail.

2.5.5 Gaussian Processes

To remove any dependence on the underlying line profile, Teague et al. (2018c) showed that modeling the underlying spectrum as a Gaussian Process can provide excellent results. In essence, this approach asks for what v_ϕ will the noise in the spectrum be minimized (and spectrally independent). Consider how the uncertainty changes for the case of no alignment and the correct alignment.

[17]:
```
fig, ax = plt.subplots()

x, y, dy = annulus.deprojected_spectrum(vrot=0.0)
ax.step(x, dy * 1e3, where='mid', lw=1.0, label='no alignment')

x, y, dy = annulus.deprojected_spectrum(vrot=v_phi)
ax.step(x, dy * 1e3, where='mid', lw=1.0, label='aligned')

ax.set_ylabel('Noise (mJy/beam)')
ax.set_xlabel('Velocity (m/s)')
ax.set_xlim(x.min(), x.max())
ax.legend(markerfirst=False)
```

[17]:
```
<matplotlib.legend.Legend at 0x1479ebf10>
```



The noise rockets around the line center when the alignment is incorrect due to the misalignment of the spectra.

To implement the fitting `eddy` will use the `emcee` MCMC sampler and celerite for the Gaussian Processes, and can be called with the `fit_method='GP'` argument. To help with the MCMC, this function takes the commonly used `nwalkers`, `nburnin` and `nsteps` arguments that were used in earlier tutorials fitting the rotation maps. By default, `get_vlos` will return the median value of the posterior samples and half the 84th to 16th percentile as the uncertainty.

[18]:
```
print('v_phi = {:.0f} +\- {:.0f} m/s'.format(*np.squeeze(annulus.get_vlos(fit_method=
↪'GP'))))
```

```
100%|| 1000/1000 [00:06<00:00, 161.52it/s]
```

```
v_phi = 2687 +\- 9 m/s
```

2.5.6 Simple Harmonic Oscillator

As we saw in the previous tutorial we can also fit the line centroids with a simple harmonic oscillator model.

[19]:
```
v, dv = annulus.get_vlos(fit_method='SHO')
print('v_phi = {:.0f} +\- {:.0f} m/s'.format(v[0], dv[0]))
```

```
v_phi = 2613 +\- 65 m/s
```

2.5.7 More Control

In fact, `get_vlos` is just a convenience wrapper for the underlying functions: `get_vlos_*` where `*` is the `fit_method` provide to `get_vlos`. These functions will have more functionality to aid in the fitting, such as producing plots of the walkers or posteriors for the `GP` method.

[1]:
```
from eddy import rotationmap
import numpy as np
```

2.6 5 - Self-Gravity and Pressure Supported Disks

It is becoming ever apparent that the background motion in a disk is subtly perturbed away from a pure Keplerian profile of $v_{\phi} \propto r^{-0.5}$ due to either the background pressure support slowing the rotation (e.g., Dullemond et al., 2020) or the self-gravity of the disk hastening the rotation (e.g., Veronesi et al., 2021). If the goal of your science is to better understand these properties then undertaking a full modeling of the v_{ϕ} profile is the best idea. However, if your goal is just to remove some average background profile to search for localized deviations then parameterization adopted in `eddy` might be worth a shot.

2.6.1 Parameterization

For a 2D, purely Keplerian disk the rotation profile is given by

$$v_{\phi} = \sqrt{\frac{GM_*}{r}},$$

where M_* is the dynamical mass of the central star. This can be modified to include an additional radial-dependent mass component, $M_d(r)$, that describe the disk mass *within* cylindrical radius r (we explain why we have adopted this simplification later on). This gives a modified rotation curve of

$$v_{\phi} = \sqrt{\frac{G(M_* + M_d(r))}{r}}.$$

Assuming a power-law surface density profile of the form, $\Sigma(r) = \Sigma_0 \times r^{-\gamma}$, then $M_d(r)$ is given by

$$M_d(r) = M_{\rm disk} \times \frac{r^{2-\gamma} - r_{\rm in}^{2-\gamma}}{r_{\rm out}^{2-\gamma} - r_{\rm in}^{2-\gamma}},$$

with $r_{\rm in}$ and $r_{\rm out}$ describing the disk inner and outer edge, respectively, and $M_{\rm disk}$ is the total disk mass. Note that for the commonly assumed $\gamma = 1$ case this is a simple linear interpolation between $r_{\rm in}$ and $r_{\rm out}$.

These parameters can be controlled with the parameter names `'mdisk'`, `'gamma'`, `'r_in'` and `'r_out'` (note here to differentiate these from the `'r_min'` and `'r_max'` parameter that describe the mask).

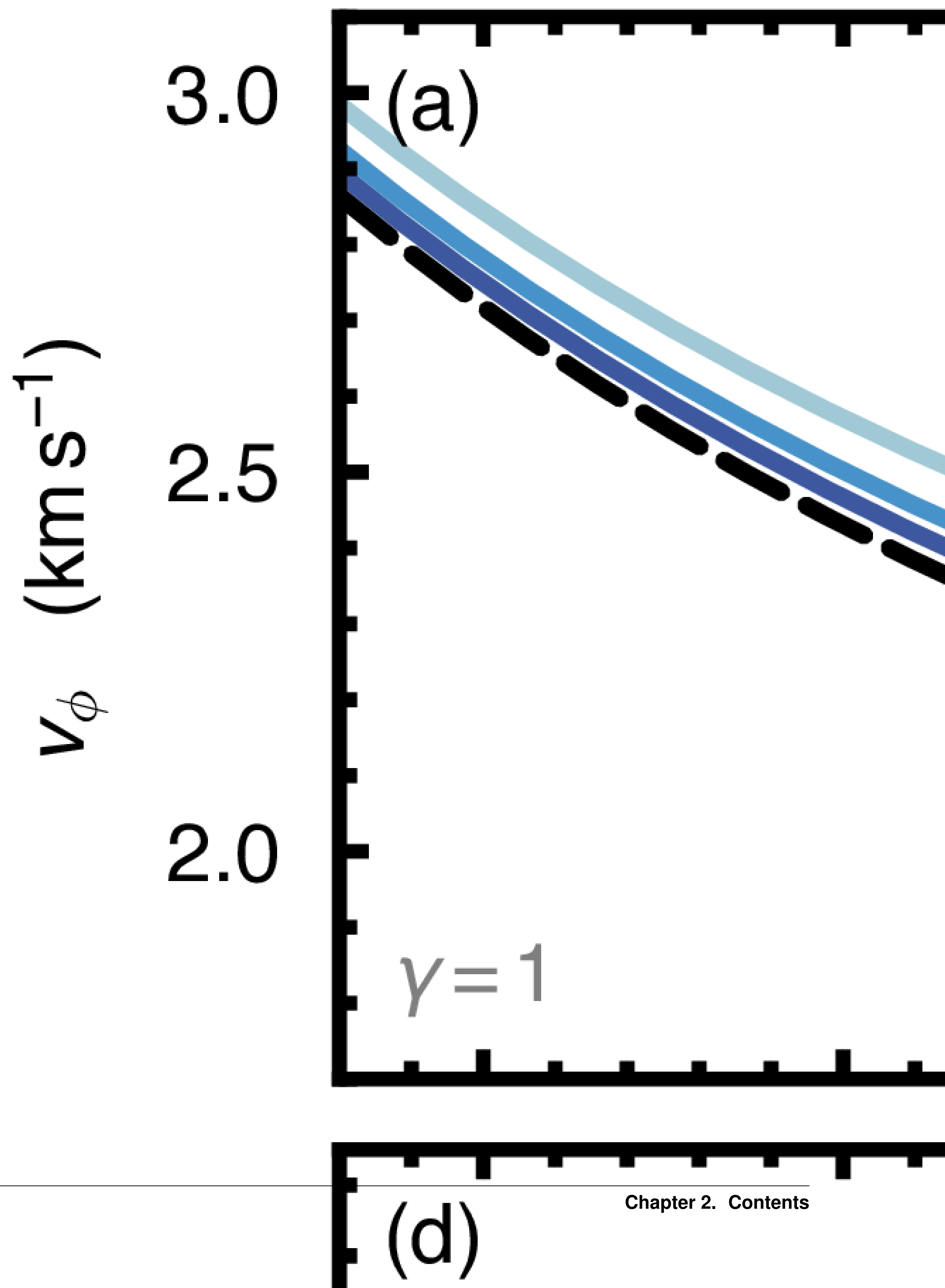
Although the parameterization described above was motivated by the need for a hastening of the disk's rotation due to the added gravitational potential of the disk, this functional form allows for a handy form to also account for the pressure support in the outer edge of the disk. This connection can most easily be seen when you consider the slowing of the disk as due to a radially dependent *negative* disk mass.

With this interpretation you can broadly relate the parameters to properties of the pressure support:

- `'mdisk'` - Sets the magnitude of sub-Keplerian rotation.
- `'r_in'` - Sets the radius where the rotation profile deviates from Keplerian rotation, likely the start of the exponential tail in the surface density profile.

- `'r_out'` - Sets the radius beyond which the rotation curve looks Keplerian again, but around a star with a reduced dynamical mass.
- `'gamma'` - Sets how quickly the velocity profile transforms between `'r_in'` and `'r_out'`.

This figure from Teague et al. (2022) shows how these parameters can be used to mimic a super-Keplerian (panels *a*, *b*, *d* and *e*) and sub-Keplerian (panels *c* and *f*) rotation curve where the black dashed line shows a purely Keplerian profile.


(a)
3.0
2.5
2.0
v_ϕ (km s^{-1})
$\gamma = 1$
(d)

2.6.2 Application to a Self Gravitating Disk

[2]:
```
# coming soon
```

2.6.3 Application to a Pressure Supported Disk

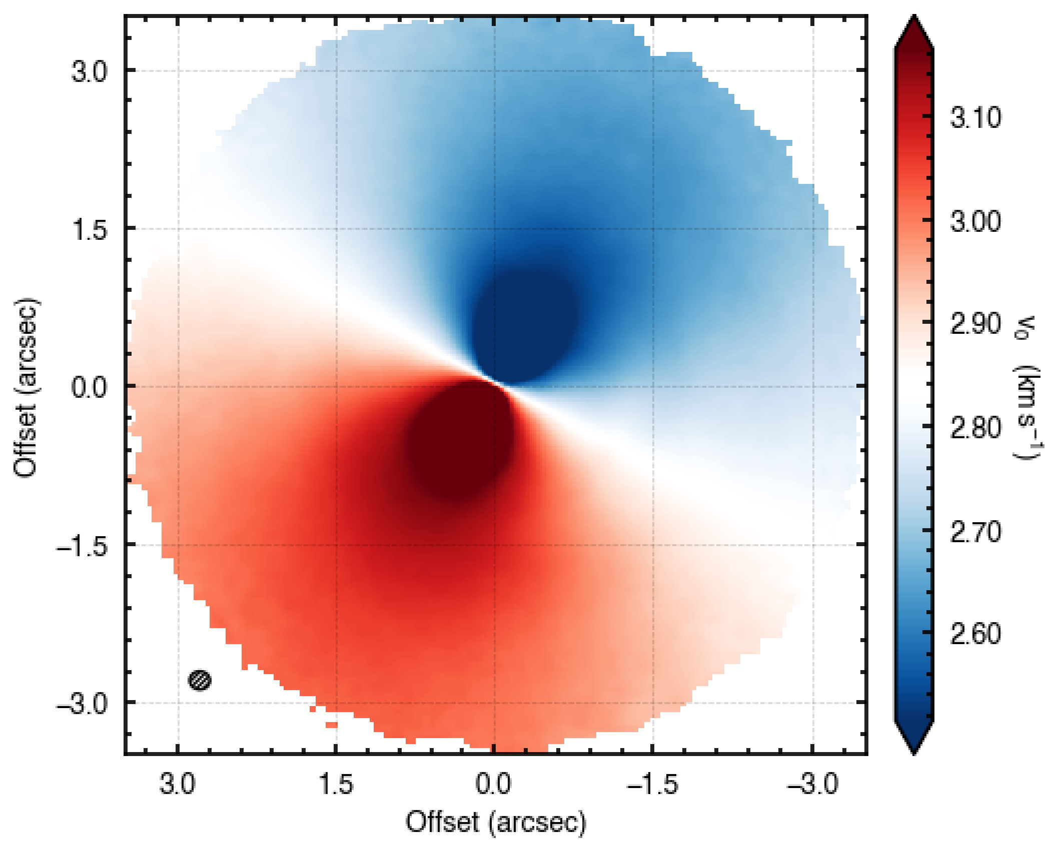
First we load up the data from Teague et al. (2022). The `v0` and `dv0` maps can be obtained from eddy Dataverse, or using the cell below.

[3]:
```
import os
if not os.path.exists('TWHya_CO_32_cube_gtv0.fits'):
    !wget -O TWHya_CO_32_cube_gtv0.fits -q https://dataverse.harvard.edu/api/access/
↪datafile/7070646
if not os.path.exists('TWHya_CO_32_cube_dgtv0.fits'):
    !wget -O TWHya_CO_32_cube_dgtv0.fits -q https://dataverse.harvard.edu/api/access/
↪datafile/7070645
```

Note here we're using the `downsample` parameter to speed things up and setting a field of view of 7 arcseconds with the `FOV` argument.

[4]:
```
cube = rotationmap('TWHya_CO_32_cube_gtv0.fits', FOV=7.0, downsample=2)
cube.plot_data()
```

```
Assuming uncertainties in TWHya_CO_32_cube_dgtv0.fits.
```

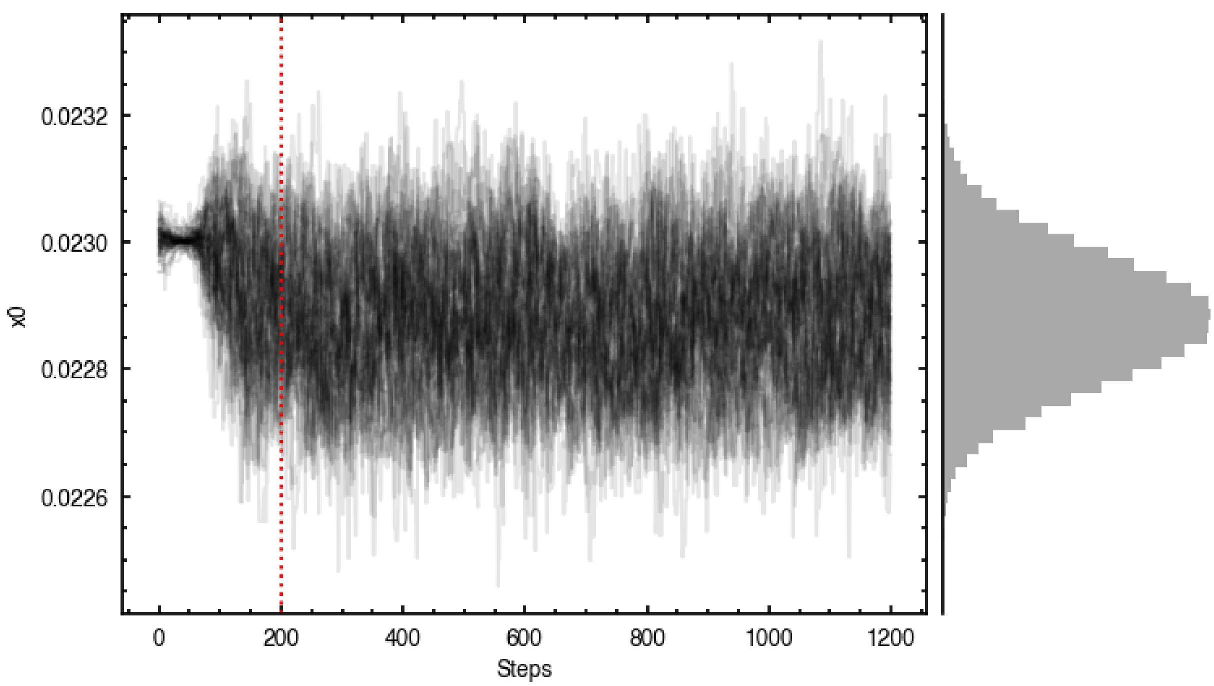


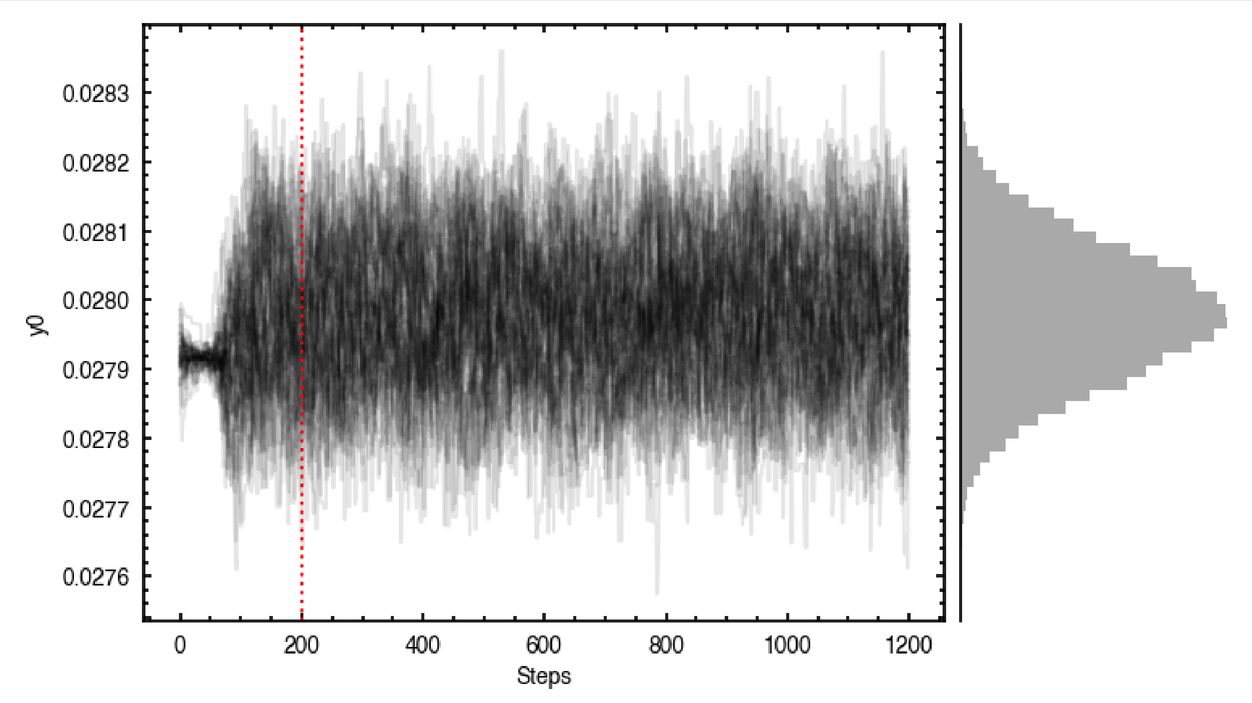
To start we try and fit a simple Keplerian model. Remember to mask out those inner regions where beam smearing becomes an issue.

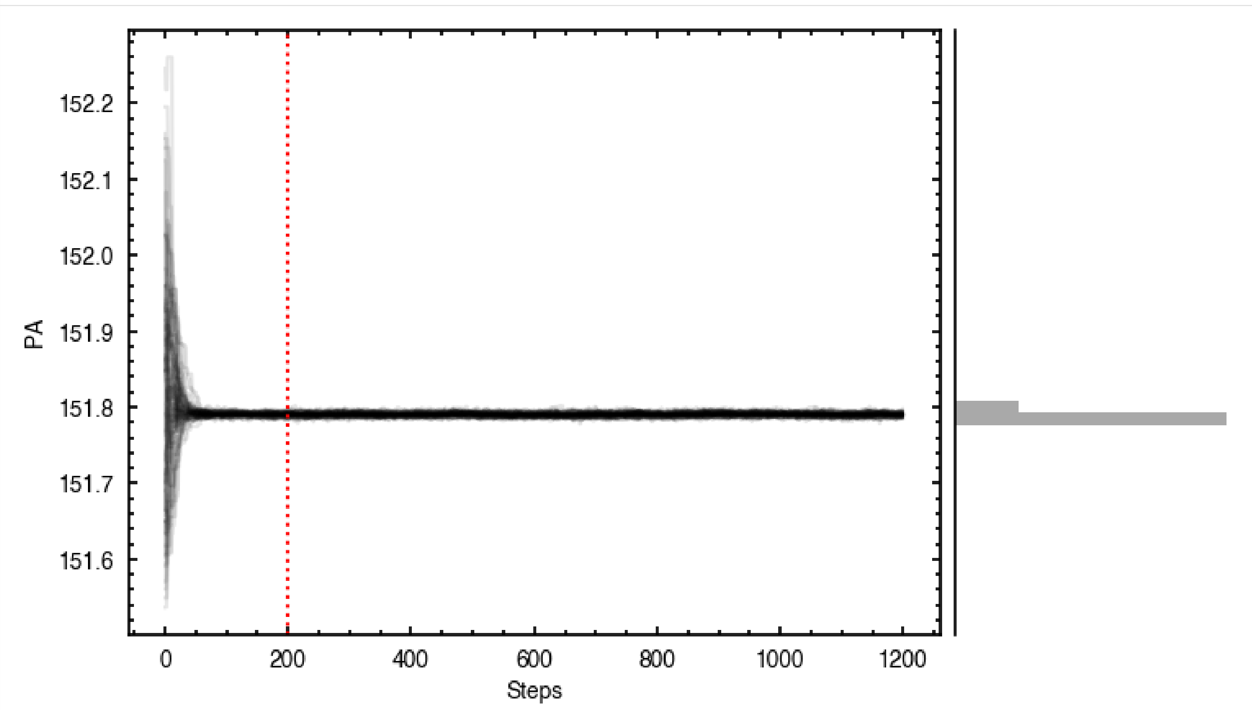
[5]:
```
# Dictionary to contain the disk parameters.

params = {}

# Start with the free variables in p0.

params['x0'] = 0
params['y0'] = 1
params['PA'] = 2
params['mstar'] = 3
params['vlsr'] = 4

# Provide starting guesses for these values.

p0 = [0.0, 0.0, 151., 0.81, 2.8e3]

# Fix the other parameters.

params['inc'] = 5.8
params['dist'] = 60.1
params['r_min'] = 2.0 * cube.bmaj

# Run the sampler.

samples = cube.fit_map(p0=p0, params=params, nwalkers=64, nburnin=200, nsteps=1000)
```

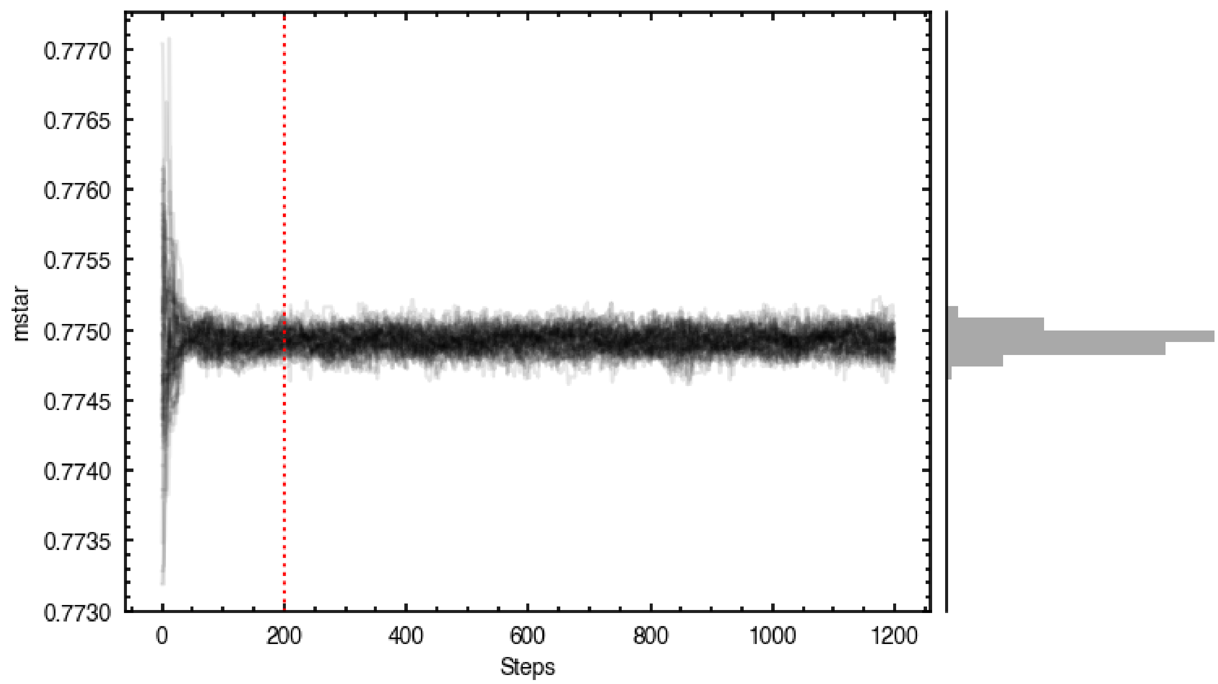
```
Assuming:
        p0 = [x0, y0, PA, mstar, vlsr].
Optimized starting positions:
        p0 = ['2.30e-02', '2.79e-02', '1.52e+02', '7.75e-01', '2.84e+03']
```

```
100%|| 1200/1200 [01:10<00:00, 17.10it/s]
```

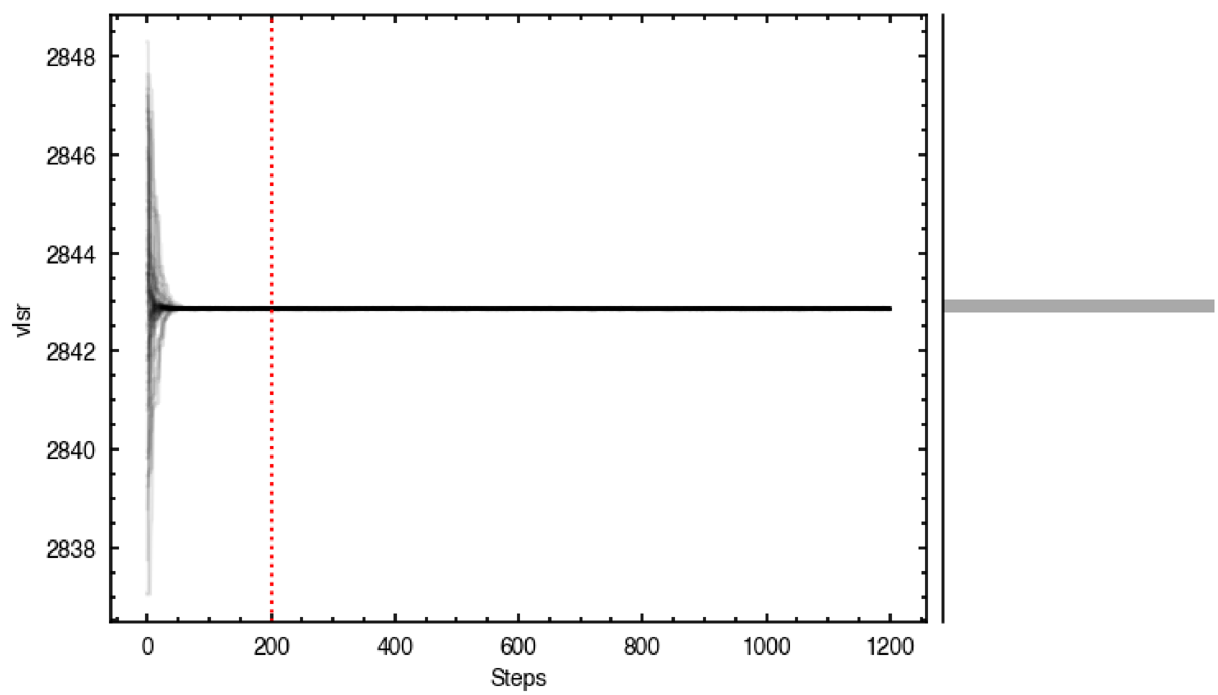



0.0283
0.0282
0.0281
0.0280
0.0279
0.0278
0.0277
0.0276
y0
0
200
400
600
800
1000
1200
Steps

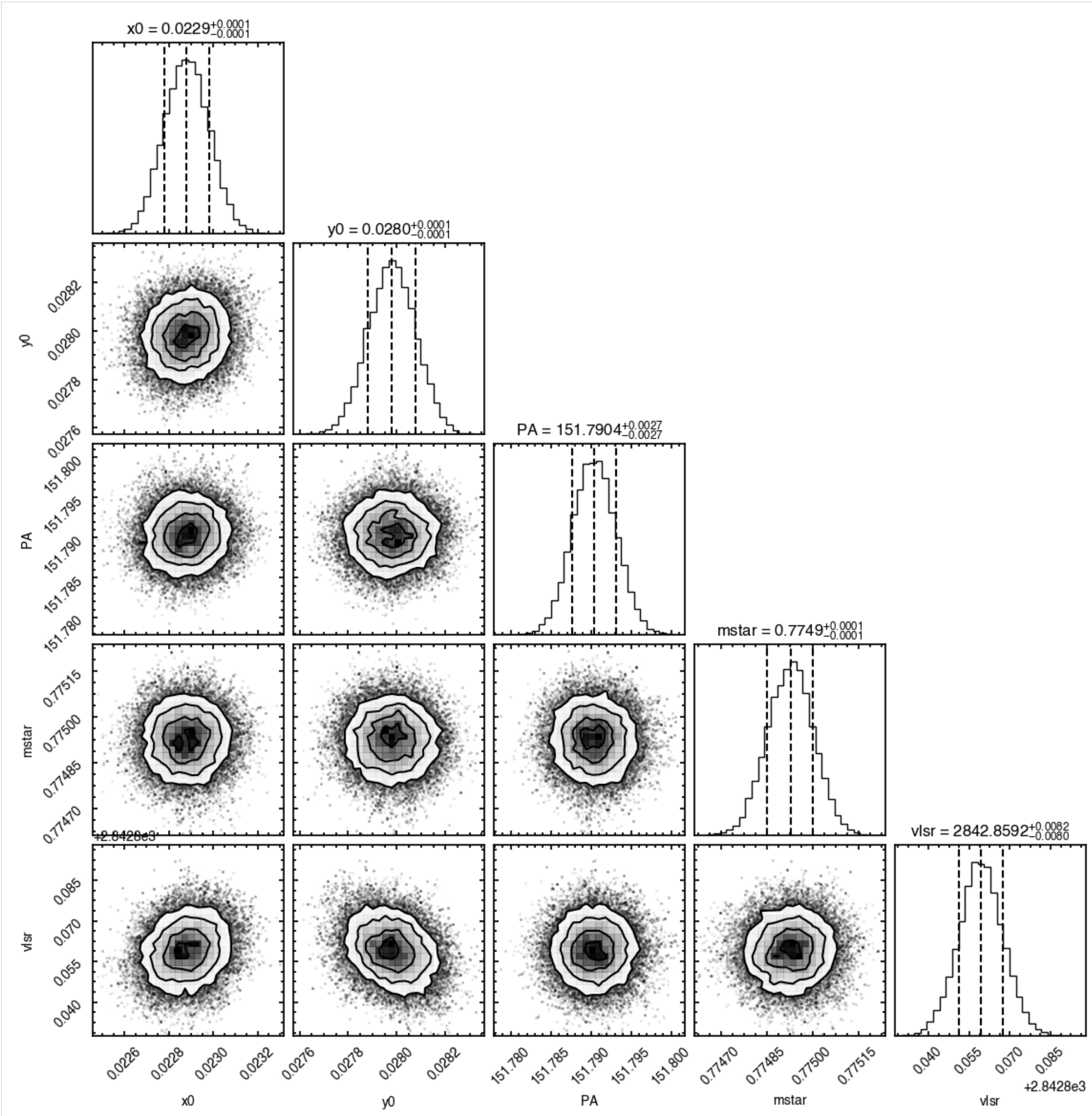

152.2
152.1
152.0
151.9
151.8
151.7
151.6
PA
0
200
400
600
800
1000
1200
Steps

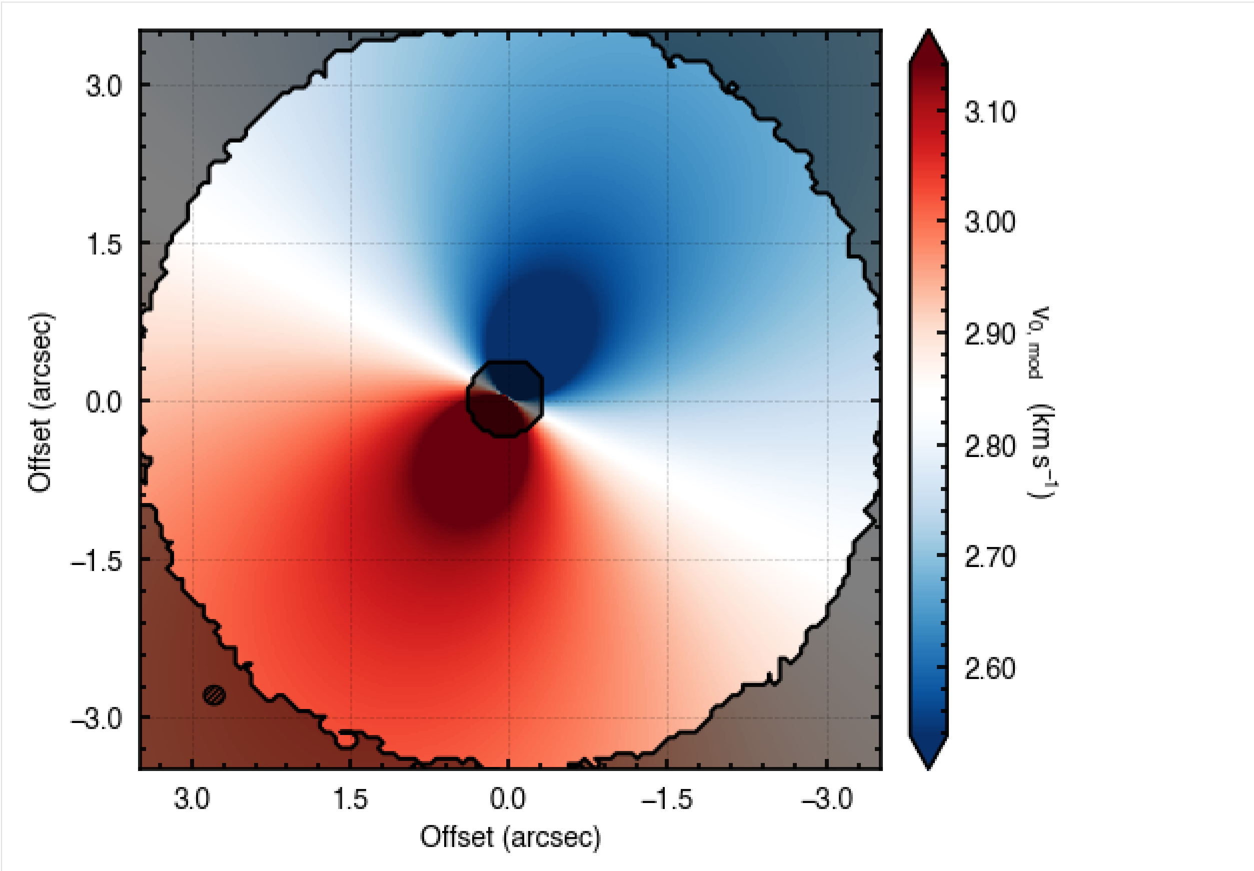


0.7770
0.7765
0.7760
0.7755
0.7750
0.7745
0.7740
0.7735
0.7730
mstar
0
200
400
600
800
1000
1200
Steps

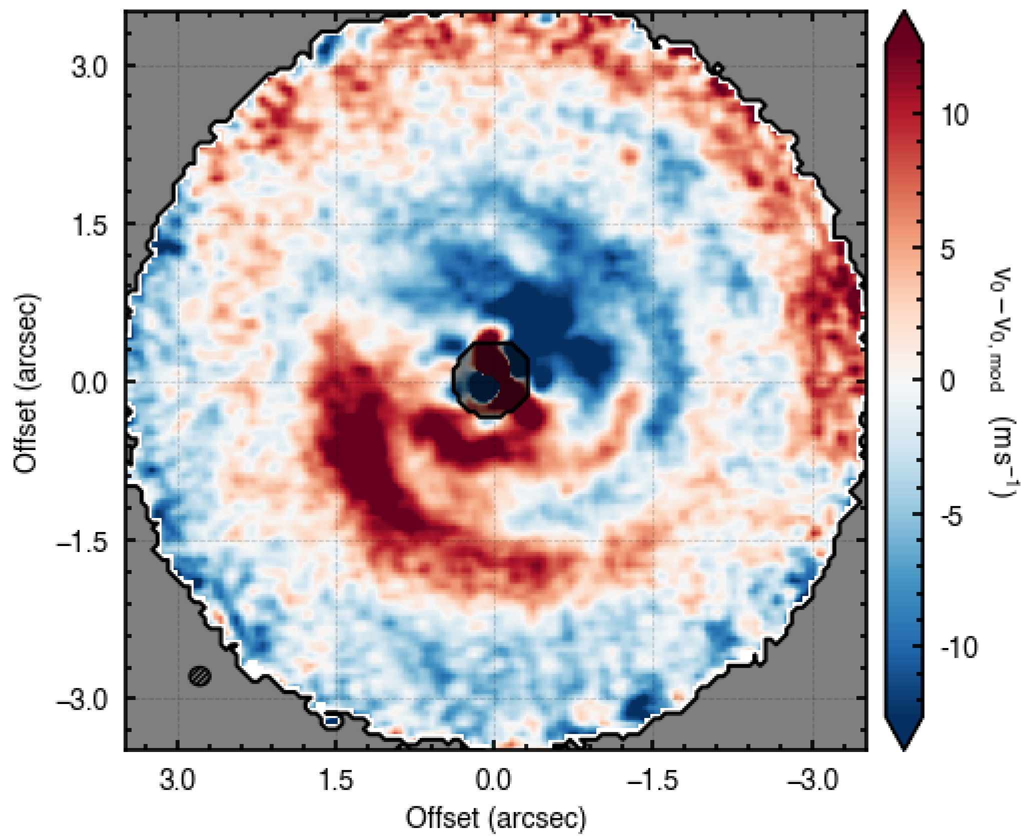


2848
2846
2844
2842
2840
2838
vlsr
0
200
400
600
800
1000
1200
Steps


x0 = $0.0229^{+0.0001}_{-0.0001}$
y0 = $0.0280^{+0.0001}_{-0.0001}$
PA = $151.7904^{+0.0027}_{-0.0027}$
mstar = $0.7749^{+0.0001}_{-0.0001}$
vlsr = $2842.8592^{+0.0082}_{-0.0080}$
y0
PA
mstar
vlsr
+2.8428e3
x0
y0
PA
mstar
vlsr
+2.8428e3



3.0
1.5
0.0
–1.5
–3.0
Offset (arcsec)
3.0
1.5
0.0
–1.5
–3.0
Offset (arcsec)
3.10
3.00
2.90
2.80
2.70
2.60
$v_{0,\,\mathrm{mod}}$ (km s^{-1})



Although we get nice convergence of the walkers with seemingly independent posterior distributions, the residuals highlight that there's clearly some residual structure. Once major component is the large spiral arm which is discussed in the aforementioned paper. However, there's a clear red / blue residual along the major axis that can be attributed to sub-Keplerian rotation (see Fig. 5 in the Appendix of Teague et al. 2019a).

To account for this structure we can use the parameterization described above. Here we set `'r_out'` to the disk outer edge, 4 arcseconds, as this should be the slowest part of the rotation.

[6]:
```
# Dictionary to contain the disk parameters.

params = {}

# Start with the free variables in p0.

params['x0'] = 0
params['y0'] = 1
params['PA'] = 2
params['mstar'] = 3
params['vlsr'] = 4
params['r_in'] = 5
params['gamma'] = 6
params['mdisk'] = 7

# Provide starting guesses for these values.

p0 = [0.0, 0.0, 151., 0.81, 2.8e3, 1.0, 1.0, 0.0]
```

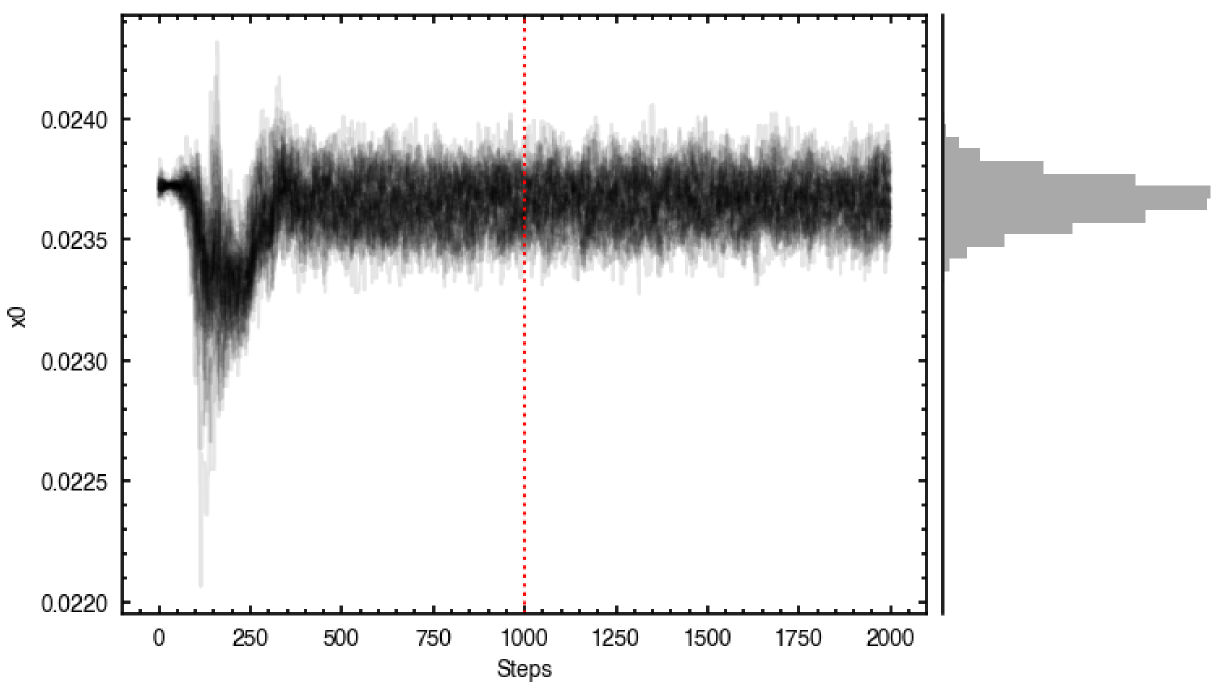
(continues on next page)

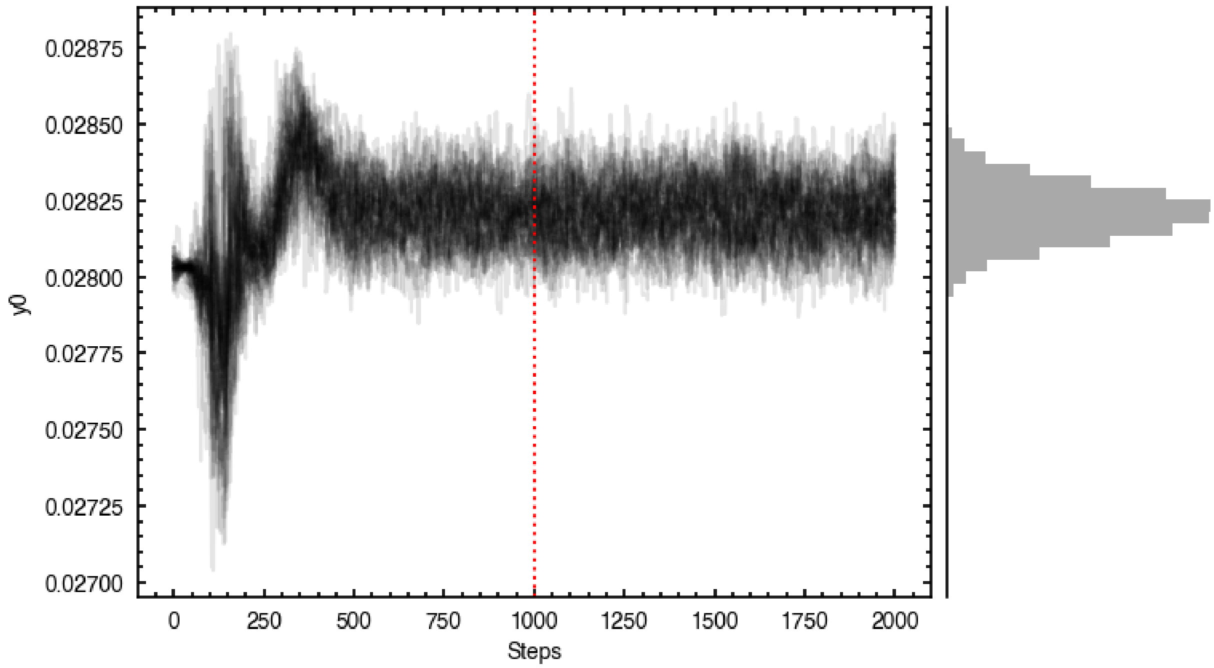
(continued from previous page)

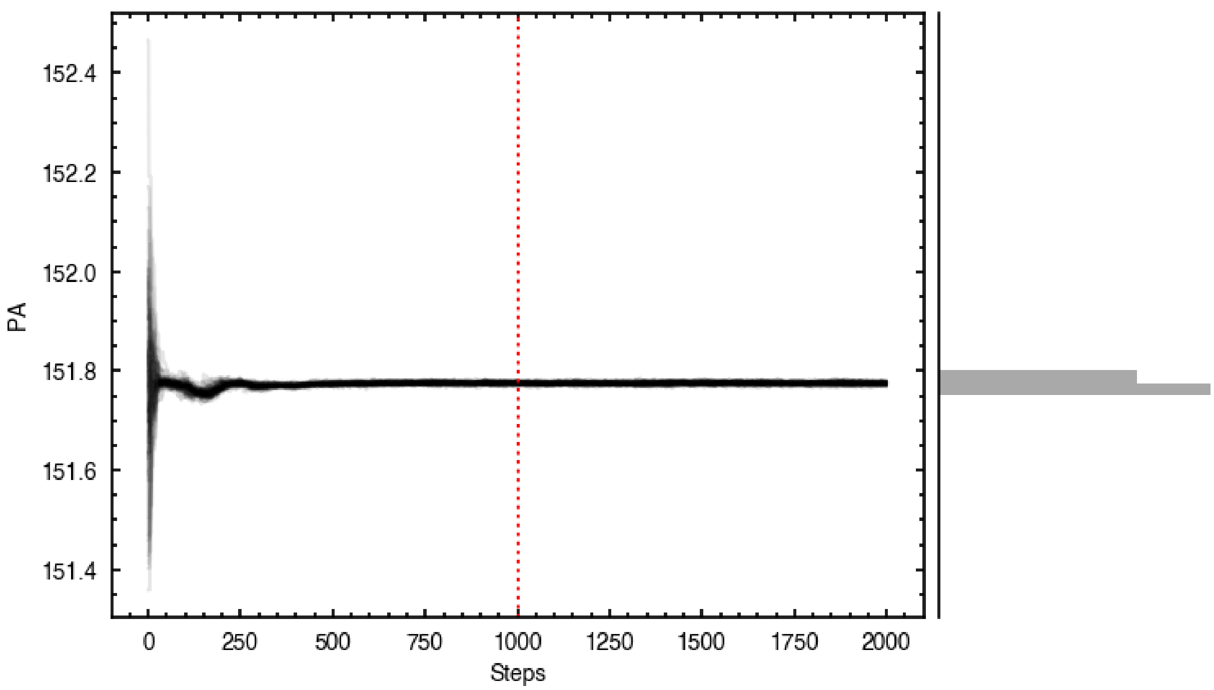
```
# Fix the other parameters.

params['inc'] = 5.8
params['dist'] = 60.1
params['r_min'] = 2.0 * cube.bmaj
params['r_out'] = 4.0

# Run the sampler.

samples = cube.fit_map(p0=p0, params=params, nwalkers=64, nburnin=1000, nsteps=1000)
```

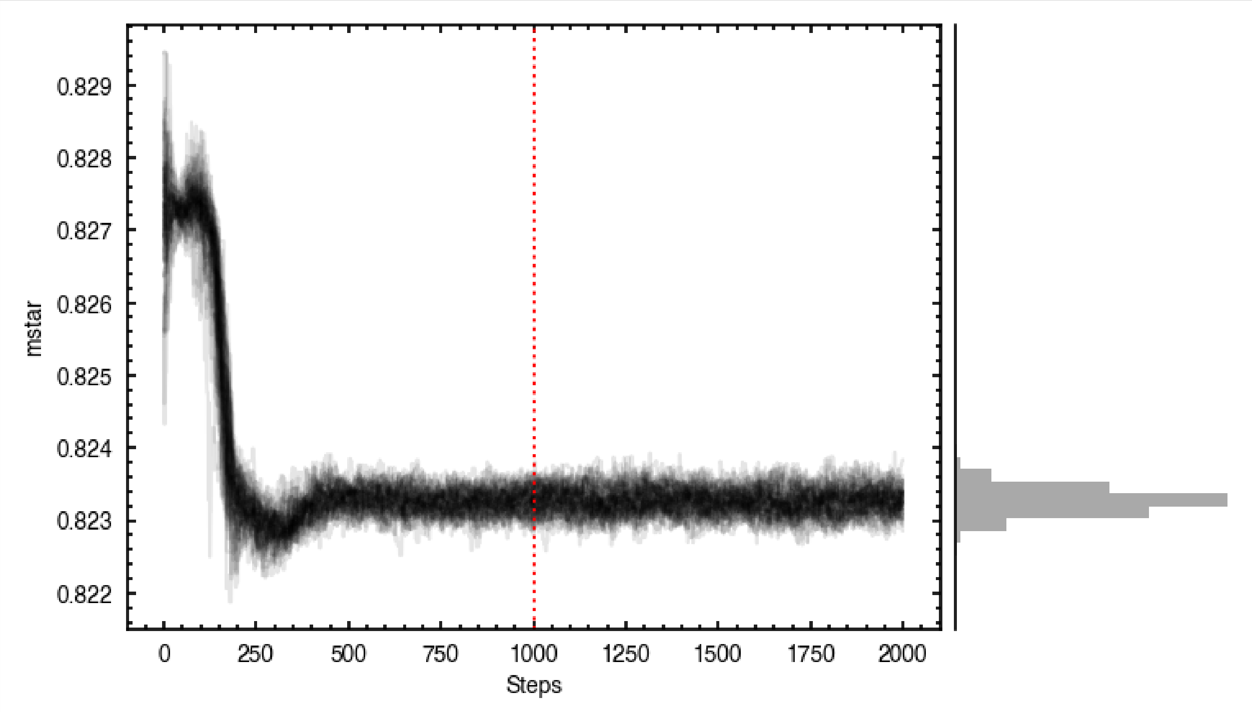
```
Assuming:
        p0 = [x0, y0, PA, mstar, vlsr, r_in, gamma, mdisk].
Optimized starting positions:
        p0 = ['2.37e-02', '2.80e-02', '1.52e+02', '8.27e-01', '2.84e+03', '1.47e+00',
↪'9.19e-01', '-1.37e-01']
```

```
100%|| 2000/2000 [02:22<00:00, 13.99it/s]
```

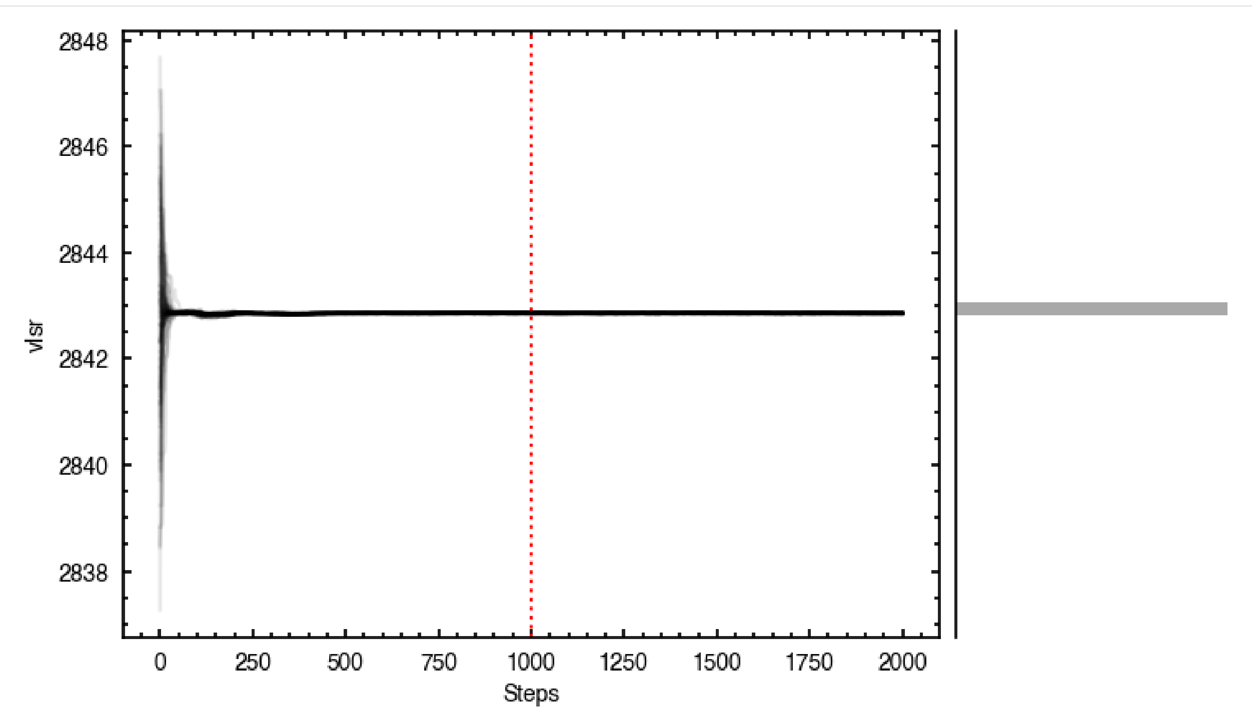



0.02875
0.02850
0.02825
0.02800
0.02775
0.02750
0.02725
0.02700
y0
0
250
500
750
1000
1250
1500
1750
2000
Steps

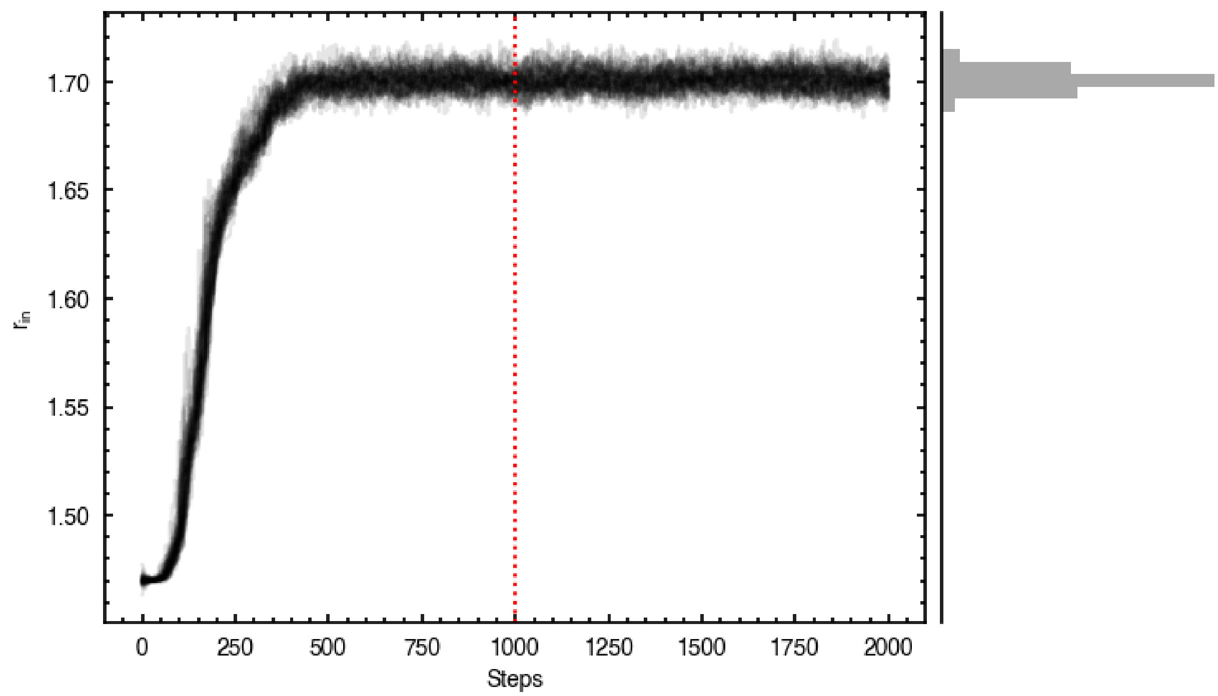

152.4
152.2
152.0
151.8
151.6
151.4
PA
0
250
500
750
1000
1250
1500
1750
2000
Steps

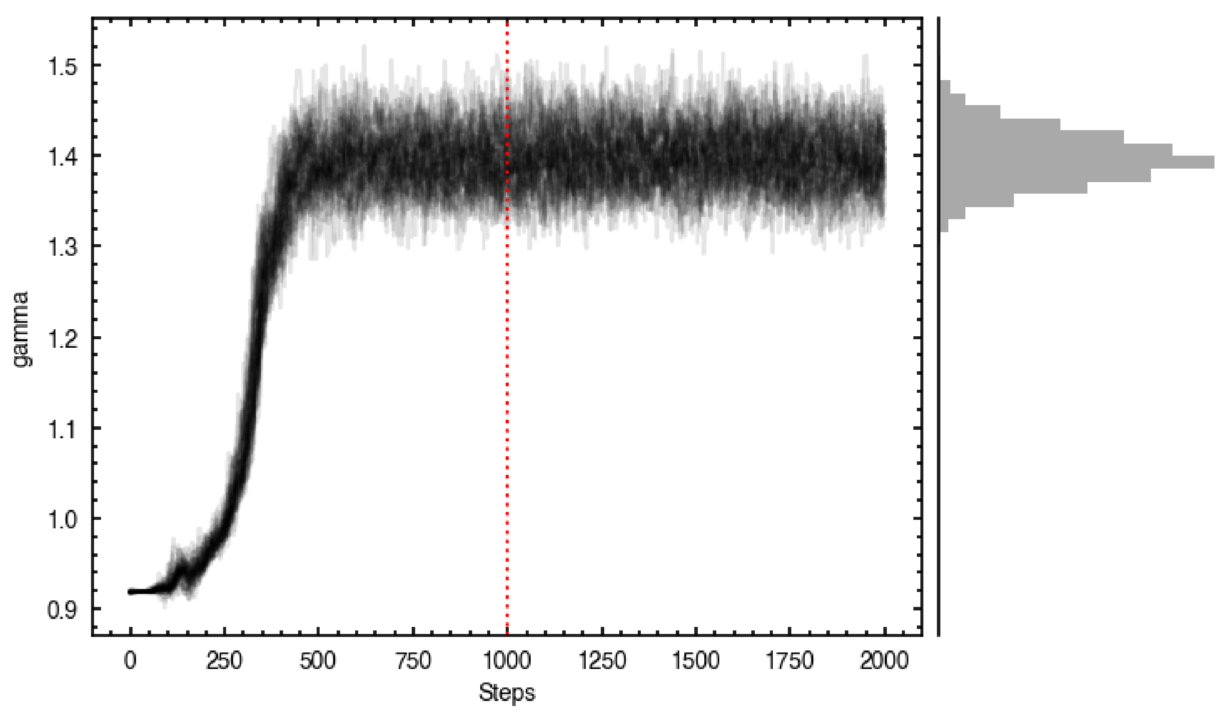


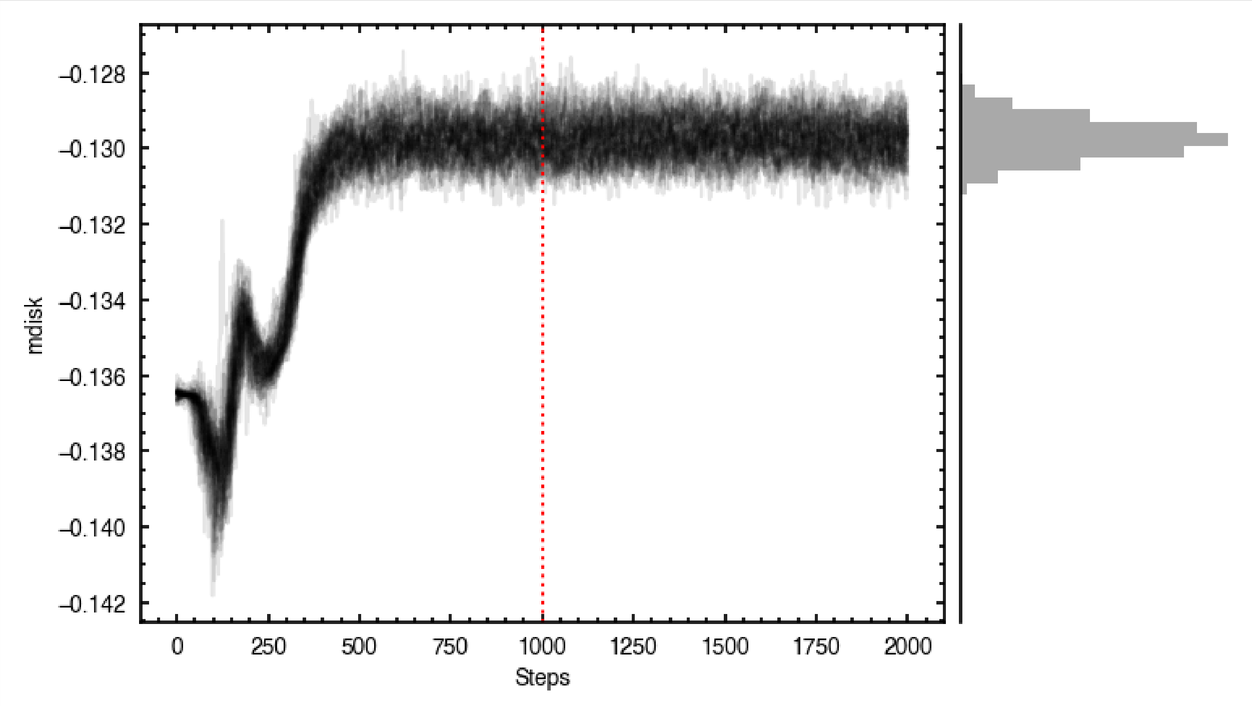
mstar
0.829
0.828
0.827
0.826
0.825
0.824
0.823
0.822
0
250
500
750
1000
1250
1500
1750
2000
Steps

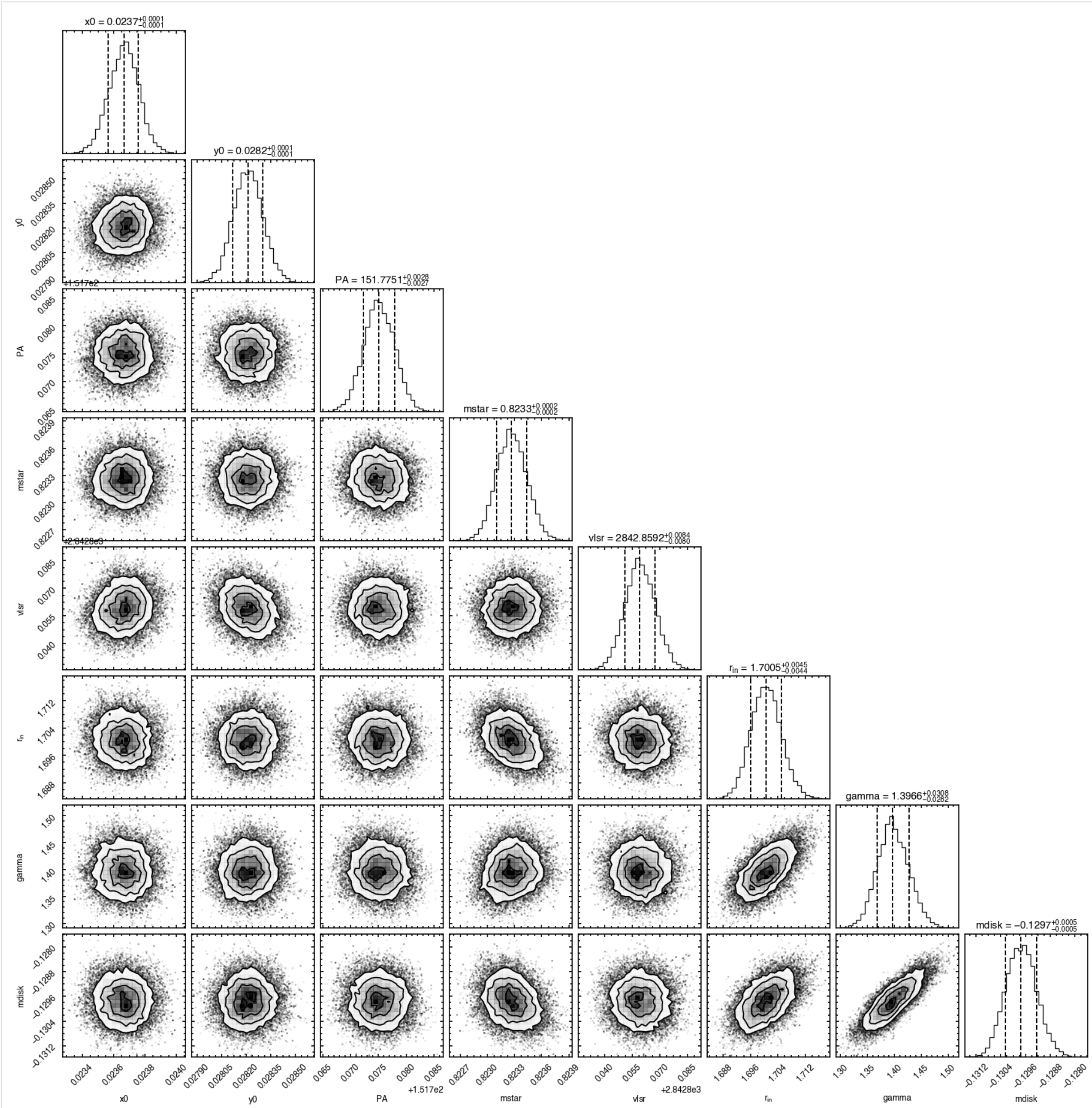


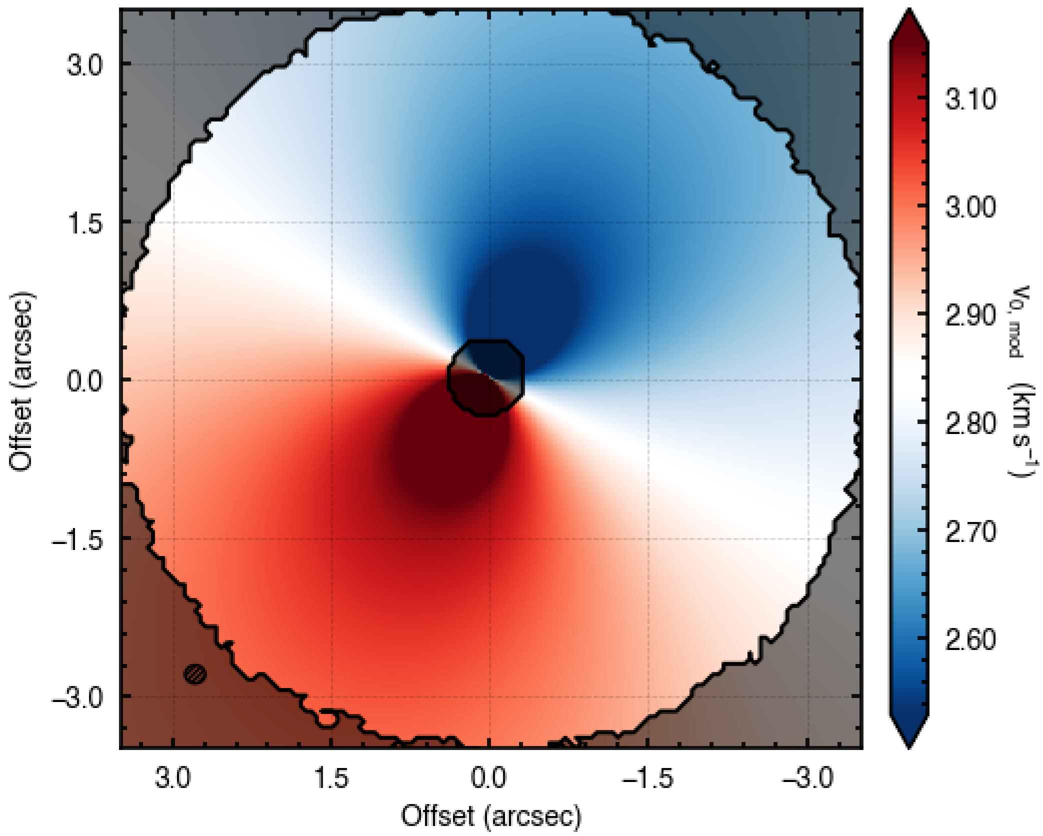
vlsr
2848
2846
2844
2842
2840
2838
0
250
500
750
1000
1250
1500
1750
2000
Steps

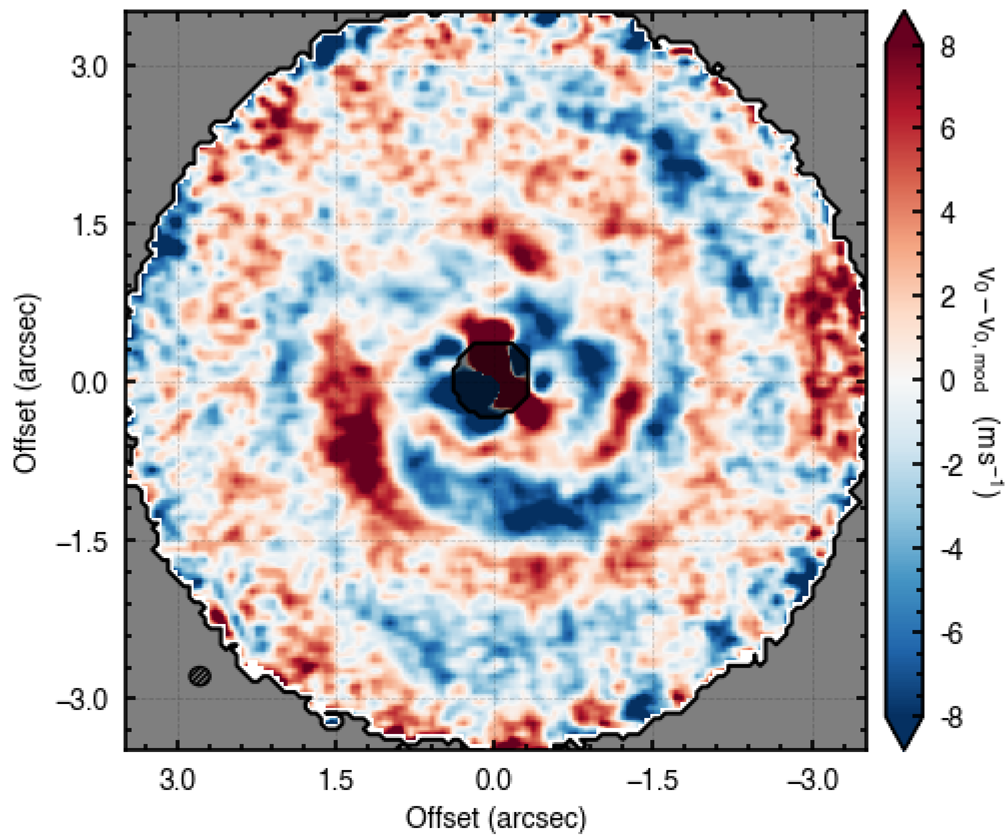

1.70
1.65
1.60
1.55
1.50
r_{in}
0
250
500
750
1000
1250
1500
1750
2000
Steps


1.5
1.4
1.3
1.2
1.1
1.0
0.9
gamma
0
250
500
750
1000
1250
1500
1750
2000
Steps


mdisk
−0.128
−0.130
−0.132
−0.134
−0.136
−0.138
−0.140
−0.142
0
250
500
750
1000
1250
1500
1750
2000
Steps


x0 = 0.0237
y0 = 0.0282
PA = 151.7751
mstar = 0.8233
vlsr = 2842.8592
r_in = 1.7005
gamma = 1.3966
mdisk = −0.1297
x0
y0
PA
mstar
vlsr
r_in
gamma
mdisk


3.0
1.5
0.0
–1.5
–3.0
Offset (arcsec)
3.0
1.5
0.0
–1.5
–3.0
Offset (arcsec)
3.10
3.00
2.90
2.80
2.70
2.60
$v_{0,\,\mathrm{mod}}$ $(\mathrm{km\,s^{-1}})$



Clearly this does a much better job of removing those outer residuals and leaving a more prominent spiral pattern.

CHAPTER 3

Support

If you are having issues, please open a issue on the GitHub page.

CHAPTER 4

Attribution

If you use *eddy* in any of your work, please cite the JOSS article,

```
@article{eddy,
    doi = {10.21105/joss.01220},
    url = {https://doi.org/10.21105/joss.01220},
    year = {2019},
    month = {feb},
    publisher = {The Open Journal},
    volume = {4},
    number = {34},
    pages = {1220},
    author = {Richard Teague},
    title = {eddy},
    journal = {The Journal of Open Source Software}
}
```

CHAPTER 5

License

The project is licensed under the MIT license.