qubefit

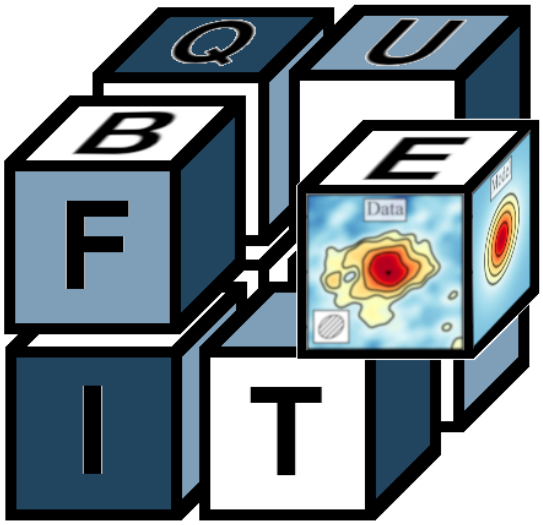
Release 1.0.2

Marcel Neeleman

Sep 11, 2023

USER MANUAL:

Qubefit is a python-based code designed to fit models to astronomical observations that yield three dimensional 'data cubes'. These 'data cubes' consist out of two spatial directions and one spectral, and are the end products of both radio / (sub-)millimeter interferometry observations, as well as observations with integral field units in the optical and near-infrared. Qubefit was specifically designed to model the emission from high redshift galaxies, for which the emission is only barely resolved and the resolution of the observations, which is known as the point spread function (PSF) or the beam, substantially affects the emission (often called beam-smearing).



Currently there are several packages available that can model the emission from high redshift **disk** galaxies. However, for some high redshift galaxies this might not be a good assumption. Some galaxies might not show any sign of rotation, while other galaxies are actually multiple merging clumps. Modeling these galaxies with disks could bias observations by erroneously fitting disks to objects that might not be disks at all. Qubefit has several non-disk like models, and other user-defined models can easily be added into the code. This can help assess if galaxy emission truly arises from disks or if other configurations can also reproduce the observed emission.

Qubefit uses a fully Bayesian framework to explore the multi-dimension parameter space and find the best fit parameters and uncertainties. It does this by applying the affine invariant Markov Chain Monte Carlo (MCMC) ensemble sampler via the emcee package. Although this fully Bayesian approach comes with a price in computational speed, it is relatively robust to the choice of priors and initial guesses.

Within the code, there are several convenience functions defined to assess the results of the fitting routine, as well as to assess the kinematics of the data, such as moment images and positon-velocity diagrams. Tutorials on how to fit a model to a data cube, as well as how to use the kinematics functions are given in the tutorial section in the menu. These are good places to start after you have installed qubefit.

CHAPTER

ONE

CITING THE CODE

If you use this code in your work, please cite the following paper: `link to paper`. Here is the bibtex reference for this work:

```
Put bibtex reference here
```

CHAPTER

TWO

LICENSE

Qubefit is free software made available under the MIT License, and comes with no guarantee whatsover. For details see the `LICENSE`.

CHAPTER

THREE

BUGS AND SUGGESTIONS

If you spot any bugs (there are plenty), have suggestions or comments, and/or wish to contribute to the code, please let me know at neeleman-at-mpia.de.

CHAPTER

FOUR

DOCUMENTATION

4.1 Installation

Qubefit has been tested on both macOS as well as Linux (Red Hat 7). To install the code clone or download the latest version from github: qubefit. After downloading the code (and unzipping if necessary), open up a terminal window, navigate to the directory where the code lives, which contains the setup.py file, and type:

```
python setup.py install
```

If everything goes correctly, this installs the code. To test if the installation was successful, see if the GUI loads. Use the command line to put yourself Inside the examples folder `qubefit/examples` and then type:

```
qubemom WolfeDiskCube.fits
```

and

```
qfgui WolfeDiskSetup.py
```

If this brings up the two GUIs such as shown in the section *Graphical User Interfaces*, the code was installed correctly.

4.1.1 Dependencies

Qubefit has been tested on python 3.8 and depends heavily on the following packages, where the version numbers are the version used for testing the code:

- `numpy (v1.19.2)`
- `scipy (v1.5.2)`
- `astropy (v4.0.2)`
- `matplotlib (v3.3.2)`
- `h5py (2.10.0)`
- `scikit-image (v0.17.2)`
- `tqdm (v4.50.2)`
- `emcee (v3.0.2)`
- `corner (v2.1.0)`

The first four packages are standard packages that you probably already have installed on your computer. For these the version number is probably not particularly important except for `numpy`, which pre v.1.15 will throw an error because of certain missing functions.

The following four packages can simply be installed using conda e.g.,:

```
conda install h5py
conda install scikit-image
conda install tqdm
conda install -c conda-forge emcee
```

It should be noted that the top three packages are normally installed in the full installation of conda (v4.9.2). Alternatively, these programs can also be installed using pip. See the repsective websites for each program: h5py, scikit image, tqdm, emcee. `scikit-image` package is only needed to generate masks, and if you want to make your own masks, it does not need to be installed. The `tqdm` package provides the progress bar used in the MCMC chain.

Warning: It is very important to use version 3.X or higher for `emcee`, because version 2.X or earlier of this code will result in errors. Further details are given in the documentation for emcee v3.0.

Although the `corner` package is not required to run the fitting routine, it is recommended for displaying the output of the fitting routine, and is necessary to create the diagnostic plots. It can be installed using pip, see the documentation for corner.

4.2 Running the Fitting Routine

This page will go through the steps needed to fit an existing model to a data cube. If you wish to define your own model you can have a look at *Making your own Model*. In short, to fit an existing model to a data cube requires two steps. The first step is to setup a *Qubefit* instance. This instance should include all of the information needed to run the fitting procedure, such as the data cube, the model, and priors of the parameters of the model. The second step is running the Markov Chain Monte Carlo (MCMC) routine. This step is computationally expensive, but requires little input from the user.

4.2.1 Initialize the qubefit instance

The first step is to initialize a *Qubefit* instance and populate all of the information within this instance to successfully run the fitting routine. In theory this can be done line-by-line interactively, but in practice it is much easier to create a setup file to do this (see the *Example Setup File*). The code requires the following keys to be defined (with a short description). This list is here provided for reference, and it is strongly advised to use a setup file, and the associated helper functions, to populate these keys as shown in the *Example Setup File*.

- **data**: the data cube from the observations
- **shape**: the shape/dimensions of the data cube
- **variance**: the variance of the data cube with the same dimensions as the data
- **maskarray**: The mask used in the fitting procedure, which is a Boolean array with the same size as the data.
- **kernel**: the point spread function or beam of the observations convolved with the line spread function of the instrument. Should be a three dimensional cube smaller than the data.
- **kernelarea**: Float that contains the size of the point spread function or beam (used for normalization).
- **probmethod**: String describing the method used to calculate the probability and likelihood function.
- **modelname**: The model to fit. String name should be a function defined in qfmodels.py.
- **intensityprofile**: List of three strings describing the intensity profile in the radial, azimuthal and axial direction. Profiles should be defined, within underscores, in qfmodels.py

- **velocityprofile**: List of three strings describing the velocity profile in the radial, azimuthal and axial direction. Profiles should be defined, within underscores, in qfmodels.py
- **dispersionprofile**: List of three strings describing the dispersion profile in the radial, azimuthal and axial direction. Profiles should be defined, within underscores, in qfmodels.py
- **par**: Array with the numeric values needed for the model. These values are in intrinsic units.
- **mcmcpar**: Subset of the **par** key for the parameters that are part of the MCMC routine, i.e., those parameters that are note held fixed.
- **mcmcmap**: Map that denotes what position in the MCMC chain corresponds to what parameter.
- **initpar**: dictionary that contains the initial parameter values in the given units, conversion to internal units, flag determining of the parameters should be kept fixed and the chosen prior (see the *Example Setup File*).

4.2.2 Running the MCMC procedure

After the *Qubefit* instance has been properly initialized with all of the required information, the fitting routine can be run. This is done with a single function call:

```
sampler = QubeS.run_mcmc(nwalkers=100, nruns=1000, nproc=6)
```

The *run_mcmc* function calls `emcee` and has three keywords that can be set. The first two are the number of walkers (**nwalkers**) and the number of steps (**nsteps**). These numbers should be large enough such that the chains are converged. More details on this are given in the emcee documentation on convergence. The last keyword sets the number of processes that will be opened by the code. Emcee allows for parallelization, **nproc** sets the numer of parallel processes that will be openend. This number should be smaller than the number of available computation cores on your system (preferably one or two less to allow for other system functions, or much less if you share the system with other users).

Although this single function can be easily called by itself, it might be useful to embed this into a small helper function that initiates the model, runs the chains, and saves the model. By default the chains will be saved to a numpy array and stored as part of the QubeFit instance, but one can store the *emcee* sampler information into other formats as well depending on your preferences.

4.3 Example Setup File

To successfully run the fitting routine, you have to initialize a *Qubefit* instance and populate all of the information within this instance. In theory this can be done line-by-line interactively, but in practice it is much easier to create a setup file to do this. The setup file we use is the WolfeDiskSetup.py. Here we will go line-by-line to explain each of the lines in this setup file. It is important to note that some of these lines of code might not be needed for your data or model, or you might need to add some additional information for the code to run successful. However, this example file is probably a good starting point for the setup file for your project.

4.3.1 Structure

```
import numpy as np
import astropy.units as u
from qubefit.qubefit import QubeFit

def set_model():
    ...
return QubeS
```

The first thing to note is the overall structure of the setup file. The setup is done within a function named *set_model()* . This structure and naming convention is important **if** you wish to use the *Graphical User Interfaces* to look at the source, because the GUIs will explicitly look for this function name within the setup file. The function returns *QubeS*, which is a *Qubefit* instance that contains all of the needed information to run the fitting routine.

4.3.2 Loading the Data

```
DataFile = './examples/WolfeDiskCube.fits'
Qube = QubeFit.from_fits(DataFile)
Qube.file = DataFile
```

In these lines the *data* and *header* keys are set using the task *from_fits*. The final line sets the *file* key in the *Qubefit* instance. This line is only used to assign a title to the GUI window.

Note: Depending on the header in the fits file, the rest frequency of the line might need to be (re)set here, i.e., Qube.header['restfrq'] = freq_in_Hz, where freq_in_Hz is the value of the redshifted frequency of the line.

4.3.3 Trimming the Data Cube

```
center, sz, chan = [507, 513], [45, 45], [15, 40]
xindex = (center[0] - sz[0], center[0] + sz[0] + 1)
yindex = (center[1] - sz[1], center[1] + sz[1] + 1)
zindex = (chan[0], chan[1])
QubeS = Qube.get_slice(xindex=xindex, yindex=yindex, zindex=zindex)
```

It is crucial to trim the data cube to the smallest possible region, because the largest time sink is convolving the model with the beam/PSF. The convolution time scales roughly linearly with the size of the cube, so large data cubes will unnecessarily slow down the code. In this example, the new trimmed cube will be centered at pixel position (507, 513), have a physical size of 90 by 90 pixels (45 pixels on each side of the center), and covers the wavelength (frequency) channels between 15 and 39 inclusive.

Note that the *get_slice* routine copies (technically it is a deep copy) the structure of the *Qubefit* instance and anything that has been previously added, so QubeS is a full instance of *Qubefit*. This routine will also update the *header* and *shape* keys to reflect the new size of the data.

4.3.4 Calculating the Variance

```
QSig = Qube.calculate_sigma()
QSig = np.tile(QSig[chan[0]: chan[1], np.newaxis, np.newaxis],
               (1, 2 * sz[1] + 1, 2 * sz[0] + 1))
QubeS.variance = np.square(QSig)
```

These lines will load the *variance* key into the *Qubefit* instance. For interferometry data, such as ALMA, this can be calculated from the data cube by fitting a Gaussian to data that contains no signal. For optical observations, the uncertainties or variances per pixel are often stored as a separate fits file, which can be loaded in directly with the *from_fits* routine. The *variance* key requires a simple numpy array with the same dimensions as the *data* key. In our case the output of the routine *calculate_sigma* is a single uncertainty per channel (wavelength slice), so we have to tile the data into a full array with the same size as the trimmed data cube. Finally we take the square (to get the variance) and load this array into the *variance* key.

Note that we need to use the full data cube to calculate the uncertainties, not the trimmed one. This is because the trimmed data cube does not have enough pixels without signal to accurately estimate the uncertainty of each channel.

4.3.5 Defining the Kernel

```
QubeS.create_gaussiankernel(channels=[0], LSFSigma=0.1)
```

This line of code populates the *kernel* key of the *Qubefit* instance. The kernel is the shape of the beam (or point spread function, PSF). For most cases, the beam or PSF can be approximated by a Gaussian, and the above code will generate such a kernel from the beam parameters defined by the fits header.

Note: For optical data the "beam" parameters will need to be updated to the values corresponding to the spatial resolution or seeing of the observations. This can be done by setting the corresponding values in the header: QubeS.header['BMAJ'] = psf_in_degrees and QubeS.header['BMIN'] = psf_in_degrees, where psf_in_degrees is the size of the point spread function (i.e., the seeing) in degrees.

After creating a 2D kernel, the kernel is convolved with the line spread function of the instrument. For ALMA observations, the line spread function is often negligible, because the channels are Hanning smoothed to much coarser resolution. Setting the width of the line spread function to something small, like 0.1 times the channel width, will make a correct 3D kernel.

4.3.6 Setting the Mask

```
QubeS.create_maskarray(sigma=3, fmaskgrow=0.01)
```

This code populates the *mask* key in the QubeS object. This mask is a simple array of ones and zeros that has the same size as the data cube. A one means to include a pixel in the fitting procedure, whereas a zero means to not include the pixel. The mask array is stored in QubeS.mask, so if you want to use your own custom mask, you would need to set this keyword simply as: QubeS.mask = your_mask_array, where your_mask_array is an array of equal size as the data cube and filled in with ones and zeros.

4.3.7 Defining the Model

```
QubeS.modelname = 'ThinDisk'
QubeS.intensityprofile[0] = 'Exponential'
QubeS.velocityprofile[0] = 'Constant'
QubeS.dispersionprofile[0] = 'Constant'
```

Here we are defining which model to use. Note that the models that come with qubefit package are described on the page *Pre-Defined Models* . You could also think about *Making your own Model*, which is part of the strength of the qubefit package.

In this example we will use the ThinDisk model. Within the ThinDisk model, we can also set several profiles for the intensity, velocity and the dispersion. In this case we assume a simple exponential profile for the emission and both a constant velocity and dispersion profile across the disk. Options available are described in detail in *Pre-Defined Models*.

4.3.8 Parameters and Priors

```
PDict = {'Xcen': {'Value': 45.0, 'Unit': u.pix, 'Fixed': False,
                  'Conversion': None,
                  'Dist': 'uniform', 'Dloc': 35, 'Dscale': 20},
                  ..}
QubeS.load_initialparameters(PDict)
```

The next thing to load into the *Qubefit* instance are the parameters for the model. The parameters are stored in a nested dictionary. For each parameter in the dictionary, 7 keys need to be defined that will determine the initial value of the parameter and its prior. The function *load_initialparameters* populates several keys in the *Qubefit* instance, namely *initpar*, *par*, *mcmcpar*, *mcmcmap* and *mcmcdim*. Although these could be set individually, it is **highly** recommended to use the load_initialparameters keyword, to make sure the mapping gets done correctly. The structure of the dictionary for each parameter is as follows:

- **Value**: The initial value of the parameter in whatever unit you specify.
- **Unit**: The unit of the parameter. Any unit can be used as long as you apply the correct conversion to the native units of the cube with the **Conversion** key.
- **Fixed**: if sets to True, the code will keep this parameter fixed during the fitting routine.
- **Conversion**: This will convert the **Value** parameter into the native units of the data cube. For instance, the velocities are often wanted in units of km/s, but the native units of the cube are pixels (in the spectral direction). Note that this conversion can also be used to convert degrees into radians (for angles).
- **Dist**: This is the prior distribution for the parameter. The valid distributions here are those defined in the scipy.stats module. A large list of valid priors are allowed, but beware that not all will give reasonable results. One should be very careful selecting these distributions, and when in doubt take the least constraining, which is often an uniformed (here called *uniform*) prior.
- **Dloc**: This sets the location of the distribution and is equal to the *loc* parameter in the functions defined in scipy.stats. Look at these pages to see what this parameter means for the distribution that you have chosen. For example, in the *uniformed* prior this corresponds to the lower bound of the acceptable range.
- **Dscale**: This sets the scale of the distribution and is equal to the *scale* parameter in the functions defined in scipy.stats. Look at these pages to see what this parameter means for the distribution that you have chosen. For example, in the *uniformed* prior this corresponds to the range of the acceptable values starting at **Dloc**. Therefore in our example, all *Xcen* values between 35 and 55 pixels have equal probability, and outside this range the probability is zero.

4.3.9 Making the Model

```
QubeS.create_model()
```

We are now finally in a postion to create a model cube. This is done with the above command. The model will be stored *model* key, and it will have the same dimension as the data cube. The model cube has also been convolved with the kernel (Beam/PSF). If you want to make a model that is not convolved with the kernel, you can set the keyword convolve to False, i.e., *QubeS.create_model(convolve=False)*.

4.4 Graphical User Interfaces

Qubefit comes with two graphical user interfaces (GUIs). The *qfgui* and the *qubemom* GUI. The first GUI, *qfgui*, can be used to help determine the correct mask and initial parameter guesses before running the fitting procedure. This step is crucial to make sure that the code converges relatively quickly and that the correct data points are used during the fitting routine. The second GUI, *qubemom*, allows you to make different moment images of the data, as well as show the data cube using a simple GUI.

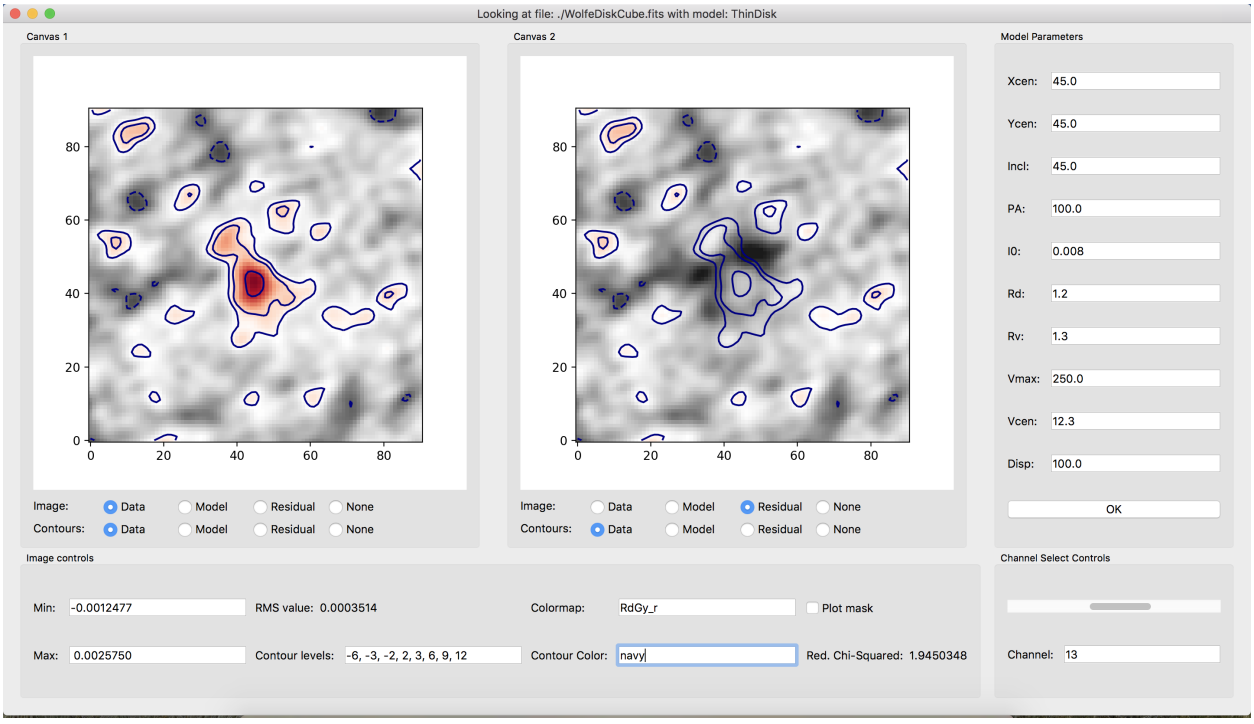
4.4.1 The Qubefit GUI (qfgui)

The aim of *qfgui* is to provide a graphical user interface to be able to quickly select decent initial parameters for the fitting procedure. Starting too far away from the best-fit solution would result in a chain not converging very fast, and would unnecessarily lengthen the time for the fitting routine. The GUI can also be used **after** the fitting routine has run, to see how well the best-fit solution reproduces the data. To start the GUI simply type:

```
qfgui setupfile.py (arg2)
```

Here *setupfile.py* is the setup file created for the data cube and the chosen model (see *Example Setup File*). This file should define the model in a function of the name *set_model()*. This function should only return the full instantiated *Qubefit* instance. The code in addition can take an optional second argument. This string gets directly passed to the *set_model()* function as an argument. This can be useful for customizing a single setup file for multiple objects.

Below is a screenshot of the *qfgui*.



The window contains five different panels. Here is a description of each of the panels;

- There are two *canvas* panels, which can show different parts of the *Qubefit* instance, either the data, the model, the residual (data minus the model), or nothing. You can overplot contours of these data as well using the radio buttons at the bottom of the panels.
- The *image controls* panel deals with the image properties of the canvases. The minimum and maximum value of the color map (default is set at -3 and 11 times the RMS of the first channel). The RMS value, which is the median

of the square root of the variance of each channel. Contour levels to draw. This should be a comma-separated list of increasing numbers. The next two inputs determine the colormap of the image, and the color of the contours. These should be valid `matplotlib` colormap and color names. There is also a check box that toggles the mask used for the fitting procedure. This is very important to check, to make sure that all the emission is captured by the mask, as well as a bit more around the edges to show where the emission drops below the detection threshold. Finally, the last line shows the reduced chi-square statistic for the data, the model and the mask.

- The *channel select* panel is a simple panel that has a slider and textbox, so you can select the wanted channel or quickly scan through the channels.
- The *model parameters* panel is populated with all of the parameters that are defined in the model. The initial values are those that are defined in the *setupfile.py* file and are given in the physical units that are defined in this file. You can change each of these values by simply clicking in the box and typing a new value. Note that the code does not check that the values are within the prior ranges set by the code, this is up to you to adhere to. After changing the parameters to your liking, hit the *OK* button and the new model will be generated. Depending on the model and data, this could take a bit (for reasonable models and data cube sizes, normally less than five seconds). Note that hitting the return (or enter) key in each of the text boxes does *NOT* reload the model, only the *OK* button does this.

After you have found an acceptable initial guess, you should copy over the values into the *setupfile.py* file before you close the GUI. If you do this, then next time you load the GUI it should show these values as the new default. If you have run the code and found the best-fit solution, then you can also use this GUI to see how successful the fit was. Simply copy over the best-fit solutions for each parameter into the *model parameters* section, and hit the `OK` button.

4.4.2 The Qube Moment GUI (qubemom)

The aim of *qubemom* is to provide a quick view of a data cube, and perform some simple kinematic operations such as determining the velocity field and dispersion field. Two methods can be used to calculate these fields, either the first and second moments of the data cube in the spectral direction or through Gaussian fitting of the spectrum at each pixel (see appendix C of the `reference paper`). For the first method a mask is often defined, in order to only count those pixels detected above a certain noise threshold. For the second method, beware that Gaussian fitting is slow, and you likely will have to wait a few minutes.

The *qubemom* GUI is currently under active development, and will continue to change in the near-future.

4.5 Pre-Defined Models

Here is a list and description of the current models that have been tested, and are supplied with the code. If you wish to create your own custom model, please see the page on *Making your own Model*.

Note: To create the models, we need to switch between the frame of the galaxy (used to create the models) and the frame of the sky (the observed frame). To distinguish between these two frames, here and elsewhere, we define the unprimed frame (e.g., r, and z) as the frame of the galaxy and the primed frame (e.g., r', ', z') as the frame of the sky. That is in the cylindrical coordinate system, r and are within the plane of the disk, and z is perpendicular to the disk, whereas the primed coordinates r' and ' are within the plane of the sky and z' is pointing towards us.

4.5.1 Thin Disk Model

This model describes the kinematic signature of emission lines that stem from an infinitely thin rotating disk, where all of velocities arise solely from rotation. The thin disk model is described in detail in Neeleman et al. 2021 `link this`. Here we provide a more general approach to the discussion in this paper. First, to create a thin disk model requires a mathematical description of the spatial distribution of the intensity (I), rotational velocity ($v_{\rm rot}$) and line-of-sight velocity dispersion ($_{\rm v}$). In its most general form, each of these quantities could be varying as a function of radius (r), and azimuthal angle (), since the emission is constrained within the plane z=0. Therefore we have the following equations:

$$I = I(r, \phi)$$
$$v_{\rm rot} = v_{\rm rot}(r, \phi)$$
$$\sigma_v = \sigma_v(r, \phi)$$

Here the functional forms that can be chosen for the profiles are defined in the *Qfmodels* function. There is a list of basic functions available, such as a constant, exponential, linear or Sérsic profile, but it is straightforward to add additional profiles. The code assumes that the profiles are separable; meaning that I(r,) = I(r) x I(), and in almost all cases you probably only care for profiles that only vary with radius (i.e., I() is constant). However it is relatively straightforward to relax this assumption in a new custom model (see *Making your own Model*).

Most profiles are determined by a single scale factor (either $R_{\rm d}$ or $R_{\rm v}$ for radial profiles), except for the power and Sérsic profiles, which require an additional parameter. The profiles are scaled using a scaling factor, which is either I_0, $V_{\rm max}$, or $_{\rm v}$ depending on which profile.

These profiles need to be transformed into the frame of the sky (the primed frame). In theory, we could do this through matrix rotations. However in practice this produces incorrect results, because the disk is infinitely thin and most of the emission will fall between the grid after rotation, and correctly distributing the flux onto the regular grid is difficult. Fortunately, the infinitely thin disk has easy-to-calculate transformations. Following the work in Chen et al. 2005:

$$r = r' \times \sqrt{1 + \sin^2(\phi' - \alpha)\tan^2(i)}.$$
$$v_{0,z'} = \frac{\cos(\phi' - \alpha)\sin(i)}{\sqrt{1 + \sin^2(\phi' - \alpha)\tan^2(i)}} v_{\rm rot} + v_{\rm c}(z_{\rm kin})$$

In the last equation, $v_{0,z'}$ is the velocity along the direction perpendicular to the sky (i.e., the spectral velocity direction). The parameter $v_{\rm c}$ is the zero-point offset, which is determined by the redshift of the emission line, is the position angle of the major axis, and i is the inclination of the disk. Assuming azimuthally constant profiles (i.e., profiles solely dependent on r), and assuming the velocity dispersion obey a 1D Gaussian distribution, yields:

$$I(r', \phi', v') = I(r')e^{(v' - v_{0,z'})^2/2\sigma_v^2}.$$

Here are r' and $v_{0,\,\rm z}$ are defined by the above equations, and $_{\rm v}$ is the one-dimensional velocity distribution. The above model cube has been created out of ten different parameters. Here is a list of the different parameters with a short description:

- **Xcen**: The position of the center of the disk in the x-direction.
- **Ycen**: The position of the center of the disk in the y-direction.
- **PA**: The position angle of the major axis (equal to in the above equations).
- **Incl**: The inclination of the disk (equal to i in the above equations).
- **I0**: The scale factor of the intensity. For most profiles this gives the maximum intensity of the emission at the center of the disk. This intensity is really a scaling of the whole cube, and providing an actual physical meaning is very difficult for any but the most simple models.
- **Rd**: The scale distance of the intensity profile, both **I0** and **Rd** define the intensity profile.

- **Vmax**: The (maximum) rotational velocity of the disk.
- **Vcen**: The systemic velocity of the galaxy.
- **Rv**: The scale distance of the velocity profile (and dispersion profile). The **Rv** and **Vmax** parameter define the velocity profile.
- **Disp**: The 1D (maximum) total velocity dispersion. Together with **Rv** this determines the velocity dispersion profile.

Besides these ten main parameters, there are an additional three parameters, **IIdx**, **VIdx** and **DIdx** that might be needed if you wish to specify the intensity, velocity or dispersion profiles with one of the parametric functions.

Note: Note on velocity dispersions. Having an infinitely thin disk with only azimuthal velocities (i.e., no radial or out-of-the-disk velocities) by its very nature implies a distribution that has zero velocity dispersion. This is because velocity dispersion implies motion perpendicular to the disk or along the radial direction, and addition of a velocity dispersion is therefore unphysical in an infinitely thin disk. However, physical disks have some thickness, and one can interpret the added velocity dispersion as the **maximum** amount of velocity dispersion (or total velocity dispersion) that is needed to make the model agree with the data. In practice, for a physical disk the total velocity dispersion, $_{\rm v}$, is composed of bulk radial motions, bulk motions perpendicular to the disk, as well as the velocity dispersion of the individual gas clouds that emit the emission lines. It, however, does not include the dispersion due to the line spread function of the instrument as well as beam smearing. These are factored out during the fitting procedure.

4.5.2 Dispersion Bulge Model

This model describes the emission line signature that arises from gravitationally bound gas that does not show bulk rotation. An example of such motion is that from stars within a classical bulge or elliptical galaxy. In such systems, the random orientation of the rotation manifest themselves as a gaussian velocity distribution around the systemic velocity of the galaxy. The dispersion-dominated bulge model is described in detail in Neeleman et al. (2021) `link this`. Here we summarize the discussion in this paper.

To describe the bulge model requires a description of the intensity profile (I) of the emission and the profile of the velocity dispersion ($_{\rm v}$). In this simple model, we assume that the bulge is spherically symmetric, and therefore we can set the galaxy frame to the sky frame. In this model, we also wish to describe the intensity profile by a 2D function (and not the intrinsic 3D density distribution). We therefore have:

$$I = I(r, \phi)$$
$$\sigma_v = \sigma_v(r, \phi)$$

For these two profiles several shapes can be chosen. The list of available profiles are given in *Qfmodels*, but it is straightforward to add your own profile to this list. Most profiles are normalized to some specific value (where possible, unity), and have a single scale distance. However, some of the profiles are parametric (i.e., the Sérsic and power profiles) and require an additional parameter, IIdx or DIdx. The profiles are normalized by the scaling factor in intensity, *I0* and velocity dispersion *Disp*.

Assuming that the velocity dispersion has a Gaussian velocity distribution around the systemic velocity of the cube yields:

$$I(r', v') = I_0 e^{-r'/R_{\rm D}} e^{(v'-v_{\rm c})^2/2\sigma_v^2}.$$

Here the primed and unprimed coordinate frame are equal. For this model, we require a total of 7 parameters:

- **Xcen**: The position of the center of the bulge in the x-direction.
- **Ycen**: The position of the center of the bulge in the y-direction.

- **I0**: The scale factor of the intensity. For most profiles this gives the maximum intensity of the emission at the center of the disk. This intensity is really a scaling of the whole cube, and providing an actual physical meaning is very difficult for any but the most simple models.
- **Rd**: The scale distance of the intensity profile, both **I0** and **Rd** define the intensity profile.
- **Vcen**: The systemic velocity of the galaxy.
- **Disp**: The 1D (maximum) total velocity dispersion.
- **Rv**: The scale distance of the velocity dispersion profile. The **Rv** and **Disp** parameter together set the velocity dispersion profile.

In addition, the parameters, **IIdx** and **DIdx** are needed if a parametric profile, such as the Sérsic profile is used.

4.5.3 Two Clumps Model

This model is a combination of two bulge models. It can be used to test if the velocity gradient in marginally resolved observations can be reproduced using simple non-rotating clumps that are moving w.r.t. each other. This model is a simple linear combination of the bulge model described above, and shows how other models can be built from the simple model above.

The model requires 2 x 7 = 14 parameters to define the model. As described in the bulge model, a possible 4 additional parameters need to be defined depending on the chosen intensity or velocity dispersion profile. Because two intensity profiles are required, the *Qubefit* instance must contain a list of profiles such as:

> qube.intensityprofile = [[‘Exponential’, None, ‘Step’], [‘Sersic’, None, None]] qube.dispersionprofile = [[‘Constant’, None, None], [‘Constant’, None, None]]

The 14 parameters that need to be defined are:

- **Xcen1, Xcen2**: The position of the center of the first (second) clump in the x-direction.
- **Ycen1, Ycen2**: The position of the center of the first (second) clump in the y-direction.
- **I01, I02**: The scale factor of the intensity of the first (second) clump.
- **Rd1, Rd2**: The scale distance of the intensity profile of the first (second) clump, both **I0** and **Rd** define the intensity profile.
- **Vcen1, Vcen2**: The systemic velocity of the first (second) clump
- **Disp1, Disp2**: The 1D (maximum) total velocity dispersion of the first (second) clump.
- **Rv1, Rv2**: The scale distance of the velocity dispersion profile of the first (second) clump. The **Rv** and **Disp** parameter together set the velocity dispersion profile.

In addition, the parameters, **IIdx1, IIdx2, DIdx1** and **DIdx2** are needed if a parametric profile, such as the Sérsic profile is used.

4.5.4 Thin Spiral Model

This model builds on the thin disk model by adding in a spiral density component. This model was used in Chittidi et al. (2021). In this model the spiral pattern does not affect the velocity of the gas, it is just modeled by a modification of the intensity profile compared to the thin disk:

$$I(r,\phi) = I_1(r,\phi) + I_2(r,\phi) = I_0[I_1(r,\phi) + I_{spf} I_2(r,\phi)]$$

Here I_1 is the density profile of the disk as in the Thin Disk model, and I_2 is the density profile of the spiral pattern. In the second equality we explicitly show how the spiral pattern is scaled with the scale factor $I_{\rm spf}$. To model the spiral

pattern we take the very simple approach that the spiral pattern has a Gaussian distribution as a function of azimuthal angle at a given radius, and the wrapping pattern is proportional to the radius:

$$I_2(r,\phi) = \begin{cases} \sum_i^{n_{sp}} e^{-\frac{(\phi-\phi_c)^2}{2\sigma_\phi^2}} & \text{if } r < R_s, \text{ where } \phi_c = \phi_0 + \frac{2\pi i}{n_{sp}} + \alpha_{sp} r \\ 0 & \text{if } r > R_s \end{cases}$$

Here the spiral pattern is defined only up to an outer radius, R_s. The sum is over the number of spiral arms, n_{sp} where the width of the spiral arms is given by the parameter . To calculate the central position of the i^{th} spiral arm, we need three parameters, the initial position of the first spiral arm, $_0$, the number of spiral arms, n_{sp} and the wrapping frequency, $_{sp}$.

The thin spiral model has besides the ten plus three parameters of the thin disk model an additional six parameters describing the spiral pattern. The list of all parameters of the model is:

- **Xcen**: The position of the center of the disk in the x-direction.
- **Ycen**: The position of the center of the disk in the y-direction.
- **PA**: The position angle of the major axis (equal to in the thin disk model).
- **Incl**: The inclination of the disk (equal to *i* in the thin disk model).
- **I0**: The scale factor of the intensity.
- **Rd**: The scale distance of the intensity profile.
- **Vmax**: The (maximum) rotational velocity of the galaxy.
- **Vcen**: The systemic velocity of the galaxy.
- **Rv**: The scale distance of the velocity profile and dispersion profile.
- **Disp**: The 1D (maximum) total velocity dispersion.
- **IIdx**: (optional) Intensity index for parameteric profiles.
- **VIdx**: (optional) Velocity index for parameteric profiles.
- **DIdx**: (optional) Dispersion index for parameteric profiles.
- **Nspiral**: Number of spiral arms.
- **Phi0**: The azimuthal angle of the first spiral arm.
- **Spcoef**: The wrapping frequency of the spiral arm ($_{sp}$ in the above equations).
- **Dphi**: Gaussian width of the spiral arm (in the above equations).
- **Ispf**: The fractional contribution of the spiral pattern. Depending on the profile, a value of unity would imply that the spiral density has a maximum contribution similar to the disk component.
- **Rs**: The cut-off radius of the spiral arm. This is a sharp cutoff, but this can be changed in the model by taking a different density profile (not a step function).

4.6 Making your own Model

One of the key features of qubefit is the ability to generate your own models. The model can be simple or complex and can have any number of parameters that you want or need. The code will look for the model in the `qfmodels.py` file, so any models you wish to add needs to be added here, and they will be available for you to call using the methods defined in the tutorial and manual. The model should be structured as:

```
def your_model(**kwargs):

   # create a model with the parameters stored in the format:
   # kwargs['par']['name_of_parameter']
   Model = ...

   # you probably want to add these line last to your function
   # to convolve the model with the PSF/LSF
   if kwargs['convolve']:
      Model = convolve(Model, kwargs['kernel'])

return Model
```

Here the model that is returned is a simple numpy array with dimension equal to the shape of the data cube. The parameters get passed in through **kwargs. This last step is necessary because of the way *emcee* reads in the model. Any of the parameters that are defined in the model can be read in, in exactly the same way as with any of the pre-defined models. This is preferably done through a setup file (see *Example Setup File*). For examples of models you can look at the pre-defined models in *Qfmodels*. If you have created a model that might be useful for others, and wish to add it to the pre-defined list, please contact me, and I can add it.

Note: This tutorial is available as a python notebook here.

4.7 Fitting a ThinDisk Model

The goal of this tutorial is to fit a thin disk model to ALMA observations of a high redshift galaxy. Our scientific aim is to measure the rotational velocity and dispersion of the galaxy. For the data, we will be using the full calibrated continuum-subtracted data cube of the ionized carbon emission from a z=4.26 galaxy ('the Wolfe disk') described in Neeleman et al. (2020). The emission from this galaxy appears to have a smooth velocity field, which is consistent with emission emitted by gas within a rotating disk.

To run this example, you will need to have access to the data. Currently the example file is part of the github code and lives in the `examples` folder, because it also is used to verify the code was installed correctly. This might change in future versions, in which case you will need to download the file and add it to the examples folder manually. The fits file `examples/WolfeDiskCube.fits` (6MB) is actually a sub-cube of the full continuum-subtracted data cube, which is signficantly larger (25MB). For the paper, a slightly different qube was used with smaller channel spacings, but to save on time we will use this cube which gives similar results.

4.7.1 Setup the model

The first step is to slice the cube to a small enough region surrounding the emission. Calculate or load the variance, load the beam (or PSF) and setting the mask. The second step is to define the model we wish to use to fit this modified data cube within the defined mask and define the initial parameters of the model. Finally we wish to create the model from our initial parameters. These steps are easiest placed inside a function. This function can then be placed into a separate file and loaded with the GUI: `qfgui`. Below is a repeat of the setup file `DiskModel.py` that is in the example folder. For a detailed description of this setup file, see the documentation on creating a setup file.

```
# import the required modules
import numpy as np
import astropy.units as u
from qubefit.qubefit import QubeFit

# set up the model
def set_model():

    # Initialize the QubeFit Instance
    DataFile = '../../examples/WolfeDiskCube.fits'
    Qube = QubeFit.from_fits(DataFile)
    Qube.file = DataFile

    # Trimming the Data Cube
    center, sz, chan = [128, 128], [45, 45], [6, 19]
    xindex = (center[0] - sz[0], center[0] + sz[0] + 1)
    yindex = (center[1] - sz[1], center[1] + sz[1] + 1)
    zindex = (chan[0], chan[1])
    QubeS = Qube.get_slice(xindex=xindex, yindex=yindex, zindex=zindex)

    # Calculating the Variance
    QSig = Qube.calculate_sigma()
    QSig = np.tile(QSig[chan[0]: chan[1], np.newaxis, np.newaxis],
                   (1, 2 * sz[1] + 1, 2 * sz[0] + 1))
    QubeS.variance = np.square(QSig)

    # Defining the Kernel
    QubeS.create_gaussiankernel(channels=[0], LSFSigma=0.1)

    # Setting the Mask
    QubeS.create_maskarray(sigma=3, fmaskgrow=0.01)

    # Define the Model
    QubeS.modelname = 'ThinDisk'
    QubeS.intensityprofile[0] = 'Exponential'
    QubeS.velocityprofile[0] = 'Constant'
    QubeS.dispersionprofile[0] = 'Constant'

    # Parameters and Priors
    PDict = {'Xcen': {'Value': 45.0, 'Unit': u.pix, 'Fixed': False,
                      'Conversion': None,
                      'Dist': 'uniform', 'Dloc': 35, 'Dscale': 20},
             'Ycen': {'Value': 45.0, 'Unit': u.pix, 'Fixed': False,
                      'Conversion': None,
```

(continues on next page)

(continued from previous page)

```
                    'Dist': 'uniform', 'Dloc': 35, 'Dscale': 20},
            'Incl': {'Value': 45.0, 'Unit': u.deg, 'Fixed': False,
                     'Conversion': (180 * u.deg) / (np.pi * u.rad),
                     'Dist': 'uniform', 'Dloc': 0, 'Dscale': 90},
            'PA': {'Value': 100.0, 'Unit': u.deg, 'Fixed': False,
                   'Conversion': (180 * u.deg) / (np.pi * u.rad),
                   'Dist': 'uniform', 'Dloc': 0, 'Dscale': 360},
            'I0': {'Value': 8.0E-3, 'Unit': u.Jy / u.beam, 'Fixed': False,
                   'Conversion': None,
                   'Dist': 'uniform', 'Dloc': 0, 'Dscale': 1E-1},
            'Rd': {'Value': 1.5, 'Unit': u.kpc, 'Fixed': False,
                   'Conversion': (0.1354 * u.kpc) / (1 * u.pix),
                   'Dist': 'uniform', 'Dloc': 0, 'Dscale': 5},
            'Rv': {'Value': 1.0, 'Unit': u.kpc, 'Fixed': True,  # not used
                   'Conversion': (0.1354 * u.kpc) / (1 * u.pix),
                   'Dist': 'uniform', 'Dloc': 0, 'Dscale': 5},
            'Vmax': {'Value': 250.0, 'Unit': u.km / u.s, 'Fixed': False,
                     'Conversion': (50 * u.km / u.s) / (1 * u.pix),
                     'Dist': 'uniform', 'Dloc': 0, 'Dscale': 1000},
            'Vcen': {'Value': 6.0, 'Unit': u.pix, 'Fixed': False,
                     'Conversion': None,
                     'Dist': 'uniform', 'Dloc': 4, 'Dscale': 20},
            'Disp': {'Value': 80.0, 'Unit': u.km/u.s, 'Fixed': False,
                     'Conversion': (50 * u.km / u.s) / (1 * u.pix),
                     'Dist': 'uniform', 'Dloc': 0, 'Dscale': 300}
            }
    QubeS.load_initialparameters(PDict)

    # Making the Model
    QubeS.create_model()

    return QubeS
```

Now that we have a function that defines the model and loads the parameters, we can easily load in an instance of `qubefit`

```
Cube = set_model()
```

4.7.2 Running the MCMC Fitting

Running the fitting code is now a simple function call where we just need to set the number of walkers, the number of steps and the number of parallel processes. In this case, to sufficiently sample the parameter space requires quite a few walkers, as well as enough steps in the chain. We will set the number of walkers to 100 and the number of steps to 1000. This will take quite a long time to run (2.5 hours on a 20-core system), so in this tutorial we will just read in the output file (an HDF5 file) for speed in the next section. You can rerun the code for yourself if you set the variable `you_are_very_patient` to `true`.

```
you_are_very_patient = False
if you_are_very_patient:
    Cube.run_mcmc(nwalkers=100, nsteps=1000, nproc=20,
                  filename='../../examples/WolfeDiskPar.hdf5')
```

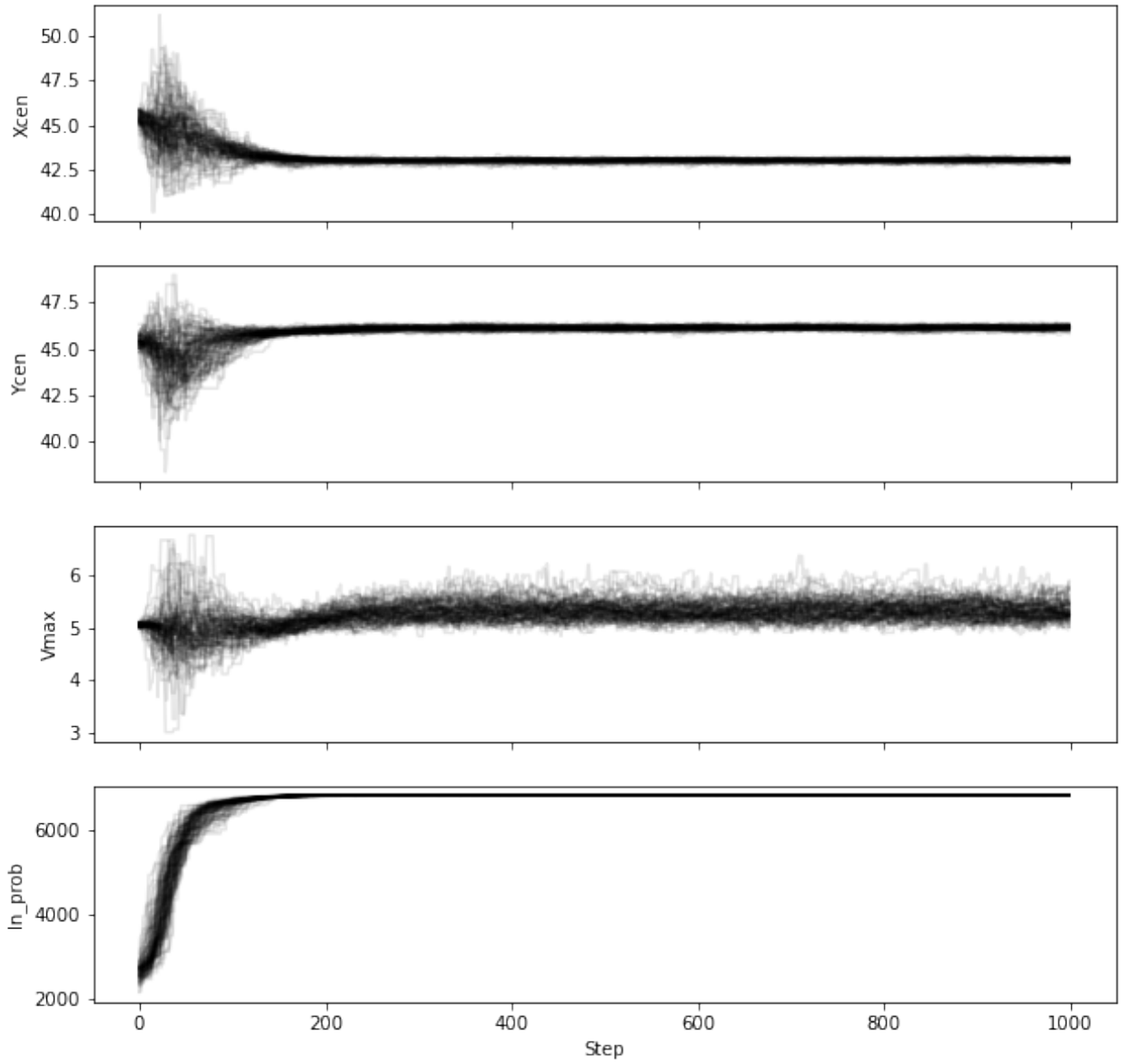
4.7.3 Analyzing the Output

After running the fitting routine, we now wish to analyze the results. The first thing that we need to do is to load the data into the qubefit instance. If you ran the fitting procedure, the chain and log-probability are stored automatically in the qubefit instance. If not, you will need to load them from file. To load the output file of the tfitting routine into the qubefit instance, we use the function `get_chainresults` with the keyword `filename`. This function will also calculate some useful quantities, such as median values for each parameter probability distribution function and 1, 2 and 3-'σ' uncertainties. For convenience, these are given in the same units as the initial parameters (i.e., the physical units not the intrinsic units). Finally the function reruns the function `create_model`, which updates the model `Cube.model` with the model using the median of each of the parameter distributions.

```
you_ran_the_fitting_code = False
if you_ran_the_fitting_code:
    Cube.get_chainresults(burnin=0.3)
else:
    # load the qubefit instance, if you didn't already do this before
    Cube = set_model()
    # burnin sets the fraction of intial steps to discard, here we want to look at the
    # full chain, so we are setting it to 0.0. To get reliable estimates on the
    # parameters, you want to discard some of the runs (typically 30% is a good
    # conservative number, see below).
    Cube.get_chainresults(filename='../../examples/WolfeDiskPar.hdf5', burnin=0.0)
```

The first thing that we can do is to check the chain for each parameter. Here is the code to look at the chain for three parameters, the x-position of the center, the y-postion of the center, and the rotational velocity. We also plot the log-probability for the chain. In this figure, each line is a single walker, and in this example we have 100 walkers.

```
import matplotlib.pyplot as plt
import numpy as np

figure, axs = plt.subplots(nrows=4, ncols=1, sharex=True, figsize=(10, 10))

pars_to_plot = [0, 1, 6]
nsteps = Cube.mcmcarray.shape[0]
nwalkers = Cube.mcmcarray.shape[1]


# plot each parameter
for idx, par_idx in enumerate(pars_to_plot):
    for walker in np.arange(nwalkers):
        axs[idx].plot(np.arange(nsteps), Cube.mcmcarray[:, walker, par_idx], alpha=0.1,
                      color='k')
    axs[idx].set_ylabel(Cube.mcmcmap[par_idx])

# plot the log_probability
for walker in np.arange(nwalkers):
        axs[-1].plot(np.arange(nsteps), Cube.mcmclnprob[:, walker], alpha=0.1, color='k')
axs[-1].set_ylabel('ln_prob')
axs[-1].set_xlabel('Step')

plt.show()
```



Note that in this figure, we are plotting the intrinsic units of the parameters (i.e., Vmax is given in pixels and not km/s). The conversion from one to the other is stored in `Cube.chainpar`. The thing here that we are looking for is to see if the chain has reached convergence. Convergence is a rather hard-to-define term, and more numerical approaches are given in the emcee documentation. However, we are looking to see at what step the ensamble average does not seem to change for any of the parameters in the model. This step marks the end of the burn-in phase. In our case, it appears the chain has converged after step ~200 (out of 1000), we therefore conservatively remove 30% of the chains as our burn-in phase to estimate the probability distribution functions of each parameter.

```
Cube.get_chainresults(filename='../../examples/WolfeDiskPar.hdf5', burnin=0.3)
```

Now we can look at some of the characteristics of the probability distribution function for each parameter. For instance, the total velocity dispersion along the line of sight 'Disp' has the following information:

```
Cube.chainpar['Disp']
```

```
{'Median': 84.70743027186644,
 '1Sigma': [82.28748593864957, 87.25928474679459],
 '2Sigma': [79.83900448628935, 89.94748456707963],
 '3Sigma': [77.50548233302094, 92.65854105589592],
 'Best': 85.40119370633211,
 'Unit': Unit("km / s"),
 'Conversion': <Quantity 50. km / (pix s)>}
```

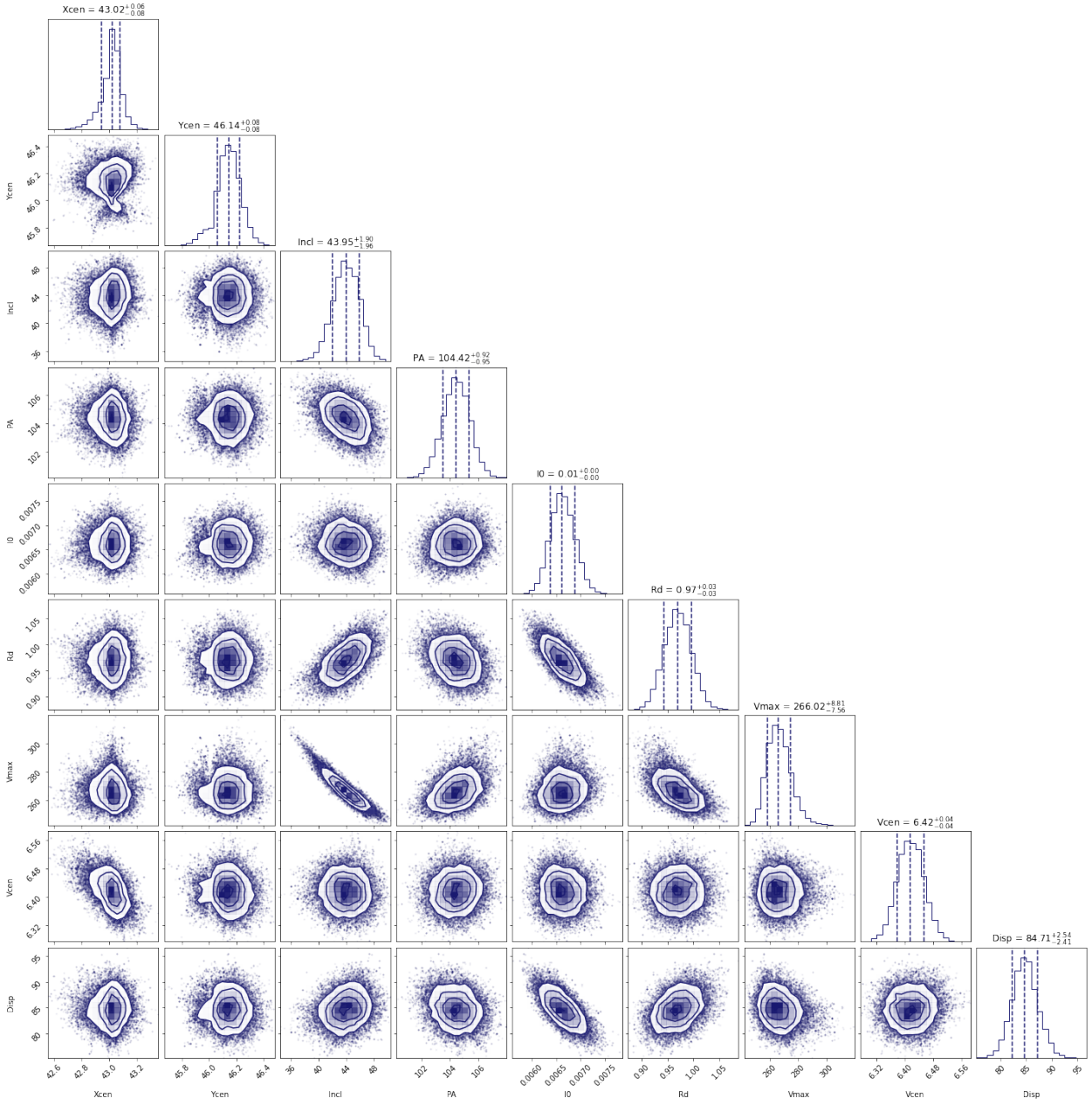
The median value and 1, 2, and 3-'σ' uncertainties. That is, these are the 0.13, 2.27, 15.87, 50, 84.13, 97.72, 99.87 percentile ranges of the probability distribution function. Note that the units have been converted into physical units, as defined by the initial parameter dictionary. The 'best' parameter is the value of the parameter space that produces the highest probability. This is **not necessarily** the best parameter to use, as the median of the distribution is often a better indicator of the overall distribution. In this case we can report the median velocity dispersion and 1-σ uncertainties as:

$\sigma_v = 85^{+2}_{-3}$ km s^{-1}

4.7.4 Making Diagnostic Plots

Besides getting the parameters for the individual parameters, we also want to see if there are dependencies between the parameters. This is easiest done using a corner plot. For this we will import the corner module.

```
import corner
from copy import deepcopy as dc

parameters = Cube.mcmcmap
labels = Cube.mcmcmap
Chain = dc(Cube.mcmcarray.reshape((-1, Cube.mcmcarray.shape[2])))

# convert to physical units
for key in parameters:
    idx = Cube.mcmcmap.index(key)
    if Cube.initpar[key]['Conversion'] is not None:
        conversion = Cube.initpar[key]['Conversion'].value
        Chain[:, idx] = Chain[:, idx] * conversion

fig = corner.corner(Chain, quantiles=[0.16, 0.5, 0.84], labels=labels,
                    color='midnightblue', plot_datapoints=True, show_titles=True)
plt.show()
```



In this figure, we can see several parameters that appear to be correlated. For instance, the inclination (*incl*) and the maximum rotational velocity (*Vmax*). This is not surprising, because if we view the disk more edge-on (higher inclination), then the would need a lower rotational velocity to produce the same line-of-sight velocity. Similarly the flux normalization (*I:math:`_0`*) and exponential scale length are inversely correlated, which is needed to keep the total amount of emission roughly constant. There also appears to be a smaller y-value for the center that gives a decent fit. This does not appear to affect the overall distribution too much though. For the paper (neeleman et al. 2020), we ran the chain longer with double the number of walkers on a cube with half the channel spacing, which produced a better result. However, these results with a coarser data cube are pretty similar.

Another interesting diagnostic plot is the total integrated flux and the velocity field of the data, the model and the residual. This will show how well the model reproduced the general characteristics of the data.

```
from mpl_toolkits.axes_grid1 import ImageGrid
```

(continues on next page)

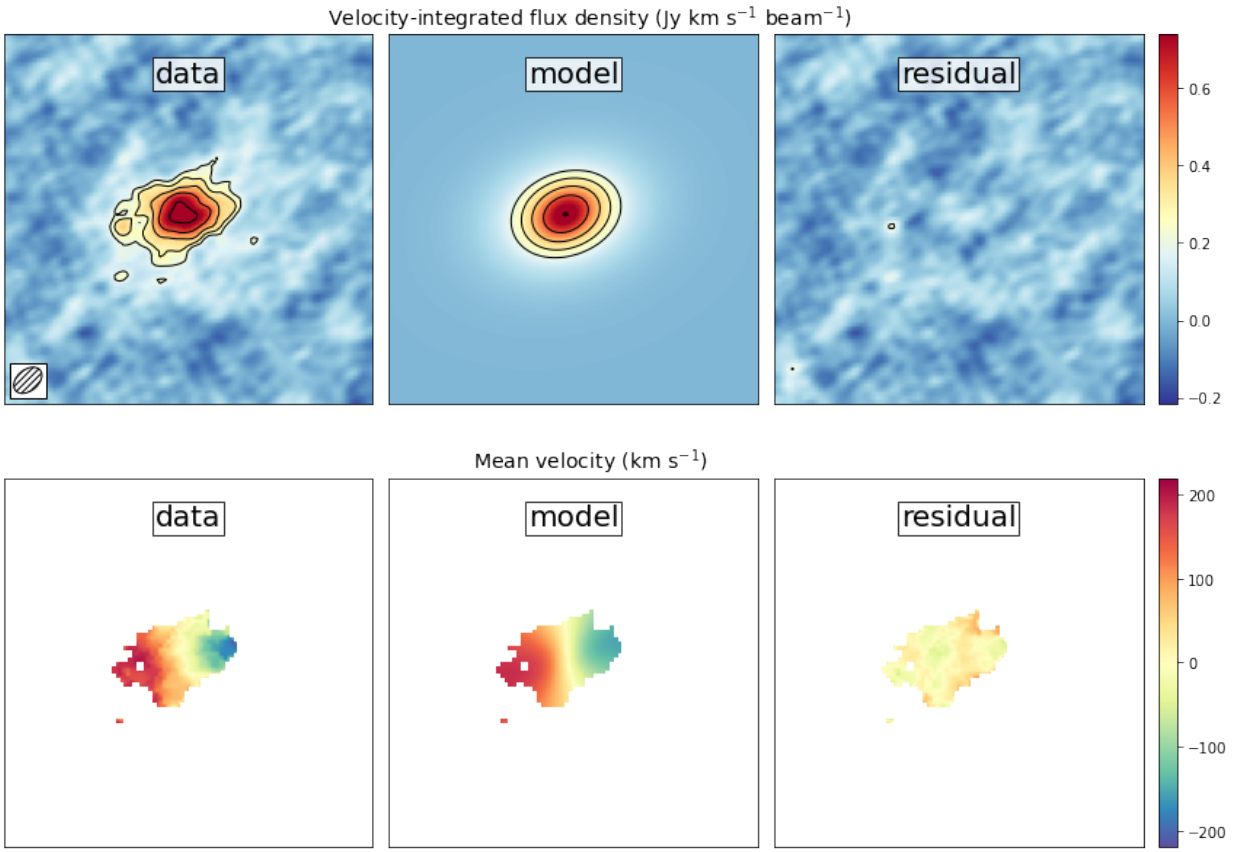
(continued from previous page)

```
from qubefit.qfutils import qubebeam
import warnings
warnings.filterwarnings("ignore")

# model, data and residual
Mom0Data = Cube.calculate_moment(moment=0)
Mom0RMS = Mom0Data.calculate_sigma()
Mom0Model = Cube.calculate_moment(moment=0, use_model=True)
Mom0Res = Mom0Data.data - Mom0Model.data

Mask = Mom0Data.mask_region(value=3*Mom0RMS, applymask=False)
CubeClip = Cube.mask_region(value=2*np.sqrt(Cube.variance[:, 0, 0]))
Mom1Data = CubeClip.calculate_moment(moment=1)
Mom1Data = Mom1Data.mask_region(mask=Mask)
Mom1Model = Cube.calculate_moment(moment=1, use_model=True)
Mom1Model = Mom1Model.mask_region(mask=Mask)
Mom1Res = Mom1Data.data - Mom1Model.data

# ranges to plot
vrange = np.array([-3.2 * Mom0RMS, 11 * Mom0RMS])
levels = np.insert(3 * np.power(np.sqrt(2), np.arange(0, 5)), 0, [-4.2, -3]) * Mom0RMS
vrange2 = np.array([-220, 220])

# figure specifics
#fig = plt.figure(figsize=(7.2, 4.67))
fig = plt.figure(figsize=(14, 9))
grid = ImageGrid(fig, (0.1, 0.53, 0.80, 0.45), nrows_ncols=(1, 3), axes_pad=0.15,
                 cbar_mode='single', cbar_location='right', share_all=True)
labels = ['data', 'model', 'residual']
for ax, label in zip(grid, labels):
    ax.set_xticks([]); ax.set_yticks([])
    ax.text(0.5, 0.87, label, transform=ax.transAxes, fontsize=20,
            color='k', bbox={'facecolor': 'w', 'alpha': 0.8, 'pad': 2},
            ha='center')

# the moment-zero images
ax = grid[0]
im = ax.imshow(Mom0Data.data, cmap='RdYlBu_r', origin='lower', vmin=vrange[0],
               vmax=vrange[1])
ax.contour(Mom0Data.data, levels=levels, linewidths=1, colors='black')
ax.add_artist(qubebeam(Mom0Data, ax, loc=3, pad=0.3, fill=None, hatch='/////',
                       edgecolor='black'))
ax = grid[1]
ax.imshow(Mom0Model.data, cmap='RdYlBu_r', origin='lower', vmin=vrange[0],
          vmax=vrange[1])
ax.contour(Mom0Model.data, levels=levels, linewidths=1, colors='black')
ax = grid[2]
ax.imshow(Mom0Res, cmap='RdYlBu_r', origin='lower',vmin=vrange[0], vmax=vrange[1])
ax.contour(Mom0Res, levels=levels, linewidths=1, colors='black')
plt.colorbar(im, cax=grid.cbar_axes[0], ticks=np.arange(-10, 10) * 0.2)

# the moment-one images
```

(continues on next page)

(continued from previous page)

```
grid = ImageGrid(fig, (0.1, 0.06, 0.80, 0.45), nrows_ncols=(1, 3),
                 axes_pad=0.15, cbar_mode='single', cbar_location='right',
                 share_all=True)
labels = ['data', 'model', 'residual']
for ax, label in zip(grid, labels):
    ax.set_xticks([]); ax.set_yticks([])
    ax.text(0.5, 0.87, label, transform=ax.transAxes, fontsize=20,
            color='k', bbox={'facecolor': 'w', 'alpha': 0.8, 'pad': 2},
            ha='center')

ax = grid[0]
im = ax.imshow(Mom1Data.data, cmap='Spectral_r', origin='lower', vmin=vrange2[0],
               vmax=vrange2[1])
ax = grid[1]
ax.imshow(Mom1Model.data, cmap='Spectral_r', origin='lower', vmin=vrange2[0],
          vmax=vrange2[1])
ax = grid[2]
ax.imshow(Mom1Res, cmap='Spectral_r', origin='lower',vmin=vrange2[0], vmax=vrange2[1])
plt.colorbar(im, cax=grid.cbar_axes[0], ticks=np.arange(-10, 10) * 100)

# some labels
fig.text(0.5, 0.49, 'Mean velocity (km s$^{-1}$)',
         fontsize=14, ha='center')
fig.text(0.5, 0.96,
         'Velocity-integrated flux density (Jy km s$^{-1}$ beam$^{-1}$)',
         fontsize=14, ha='center')
plt.show()
```



This figure reveals that the model roughly reproduces the emission and velocity field. In the velocity-integrated flux density, the contours, which start at 2σ, are almost non-existent in the residual image. This is consistent with the noise in the image. For the velocity field, the average velocity gradient is well-produced by the model. However, the residual still has some variations. This is partly due to the way that the velocity field was estimated (a simple first moment where the cube was clipped at 2σ), this can produce less reliable velocity fields in low signal to noise observations (see Appendix C of Neeleman et al. 2021 `link this`).

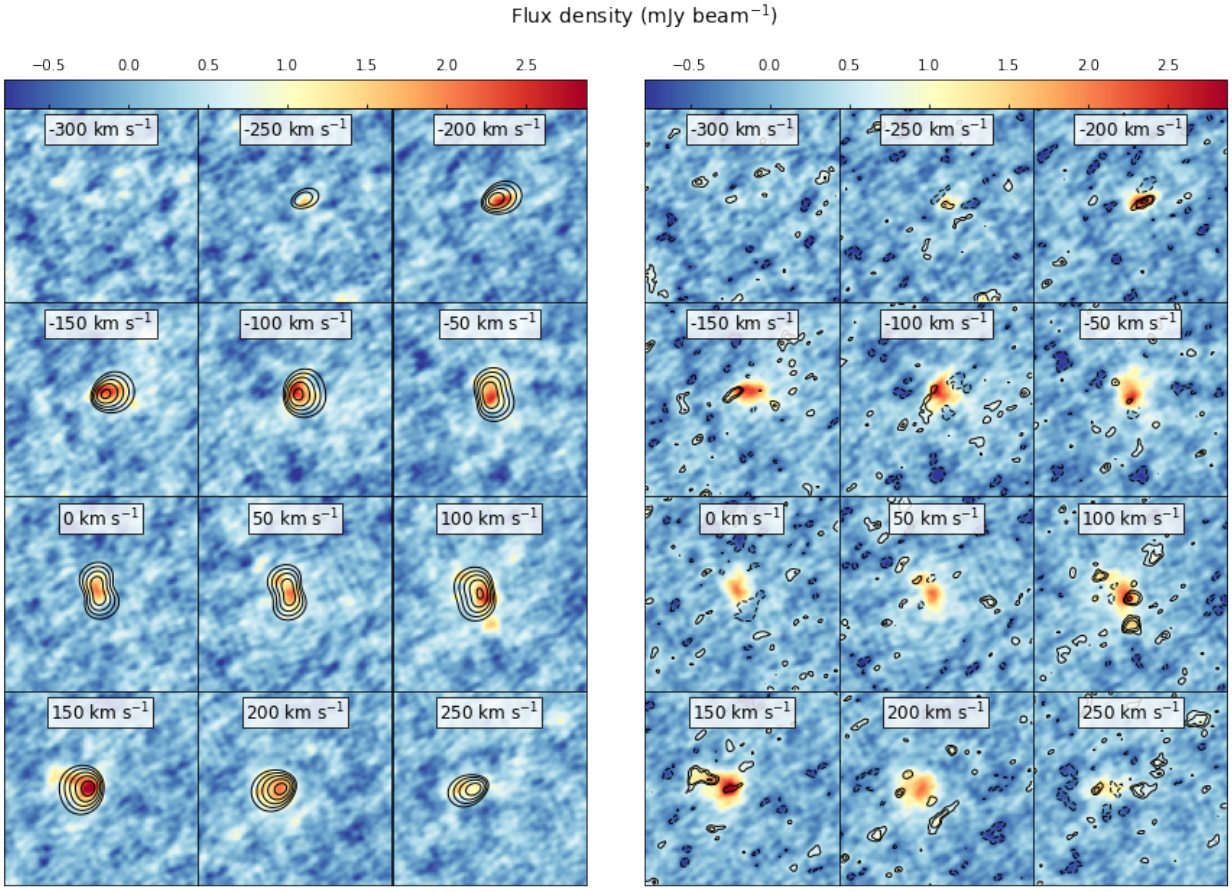
The final diagnostic plot that we will look at is the individual channel maps. This will show the most stringent constraints on how well the model fits the data. In this version, we will plot the data, and we will overlay the residuals of each channel map as contours, starting at 2σ and increasing in powers of $\sqrt{2}$.

```
fig = plt.figure(figsize=(14, 11))
grid = ImageGrid(fig, (0.06, 0.09, 0.40, 0.94), nrows_ncols=(4, 3),
                 axes_pad=0.0, cbar_mode='single', cbar_location='top',
                 share_all=True)
CubeRMS = np.sqrt(Cube.variance[:, 0, 0]) * 1E3
MedRMS = np.median(CubeRMS)
vrange = [-3 * MedRMS, 11 * MedRMS]
vel = Cube.get_velocity()
for idx, channel in enumerate(np.arange(12)):
    ax = grid[idx]
    ax.set_xticks([]); ax.set_yticks([])
    im = ax.imshow(Cube.data[channel, :, :] * 1E3, cmap='RdYlBu_r',
                   origin='lower', vmin=vrange[0], vmax=vrange[1])
    levels = (np.insert(2 * np.power(np.sqrt(2), np.arange(0, 5)), 0, -3) *
              CubeRMS[channel])
```

(continues on next page)

(continued from previous page)

```
    con = ax.contour(Cube.model[channel, :, :] * 1E3, levels=levels,
                     linewidths=1, colors='black')
    velocity = str(round(vel[channel])) + ' km s$^{-1}$'
    ax.text(0.5, 0.85, velocity, transform=ax.transAxes, fontsize=12,
            color='black', bbox={'facecolor': 'white', 'alpha': 0.8, 'pad': 2},
            ha='center')
cb = plt.colorbar(im, cax=grid.cbar_axes[0], orientation='horizontal', pad=0.1)
cb.ax.tick_params(axis='x',direction='in',labeltop=True, labelbottom=False)

grid = ImageGrid(fig, (0.50, 0.09, 0.40, 0.94), nrows_ncols=(4, 3),
                 axes_pad=0.0, cbar_mode='single', cbar_location='top',
                 share_all=True)
for idx, channel in enumerate(np.arange(12)):
    ax = grid[idx]
    ax.set_xticks([]); ax.set_yticks([])
    im = ax.imshow(Cube.data[channel, :, :] * 1E3, cmap='RdYlBu_r',
                   origin='lower', vmin=vrange[0], vmax=vrange[1])
    levels = (np.insert(2 * np.power(np.sqrt(2), np.arange(0, 5)), 0, -2) *
              CubeRMS[channel])
    con = ax.contour((Cube.data[channel, :, :] - Cube.model[channel, :, :]) * 1E3,
                     levels=levels, linewidths=1, colors='black')
    velocity = str(round(vel[channel])) + ' km s$^{-1}$'
    ax.text(0.5, 0.85, velocity, transform=ax.transAxes, fontsize=12,
            color='black', bbox={'facecolor': 'white', 'alpha': 0.8, 'pad': 2},
            ha='center')
cb = plt.colorbar(im, cax=grid.cbar_axes[0], orientation='horizontal', pad=0.1)
cb.ax.tick_params(axis='x',direction='in',labeltop=True, labelbottom=False)

fig.text(0.5, 0.96, 'Flux density (mJy beam$^{-1}$)', fontsize=14, ha='center')
plt.show()
```



This figure shows that the model recovers the overall velocity pattern quite well. The data shows a steady progression from right to left with increasing velocity. This pattern is well produced by the data. Of course, there are some individual 'clumps' in the emission that are not recovered by this smooth disk model (such as the ones seen in the 100 km s^{-1} velocity channel.

There are many more plots one can make to test the model, such as position-velocity diagrams, etc.

Note: This tutorial is available as a python notebook here.

4.8 Generating Moment Images

The goal of this tutorial is to show how you can use the `qubefit` package to make several diagnostic plots of the kinematics of your source. For this tutorial, we will be using the fully calibrated, continuum-subtracted data cube of the ionized carbon emission line from a z=4.26 galaxy discussed in Neeleman et al. (2020). In this paper, the emission is shown to arise from a smooth disk. The data cube used here has been slightly altered from the version used in the paper to reduce the size of the data file.

To run this example, you will need to have access to the data. Currently the example file is part of the github code and lives in the `examples` folder, because it also is used to verify the code was installed correctly. This might change in future versions, in which case you will need to download the file and add it to the examples folder manually. The fits file `examples/WolfeDiskCube.fits` (6MB) is actually a sub-cube of the full continuum-subtracted data cube, which is signficantly larger (25MB).

4.8.1 Image of the first 3 moments

The first image that we wish to make is a side-by-side image of the first three moments. These moments describe the velocity-integrated flux density (moment-zero), the velocity field (moment-one) and the velocity dispersion field (moment-2). In the next section, we will show an alternative way of generating the last two fields using a Gaussian fitting routine. This approach yields more robust velocity fields for these types of observations, and this is what was used in the paper.

To generate the plot, we will use the `standardfig` function, which is just a wrapper function for some `matplotlib` methods that are common among these figures. This function can be imported from the `qfutils` module.

Generating the moment images is relatively straightforward, we simply call the `calculate_moment` method of the `qube` instance. For the moment-zero we wish to calculate the noise as well, which is best done for a large cube. We therefore make two moment images, one for the full data set and one for a trimmed region. The former will only be used to estimate the noise of the data.

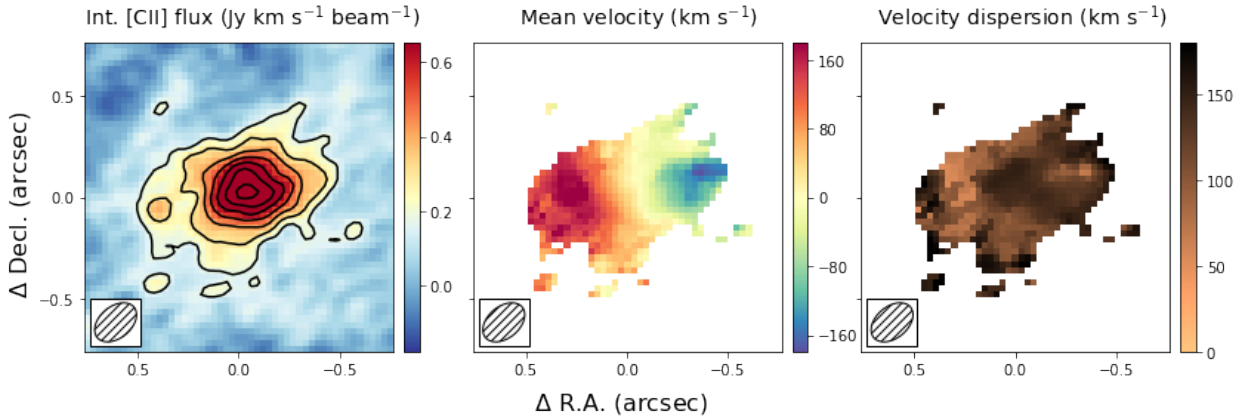
Calling the `calculate_moment` method on the full data cube will use all of the data points in the cube. This is often not desirable for the moment-one and moment-two images. For these images, we wish to first mask out some of the noise. Here we will mask out all of the values below 1σ, we will then only show the moment-one and moment-two measurments for those pixels within the 3σ contour of the moment-zero map, by applying a mask after we generated the moments.

```
from qubefit.qube import Qube
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
import numpy as np
from qubefit.qfutils import standardfig
import warnings
warnings.filterwarnings("ignore")


def make_image(Gaussian=False):
    # load the data, calculate RMS, and trim the size and make masked cube.
    Cube = Qube.from_fits('../../examples/WolfeDiskCube.fits')
    CubeSig = Cube.calculate_sigma()
    CubeS = Cube.get_slice(xindex=(100, 151), yindex=(103, 154))
    CubeSM = CubeS.mask_region(value=CubeSig)

    # calculate the noise in the moment-zero map for the 'full' data set.
    channels = (6, 19)  # channels that show emission
    tMom0 = Cube.calculate_moment(moment=0, channels=channels)
    Mom0Sig = tMom0.calculate_sigma()

    # calculate the moments (note the different qube instances used).
    Mom0 = CubeS.calculate_moment(moment=0, channels=channels)
    Mom1 = CubeSM.calculate_moment(moment=1, channels=channels)
    Mom2 = CubeSM.calculate_moment(moment=2, channels=channels)
    if Gaussian:
        Mom1, Mom2 = CubeS.gaussian_moment(mom1=Mom1, mom2=Mom2)

    # apply a mask to the moment-one and moment-two images
    Mom0Mask = Mom0.mask_region(value=3 * Mom0Sig, applymask=False)
    Mom1M = Mom1.mask_region(mask=Mom0Mask)
    Mom2M = Mom2.mask_region(mask=Mom0Mask)
```

(continues on next page)

(continued from previous page)

```
    # set up the figure and axes.
    fig = plt.figure(1, (12., 4.))
    plt.subplots_adjust(left=0.08, right=0.95, top=0.95, bottom=0.08)
    grid = ImageGrid(fig, 111, nrows_ncols=(1, 3), axes_pad=0.5,
                     cbar_mode='each', cbar_location='right', cbar_pad=0.1)

    # plot the moments.
    scale = 0.03  # pixel scale (arcsec/pixel)
    origin = (25, 25)  # origin of the x and y ticks.
    clvl = np.arange(3, 20, 2) * Mom0Sig  # contour levels to draw
    standardfig(raster=Mom0, contour=Mom0, clevels=clvl, ax=grid[0],
                fig=fig, origin=origin, scale=scale, cmap='RdYlBu_r',
                cbar=True, cbaraxis=grid.cbar_axes[0], tickint=0.5,
                vrange=[-3 * Mom0Sig, 11 * Mom0Sig], vscale=0.2, flip=True)
    standardfig(raster=Mom1M, ax=grid[1], fig=fig, cmap='Spectral_r',
                origin=origin, scale=scale, tickint=0.5, cbar=True,
                cbaraxis=grid.cbar_axes[1], vrange=[-180, 180],
                vscale=80, flip=True)
    standardfig(raster=Mom2M, ax=grid[2], fig=fig, cmap='copper_r',
                origin=origin, scale=scale, tickint=0.5, cbar=True,
                cbaraxis=grid.cbar_axes[2], vrange=[0, 180],
                vscale=50, flip=True)

    # figure text
    fig.text(0.08, 0.92, 'Int. [CII] flux (Jy km s$^{-1}$ beam$^{-1}$)',
             fontsize=14, color='black')
    fig.text(0.42, 0.92, 'Mean velocity (km s$^{-1}$)',
             fontsize=14, color='black')
    fig.text(0.70, 0.92, 'Velocity dispersion (km s$^{-1}$)',
             fontsize=14, color='black')
    fig.text(0.5, 0.02, '$\\Delta$ R.A. (arcsec)', fontsize=16, ha='center')
    fig.text(0.02, 0.5, '$\\Delta$ Decl. (arcsec)', fontsize=16, va='center',
             rotation=90)
    plt.show()

make_image(Gaussian=False)
```



This figure shows nicely the extent of the emission and the clear velocity gradient in the central velocit field, which

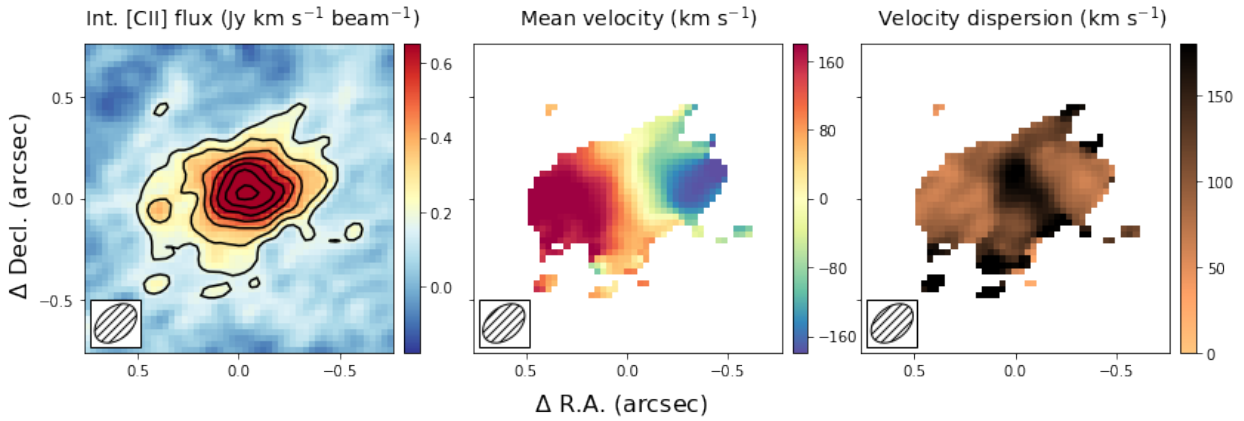
is consistent with the expected signature of a rotating disk. The right-most panel shows that the velocity dispersion is high, which is partly driven by the beam smearing of the data and partly driven by intrinsically high velocity dispersion.

4.8.2 Comparing Gaussian Spectral Fitting and Moment images.

The above section describes how to make moment images to estimate the velocity field and velocity dispersion field. This method depends on removing some of the noise of the image to produce 'nice' images. If not done properly, this could bias results, especially in the regime of low signal-to-noise (S/N). Alternatively, one can fit a functional form to the spectrum of each spatial pixel and use this functional form to estimate the mean velocity and velocity dispersion. In most high redshift (low S/N) cases, a Gaussian is used as a function form. This method has as advantage that no data is thrown away, with the cost that an assumption is made of the intrinsic shape of the profile.

In this section, we will create the same three panel figure as in the above section, but with the Gaussian image method. As you can see it simply adds a single line to the code above, which uses the moment-one and moment-two images as initial guesses to the Gaussian fitting method. The code will take a bit longe to run as it needs to fit a Gaussian to each spatial pixel in the data cube.

```
make_image(Gaussian=True)
```



The result is globally similar to the first image, but with some slight differences. The most important difference is that the velocity gradient is stronger in the central panel, and the velocity dispersion is smaller. This likely is due to some of the noise being interpreted as an increase in the velocity dispersion in the moment method, which due to beam smearing lowered the velocity gradient as well. Part of this could have been remedied using a different threshold in the masking during the moment image, but this is not needed in the Gaussian fitting approach. For low resolution imaging, where the spectral profiles are roughly Gaussian, this approach is therefore more robust.

4.8.3 Position - Velocity Diagram

Another important diagnostic figure to make is a position-velocity (p-v) diagram. The p-v diagrams are the velocity profiles along a certain line drawn through the data cube, similar to the 2D information obtained with a spectrograph using a slit. The qube class has a rather rudementary implementation of a position-velocity diagram built in. There are much more advanced packages out there (e.g., pvextractor), but this implementation will give you a good first glance.

The pv data can be loaded using a simple call of the method `pvdiagram`. The output is a dictionary with some useful plotting information in it, which can be used in conjuction with matplotlib.pyplot to make the pv-dagram image.

```
# load the data, calculate RMS, and trim the size and make mask cube
Cube = Qube.from_fits('../../examples/WolfeDiskCube.fits')
CubeSig = Cube.calculate_sigma()
```

(continues on next page)

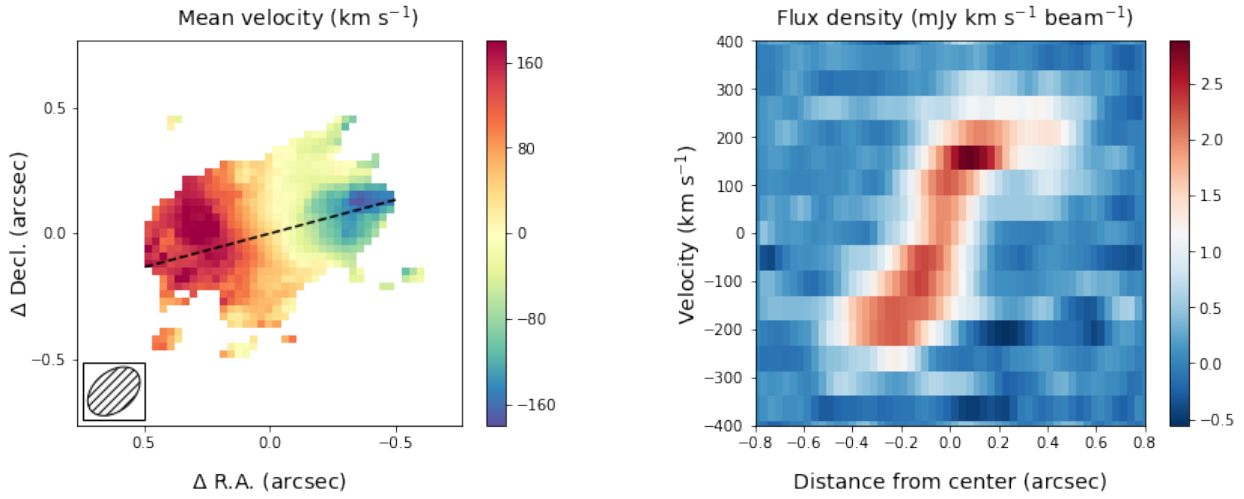
(continued from previous page)

```
CubeS = Cube.get_slice(xindex=(100, 151), yindex=(103, 154))
CubeSM = CubeS.mask_region(value=CubeSig)

# make the moment-one image
channels = (6, 19)  # channels that show emission
tMom0 = Cube.calculate_moment(moment=0, channels=channels)
Mom0Sig = tMom0.calculate_sigma()
Mom0 = CubeS.calculate_moment(moment=0, channels=channels)
Mom0Mask = Mom0.mask_region(value=3 * Mom0Sig, applymask=False)
Mom1 = CubeSM.calculate_moment(moment=1, channels=channels)
Mom1M = Mom1.mask_region(mask=Mom0Mask)

# set up the figure and axes.
fig, axs = plt.subplots(1, 2, figsize=(12, 4.5))
fig.subplots_adjust(left=0.07, bottom=0.15, right=0.97, top=0.94,
                    wspace=0.40, hspace=None)

# plot the moment-one
center = (25, 25)  # origin of the x and y ticks.
scale = 0.03  # pixel scale (arcsec/pixel)
PA = 105  # the angle of the pv-line (PA of major axis)
standardfig(raster=Mom1M, ax=axs[0], fig=fig, cmap='Spectral_r',
            origin=center, scale=scale, tickint=0.5, cbar=True,
            cbaraxis=None, vrange=[-180, 180], vscale=80, flip=True)

# the pv line to draw
x = np.array([-0.5, 0.5])
y = -1 * (np.tan((PA + 90) * np.pi / 180) * x)
# the -1 is needed because the axes are flipped.
axs[0].plot(x, y, ls='--', color='black')
axs[0].text(0.5, -0.16, '$\\Delta$ R.A. (arcsec)', fontsize=14,
            color='black', transform=axs[0].transAxes, ha='center')
axs[0].text(-0.18, 0.5, '$\\Delta$ Decl. (arcsec)', fontsize=14,
            color='black', transform=axs[0].transAxes, va='center', rotation=90)

# get the pv-diagram
PV = CubeS.pvdiagram(PA, center, width=5, scale=0.03)
im = axs[1].imshow(PV['pvdata'] * 1E3, aspect='auto', origin='lower',
                   cmap='RdBu_r', extent=PV['extent'], alpha=1.0)
cbr = plt.colorbar(im, ax=axs[1])
axs[1].set_xlim(-0.8, 0.8)
axs[1].set_ylim(-400, 400)
axs[1].text(0.5, -0.16, 'Distance from center (arcsec)', fontsize=14,
            color='black', transform=axs[1].transAxes, ha='center')
axs[1].text(-0.21, 0.5, 'Velocity (km s$^{-1}$)', fontsize=14,
            color='black', transform=axs[1].transAxes, va='center',
            rotation=90)

# additional text
fig.text(0.15, 0.97, 'Mean velocity (km s$^{-1}$)', fontsize=14,
         color='black')
fig.text(0.61, 0.97, 'Flux density (mJy km s$^{-1}$ beam$^{-1}$)',
```

(continues on next page)

(continued from previous page)

```
        fontsize=14, color='black')
plt.show()
```



The panel on the right shows the pv-diagram along the dashed line in the left panel, which is the measured major axis of the galaxy. The typical S-shape curve is a signature of a disk galaxy. Other pv-diagrams can also be drawn such as along the minor axis of the galaxy by simply changing the angle in the method call.

4.9 Qube

This is the base class for the full package. It defines some basic functions, such as reading in fits files from different instruments. It also gives access to some common operations you might wish to perform, such as getting a spectrum, estimating the noise and creating moment maps.

class qubefit.qube.**Qube**

Initate the Qube class.

Class for handeling of a variety of data cubes. This includes reading in the data cube, and preforming basic operations on the data cube. It also allows for a couple of higher level operations, such as moment generation and simple position-velocity diagrams. The cube has been primarily tested and written for (sub)-mm data cubes, but can work with a range of optical / NIR data cubes as well.

calculate_moment(*moment=0*, *channels=None*, *restfreq=None*, *use_model=False*, ***kwargs*)

Calculate the moments for the data.

This method will determine the moments for the data. These are the moments w.r.t. the spectral axis, as is typical. A detailed description of the different moments can be found elsewhere. In summary, the moment 0 will yield an integrated measurement of the flux density, the moment 1 will give an estimate of the velocity field and the moment 2 is an estimate of the velocity dispersion.

Parameters

- **moment** (*INT, optional*) – Determines which moment to return. Currently only moments up to and including 2 are defined. The default is 0.
- **channels** (*TUPLE or NUMPY.ARRAY, optional*) – The channels to use in creating the moment If not set, the full data cube is used. The channel ranges can be given as a numpy array or 2-element tuple with the first and last channel, where the last channel is NOT included. The default is None.

- **restfreq** (*FLOAT, optional*) – The rest frequency to use (in Hz). It is recommended that this is defined directly in the Qube instance, i.e., when it it read in or right after. However, for convenience this can be (re-)defined here. The default is None.
- **use_model** (*BOOLEAN, optional*) – If set to true, the moment will be calculated from the model data attribute instead of the data attribute. The default is False.
- ****kwargs** (*VARIED, optional*) – This method will take in the keywords defined in the method get_velocity. In particular the convention keyword which can change the output unit of the moments (see below)

Raises

NotImplementedError – Will raise a NotImplementedError if the moment keyword is set to something else than 0, 1, 2.

Returns

To output is a full instance of Qube where the data attribute containts the moment calculation. The units of the output varies and depend on the input and the convention used to get the velocities. In general the first moment will be an integrated flux density. For example if the data of the cube is in Jy/beam and the velocities are in km/s (the default), then the moment-0 will have the units of Jy * km/s / beam. The moment-1 and moment-2 will have the same unit as the velocity convention which for the default is km/s.

Return type

Qube

calculate_sigma(*ignorezero=True*, *fullarray=False*, *channels=None*, *use_residual=False*, *plot=False*, *plotfile='./sigma_estimate.pdf'*, ***kwargs*)

Calculate the Gaussian noise of the data.

This method will take the measurements of each channel, create a histogram and fits a Gaussian function to the histogram. It will return the Gaussian sigma of this fit as a noise estimate. This assumes that 1) the nosie is Gaussian and 2) the signal does not signficantly affect the noise property. If it does, then the data should be masked first.

Parameters

- **ignorezero** (*BOOLEAN, optional*) – If set, this will ignore any zero values in the array for the gaussian fitting. This is useful if masked values are set to zero. The default is True.
- **fullarray** (*BOOLEAN, optional*) – If set, the returned array will be a numpy array with the same shape as the input data, where each point corresponds to the sigma value of that channel. The default is False.
- **channels** (*NUMPY.ARRAY, optional*) – The list of channels to calculate the sigma value for. This can be either a list or a numpy array. When not specfied or set to None, the full range is taken. The default is None.
- **plot** (*BOOLEAN, optional*) – If set a QA file is made of the Gaussian fits, to determine the fit of the Gaussian for each channel. The default is False.
- **plotfile** (*TYPE, optional*) – The file in which to save the sigma QA plot. The default is './sigma_estimate.pdf'.
- **doguess** (*BOOLEAN, optional*) – If set, the code will calculate the bin size and the intial estimates for the gaussian fit. The default is True.
- **gausspar** (*LIST, optional*) – If a list is given it is taken as the initial guesses for the Gaussian fit. This has to be set with the number of bins for the histogram, otherwise doguess has to be set to True, which will overwrite the values given here. The default is None.

- **bins** (*INT, optional*) – The number of bins to use in the histogram. If not set, doguess needs to be set, which calculates this value.

Returns
The Gaussian noise (sigma) for each requested channel. If full array is set, it will generate a new instance of Qube and will return the array in the 'data' attribute for this Qube.

Return type
FLOAT, NUMPY.ARRAY or Qube instance

classmethod from_fits(*fitsfile*, *extension=0*, ***kwargs*)

Instantiate a Qube class from a fits file.

Read in a fits file. This is the primary way to load in a data cube. The fits file will be parsed for a header and data. These will be stored in the Qube class. Several other attributes will be defined (most notably the beam, shape and instrument).

Parameters

- **cls** (*QUBE*) – Qube Class instance.
- **fitsfile** (*STRING*) – The name of the fits file to be read in.
- **extension** (*INT, optional*) – The extension of the fots file to read in. The default is 0.
- ****kwargs** (*DICT, optional*) – keyword arguments that can be directly passed into the Qube class.

Returns
Will return an instance of the qubefit.qube.Qube class. The instance should have the data and header attribute populated, as well as several other minor data attributes.

Return type
Qube

gaussian_moment(*mom1=None*, *mom2=None*, *channels=None*, *use_model=False*, *return_amp=False*, ***kwargs*)

Calculate the Gaussian 'moments' of the cube.

This method will fit a Gaussian to the spectrum of each spatial pixel. The velocity field and velocity dispersion field are then estimated from the velocity shift of the Gaussian and the width of the Gaussian, respectively. This method is often more robust in the case of low S/N and lower resolution (see the reference paper). To help provide better convergence of the fitting routine, it is recommended to supply initial guesses to the velocity field and dispersion field, which probably come from the method 'calculate_moment'.

This method can be very slow for a large data cube in the spatial direction.

Parameters

- **mom1** (Qube, *optional*) – This is a qube instance that contains the initial guess for the velocity field. If not given this estimate will be calculated from the data cube (not recommended). The default is None.
- **mom2** (Qube, *optional*) – This is a qube instance that contains the initial guess for the velocity dispersion field. If not given this estimate will be calculated from the data cube (not recommended). The default is None.
- **channels** (*NUMPY.ARRAY or TUPLE, optional*) – The channels to use in creating the gaussian 'moments' If not set, the full data cube is used. The channel ranges can be given as a numpy array or 2-element tuple with the first and last channel, where the last channel is NOT included. It is recommended to use a large range in channels, so the zero level can be accurately estimated, which is one of the strengths of this appraoch. The default is None.

- **use_model** (*BOOLEAN, optional*) – If set to true, the moment will be calculated from the model data attribute instead of the data attribute. The default is False.
- ****kwargs** (*VARIED , optional*) – This method will take in the keywords defined in the method get_velocity. In particular the convention keyword which can change the output unit of the moments.

Returns

- **mom1** (*Qube*) – A qube instance where the data attribute contains the velocity field estimate. Typically in km/s.
- **mom2** (*Qube*) – A qube instance where the data attribute contains the velocity dispersion field estimate. Typically in km/s.

get_slice(*xindex=None*, *yindex=None*, *zindex=None*)

Slice the data cube.

This method will slice the data cube to extract smaller cubes or individual channels. For generality, the two spatial dimensions are the x and y indices. The frequency/velocity/wavelength dimension is denoted by the zindex. This method also updates the header files with the new information.

Parameters

- **xindex** (*TUPLE or NUMPY.ARRAY, optional*) – The indices to use to slice in the x-direction. This can either be a tuple, which will be intepreted as a range starting at the first up to but NOT including the second value, or a numpy array of indices. The default is None.
- **yindex** (*TUPLE or NUMPY.ARRAY, optional*) – The indices to use to slice in the y-direction. This can either be a tuple, which will be intepreted as a range starting at the first up to but NOT including the second value, or a numpy array of indices. The default is None.
- **zindex** (*TUPLE or NUMPY.ARRAY, optional*) – The indices to use to slice in the z-direction. This can either be a tuple, which will be intepreted as a range starting at the first up to but NOT including the second value, or a numpy array of indices. The default is None.

Raises

ValueError – Can only crop two and three dimensional data. If another dimension is selected, this will result in a ValueError.

Returns

Return a Qube instance with the updated data and header of the sliced data set (if present, the model, sig and variance are will also be siced).

Return type

Qube

get_spec1d(*continuum_correct=False*, *limits=None*, *beam_correct=True*, *use_model=False*, ***kwargs*)

Generate a 1D spectrum from the data cube.

This method will extract a spectrum from the data cube integrated over an area This is a simple sum over all of the pixels in the data cube and therefore it probably is most useful after some masking has been applied using the method 'mask_region'. Standard is to correct for the beam to get a flux density from the region, i.e., if the data cube is in Jy/beam then the result will be a flux density in Jy. It is also possible to correct for any residual continuum.

Parameters

- **continuum_correct** (*BOOLEAN, optional*) – If set, this will correct the spectrum for any potential continuum flux by fitting a second order polynomial to the spectrum outside the channels given by limits. The default is False.
- **limits** (*LIST, optional*) – Two-element list which contains the limits outside which the continuum will be fit, if set. The default is None.
- **beam_correct** (*BOOLEAN, optional*) – Correct the flux by dividing by the area of the beam. This should be the default when dealing with interferometric data in Jy/beam. The default is True.
- **use_model** (*BOOLEAN, optional*) – If set to true, the moment will be calculated from the model data attribute instead of the data attribute. The default is False.
- ****kwargs** (*VARIED , optional*) – This method will take in the keywords defined in the method get_velocity. In particular the convention keyword which can change the output unit of the moments.

Returns
Summed flux and 'Velocity' are returned. The first is in the same units as the data cube, whereas the latter quantity has the units as defined by the convention. This is either a velocity, frequency or wavelength array. The default is to return a velocity array in km/s.

Return type
NUMPY.ARRAY, NUMPY.ARRAY

get_velocity(*convention='radio'*, *channels=None*, *as_quantity=False*)

Get 'velocities' of the channels in a data cube.

This function will take the header information of the third dimension and the rest frequency defined also in the header, and convert the frequency values (using astropy's units and equivalencies package) to velocities.

Parameters

- **convention** (*STRING, optional*) – The convention used for converting the frequencies to velocities. For possible choices see the astropy.equivalencies documentation. They are 'radio', 'optical', 'relativistic'. In addition, the 'frequency' and 'wavelength' can be give, which will return the array in frequency (Hz) or wavelength (m). The default is 'radio'.
- **channels** (*NUMPY.ARRAY, optional*) – If None then get velocities for all of the channels in the data array. otherwise only the velocities of those channels that are specfied are returned. The default is None.
- **as_quantity** (*BOOLEAN, optional*) – If set, it will return the array as a astropy.quantity instead of a unitless numpy.array. The default is False.

Raises
ValueError – Method will raise a ValueError if the 'CTYPE3' in the header has a value that is unknown.

Returns
'Velocity' array (in km/s or Hz, m)

Return type
NUMPY.ARRAY or ASTROPY.QUANTITY

get_velocitywidth(***kwargs*)

Calculate the channel width.

This is a small function that will get the velocities and calculate the median distance between them (i.e., width of the velocity channel). It inherits the same keywords as the get_velocity method.

mask_region(*ellipse=None*, *rectangle=None*, *value=None*, *moment=None*, *channels=None*, *mask=None*, *applymask=True*)

Mask a region of the data cube.

This method will mask the data cube. Several options for masking are available. Either a rectangle or ellipse for regions can be chosen or a specfiic value in the data cube. Finally, an option is to mask the region from a summed cube (moment-zero). A mask can also be read in and finally either the mask is returned as a numpy array, or the mask is applied and another Qube instance is returned. It should be noted that the masks are multiplicative.

Parameters

- **ellipse** (*TUPLE, optional*) – The 5-element tuple describing an ellipse (xc, yc, maj, min, ang). Each of the values in the tuple correspond to: the center of the ellipse in the x- and y-direction, (xc and yc), the length of the major and minor axis (maj and min) and the angle of the major axis (ang). The default is None.
- **rectangle** (*TUPLE, optional*) – The 4-element tuple describing the rectangles bottom left (xb, yb) and uppper right (xt, yt) corners (xb, yb, xt, yt). The default is None.
- **value** (*FLOAT or LIST, optional*) – Value used to mask everything below this value. This value can also be a list with the same size as the number of channels, in which case the comparison per channel is done piece-wise. The default is None.
- **moment** (*FLOAT, optional*) – First a temporary moment image is created, using the channels keyword. Then the cube will be masked for all pixels below the value set by moment, which is given in terms of the Gaussian noise sigma. The default is None.
- **channels** (*TUPLE or NUMPY.ARRAY, optional*) – Values used to slice the data in the spectral direction. This only needs to be set if the moment mask is requested, and if not set, it will use the full spetral range. The default is None.
- **mask** (*NUMPY.ARRAY, optional*) – Predefined numpy array to use as a mask. The default is None.
- **applymask** (*BOOLEAN, optional*) – This will apply the mask and return a Qube instance. If this is not set the mask will be returned as a numpy.array. The default is True.

Returns

If applymask is true the mask data set will be returned as a Qubefit instance, if false, a numpy array of the mask will be returned.

Return type

NUMPY.ARRAY or Qube instance

pvdiagram(*PA*, *center*, *width=3.0*, *vshift=0.0*, *scale=1.0*, *use_model=False*, ***kwargs*)

Create the PV diagram along the given line.

This method will take a data or model attribute from the qube instance, and create a 2D image with position along the axis and velocity on the second axis. It uses scipy's 'rotate' to rotate the datacube along the axis defined by the PA and center and then extracts the total flux along the 'slit' with a width given by the width keyword. The final 2D output has been flipped (if needed) to have the velocity increasing upward and the extent of the pvdata is returned.

NOTE: Rotate introduces a problem in that it does not conserve the total flux of a cube when rotated. This effect is often small (<10%), but should be kept in mind when looking at these pv diagrams.

Parameters

- **PA** (*FLOAT*) – The position angle of the axis used to generate the PV diagram in degrees east of north.

- **center** (*Tuple*) – Two-element tuple describing the center of the PV line, where the first value is the x position and the second value is the y position.
- **width** (*INT, optional*) – The 'width' of the 'slit'. The slit width is actually defined as two times this width plus 1 (2 x width + 1). Because of the issues with rotation, it is not advisable to set this value to 0 (i.e., a width of exactly 1 pixel). The default is 3.
- **vshift** (*FLOAT, optional*) – This keyword allows the velocity to be shifted w.r.t. to the zero velocity as defined by the rest frequency of the cube. This is useful to make small velocity corrections.
- **scale** (*FLOAT, optional*) – The scale to use for the pixels (x and y). This can provide a convenient way to convert the distance along the PV line from the unit pixels to more useful quantities such as kpc or arcsec. The default is 1.
- ****kwargs** (*VARIED , optional*) – This method will take in the keywords defined in the method get_velocity. In particular the convention keyword which can change the output unit of the moments.

Returns
The method returns a dictionary with four items defined. 'pvdata', which is a numpy.array that contains the PV array of values'. 'extent', which is a four-element tuple with the extrema of the extent of the pv array, i.e., (xmin, xmax, vmin, vmax). This can be used with the extent keyword of matplotlib.pyplot.imshow. 'position', which is a numpy.array of positions defining each pixel along the PV line. Finally, 'velocity' is a numpy.array of velocities defining each channel of the spectrum.

Return type
DICT

to_fits(*fitsfile='./cube.fits'*, *overwrite_beam=True*)

Write the qube instance to fits file.

Parameters
fitsfile (*STRING, optional*) – The name of the fits file to save the qube instance to. The default is './cube.fits'.

Return type
None.

4.10 Qubefit

This is an extension of the base class *Qube*. Besides the functions available in this class, it defines several additional functions needed to fit a model to the data and to analyze this model.

class qubefit.qubefit.**QubeFit**(*modelname='ThinDisk'*, *probmethod='ChiSq'*, *intensityprofile=None*, *velocityprofile=None*, *dispersionprofile=None*)

Initiate the Qubefit class.

Class that is used for fitting data cubes. This class is based on the Qube class and inherits all of its functions. The additional functions defined here are to define a model, run the fitting routine and analyze the results.

When it is initiated, the following data attributes are defined and can be explcitely set: modelname, probmethod, intensityprofile, velocity profile, and dispersionprofile.

Parameters

- **modelname** (*STRING, optional*) – The name of the model used for the fitting. The default is 'ThinDisk'.

- **probmethod** (*STRING, optional*) – The name of the probability function to use. The default is 'ChiSq'.
- **intensityprofile** (*LIST, optional*) – List of three strings with the names of the profiles to use for the intesity of the model. The three strings correspond to the three dimensions used to describe the model. For a cylindrical coordinate system this is r, phi and z in that order. The default is ['Exponential', None, 'Exponential'].
- **velocityprofile** (*LIST, optional*) – List of three strings with the names of the profiles to use for the velocity of the model. The three strings correspond to the three dimensions used to describe the model. For a cylindrical coordinate system this is r, phi and z in that order. The default is ['Constant', None, None].
- **dispersionprofile** (*TYPE, optional*) – List of three strings with the names of the profiles to use for the disperdsion of the model. The three strings correspond to the three dimensions used to describe the model. For a cylindrical coordinate system this is r, phi and z in that order. The default is ['Constant', None, None].

calculate_chisquared(*reduced=True*, *adjust_for_kernel=True*)

Calcuate the (reduced) chi-squared statistic.

This method will calculate the (reduced) chi-squared statistic for the given model, data and pre-defined mask. The value can be adjusted for the oversampling of the beam using a simplified assumption, which is described in detail in the reference paper.

Parameters

- **reduced** (*BOOLEAN, optional*) – If set, this will calculate the reduced chi-square statistic, by dividing by the degrees of freedom and optionally by the adjustment factor to account for the oversampled beam. The default is True.
- **adjust_for_kernel** (*BOOLEAN, optional*) – If set, it will apply an adjustement factor to the reduced chi-suared statistic to account for oversampling the beam. If the mask is only sparsely sampled, this adjustment factor should be set to False. The default is True.

Raises

AttributeError – This method will return an AttributeError if no mask has yet been defined.

Returns

chisq – The (reduced) chi-square statistic of the model within the mask.

Return type

FLOAT

calculate_ksprobability()

Calculate the Kolmogorov-Smirnov probability of the model

create_gaussiankernel(*channels=None*, *lsf_sigma=None*, *kernelsize=4*)

Create a Gaussian kernel.

This method will generate a Gaussian kernel from the data stored in the Qube. It will look for the beam keyword and generate a gaussian kernel from the shape of the synthesized beam given here. Several different kernels can be returned. 1) A list of 3D kernels -one for each spectral channel- that takes into account both a varying LSF and a varying PSF (set this by selecting a range of channels and LSFSigma) 2) A list of 2D kernels -one for each spectral bin- that takes into account a varying PSF, but no LSF (set this by selecting a range of channels and LSFSigma=None) 3) A 3D kernel that has constant PSF and LSF Only the last option is currently implemented in the code and should therefore be used (i.e., specify a single channel not a list).

This method will populate the kernel and kernelarea data attributes of the qubefit instance.

Parameters

- **channels** (*LIST, NUMPY.ARRAY or INT, optional*) – The channels to use to create the beam. If this is not specified, then the center of the cube will be used. If multiple channels are specified, then a list of kernels will be given, one for each channel. This is useful when the kernel/PSF changes with the channel. This option, however, is currently NOT implemented in the code and therefore just a single channel should be specified. The default is None.
- **lsf_sigma** (*FLOAT or NUMPY.ARRAY, optional*) – The width of the line spread function. Note that this is the sigma or square root of the variace, NOT the FWHM!. The unit for this value is pixels. If this is not specfied, then the LSF is assumed to be neglible, which is often ok for ALMA data, which has been averaged over many channels. The default is None.
- **kernelsize** (*FLOAT, optional*) – The size of the kernel in terms of the sigma of the major axis in pixels (bsig). The actual size of the kernel will be a cube that has a size for the two spatial dimensions: 2 (n * bsig) + 1, where n is the kernelsize. The default is 4.

Raises

AttributeError – Function will raise an attribute error if the beam has not been propertly defined.

Returns

None

Return type

NoneType

create_maskarray(*sampling=2.0*, *bootstrapsamples=200*, *regular=None*, *sigma=None*, *nblobs=1*, *fmaskgrow=0.01*)

Generate the mask for fitting.

This will generate the mask array. It should only be done once so that the mask does not change between runs of the MCMC chain, which could/will result in slightly different probabilities for the same parameters. The mask returned is either the same size as the data cube (filled with np.NaN and 1's for data to ignore/include) or an array with size self.data.size by bootstrapsamples. The first is used for directly calculating the KS or ChiSquared value of the given data point, while the second is used for the bootstrap method of both functions.

Several mask options can be specified. It should be noted that these masks are muliplicative. This method will set the maskarrray data attribute.

Parameters

- **sampling** (*FLOAT, optional*) – The number of samples to choose per beam/kernel area. This keyword is only needed for the sparse sampling methods and for bootstrap arrays. The default is 2.
- **bootstrapsamples** (*INT, optional*) – The number of bootstrap samples to generate. This keyword is only needed if the probability method is set to 'BootKS' or 'BootChiSq'. The default is 200.
- **regular** (*TUPLE, optional*) – If a two element tuple is given, then a regular grid will be generated that is specified by the number of samplings per kernelarea which will go through the given tuple. The default is None.
- **sigma** (*FLOAT, optional*) – If given, a mask is created that will just consider values above the specified sigma (based on the calculated variance). Only the n largest blobs will be considered (specified optionally by nblob). It will grow this mask by convolving with the kernel to include also adjacent 'zero' values.

- **nblobs** (*INT, optional*) – The number of blobs to consider in the sigma-mask method. The blobs are ordered in size, where the largest blob which is the background is ignored. If set to 1, this will select only the largest non-background blob. The default is 1.
- **fmaskgrow** (*FLOAT, optional*) – The fraction of the peak value which marks the limit of where to cut of the growth factor. If set to 1, the mask is not grown. This value allows to include some adjacent zero values to be included.

Returns
None

Return type
NoneType

create_model(*do_convolve=True*)

Create the model cube with the given parameters and model.

This method will take the stored parameters (in self.par) and will generate the requested model. NOTE: currently no checking is done that the parameters are actually defined for the model that is requested. This will likely result in a rather benign AttributeError.

This method will set the model data attribute using the parameters stored in the par data attribute and the model defined elsewhere.

Parameters
do_convolve (*BOOLEAN, optional*) – If set, the model will be convolved with the beam. The default is True.

Returns
None

Return type
NoneType

get_chainresults(*filename=None*, *burnin=0.3*, *reload_model=True*, *load_best=False*)

Get the results from the MCMC run either from file or memory.

Calling this method will generate a dictionary with the median values, and 1, 2, 3 sigma ranges of the parameters. These have been converted into their original units using the conversions in the the initial parameter attribute (initpar).

If not present, this method will populate the self.mcmcarray and self.mcmclnprob attributes with the values in the file. It will also populate the self.chainpar data attribute with a dictionary with the median parameters, uncertainty and unit conversions.

Parameters

- **filename** (*STRING, optional*) – The name of the HDF5 file to load. If the filename is set to None, the assumption is that the qubefit instance already has the MCMC chain and log probability loaded into their respective instances. If this is not the case, the code will exit with an AttributeError. The default is None.
- **burnin** (*FLOAT, optional*) – The fraction of runs to discard at the start of the MCMC chain. This is necessary as the chain might not have converged in the very beginning. One should check the chain to make sure that this value is correct. The default is 0.3.
- **reload_model** (*BOOLEAN, optional*) – reload the model cube with the updated parameters. The default parameters to use are the median values of each parameter. The default is True.

- **load_best** (*BOOLEAN, optional*) – If set, then instead of the median parameters, the combination of parameters that yielded the highest probability will be chosen. The default is False.

Raises
AttributeError – An attribute error will be raised if the needed mcmcarray and lnprobability keys are not set in the qubefit instance and no filename is given.

Returns
None

Return type
NoneType

load_initialparameters(*parameters*)

Load the intial parameter dictionary into the qubefit instance.

This method will load in the initial parameters from a dictionary. The dictionary is transfered into a data attribute, in addition several other attributes are created which are needed for the model and the MCMC fitting procedure.

This method will set the initpar, par, mcmcpar, mcmcmap, priordist and mcmcdim data attributes. The initpar attribute is a simple copy of the parameter dictionary. The par data attribute contains the value of the parameters converted into intrinsic units. mcmcpar and mcmcmap are the values and names of the parameters not held fixed during the fitting procedure and mcmcdim are the number of free parameters. Finally priordist is the prior distribution of each parameter that is not held fixed (see scipy.stats).

Parameters
parameters (*DICT*) – The dictionary describes several important features for each parameter of the model. A detailed description is given in the online documentation. In general, it contains a 'Value' and 'Unit', key, which are in physically interesting units. A 'Conversion' to intrinsic units of the cube. A 'Fixed' key to allow a parameter to be kept fixed, and finally a 'Dist' keyword, which describes the priors of the parameters (using the keys: 'Dloc and 'Dscale').

Returns
None

Return type
NoneType

run_mcmc(*nwalkers=50*, *nsteps=100*, *nproc=None*, *init_frac=0.02*, *filename=None*, *return_sampler=False*)

Run the MCMC process with emcee.

This method is the heart of QubeFit. It will run an MCMC process on a predefined model and will return the resulting chain of the posterior PDFs of each individual parameter that was varied. It saves the results in the mcmcarray and mcmclnprob data attributes, but it is HIGHLY RECOMMENDED to also save the outputs into a HDF5 file.

Parameters

- **nwalkers** (*INTEGER, optional*) – The number of walkers to use in the MCMC chain. The default is 50.
- **nsteps** (*INTEGER, optional*) – The number of steps to make in the MCMC chain. The default is 100.
- **nproc** (*INTEGER, optional*) – The number of parallel processes to use. If set to None, the code will determine the optimum number of processes to spawn. Set this number to limit to load of this code on your system. The default is None.

- **init_frac** (*FLOAT, optional*) – The fraction to use to randomize the initial distribution of walker away from the chosen initial value. The default is 0.02.
- **filename** (*STRING, optional*) – If set, the chain will be saved into a file with this file name. The file format is HDF5, and if not directly specified, this extension will be appended to the filename. The default is None.
- **return_sampler** (*BOOLEAN, optional*) – If set, this will return the emcee ensemble sampler. The default is False.

Raises

ValueError – A ValueError will be raised if the initial probability is infinity. This likely is an indication that the initial parameters fall outside the range defined by the priors.

Yields

sampler (*EMCEE.ENSEMBLESAMPLER*) – The ensemble sampler returned by the emcee function call can be returned, if wanted.

update_parameters(*parameters*)

Update the parameters key with the given parameters.

This method is a simple wrapper to update one or multiple parameters in the par data attribute. These parameters should be given in intrinsic units.

Parameters

parameters (*DICT*) – dictionary of parameters that will be read into the par keyword.

Returns

None

Return type

NoneType

4.11 Qfmodels

Models that can be used for Qubefit.

Any additional models should be put here and can then be referenced by the code. If you have a model that could be useful to others, please feel free to contact me, and I can add the model permanently to the list.

qubefit.qfmodels.**DispersionBulge**(***kwargs*)

Create a model of a dispersion-dominated bulge.

This will create a dispersion-dominated bulge model from the stored parameters specified in kwargs. The bulge model is described in detail in the online documentation and uses and limitations to this model are discussed there and in qubefit's reference paper.

Parameters

****kwargs** (*Dictionary*) – The kwargs dictionary contains all of the information to run the fitting procedure. It consists out of several nested dictionaries. The first level consists out the keys: {'mstring', 'mcmcmap', 'par', 'shape', 'data', 'kernel', 'variance', 'maskarray', 'initpar', 'probmethod', 'kernelarea', 'convolve'}. The kwargs dictionary should be generated directly from a fully populated qubefit instance using the function kwargs = self.__define_kwargs__(). This extra step is necessary for the input structure of emcee. To create the model, the important keys are:

'mstring': Dictionary

Contains the model name and profiles used for the model. That is the 'modelname', 'intensityprofile', 'velocityprofile', and 'dispersionprofile'. The first is a string with the model name,

the latter three are list of strings with the name of the profiles in each of the three dimensions, e.g., [‘Exponential’, None, ‘Exponential’].

’par’: Dictionary
Contains the parameters needed to successfully create the model. These are given in the online documentation, but in short they are ‘Xcen’, ‘Ycen’, ‘Rd’, ‘I0’, ‘Rv’, ‘Vcen’ and ‘Disp’. Each should be given in intrinsic units. In addition, some profiles (e.g., Sersic and Power) require an additional parameter, which is given by the optional parameters: ‘IIdx’ and ‘DIdx’.

’shape’: tuple
The shape of the array to be created.

’kernel’: np.ndarray
Array representation of the PSF. Will be used by astropy.convolve to convolve with the model.

’convolve’: Boolean
If set to true the model cube will be convolved with the kernel.

Returns
Model – Array of size kwargs[‘shape’] with the model that was generated from the parameters in kwargs[‘par’].

Return type
np.ndarray

qubefit.qfmodels.**ThinDisk**(***kwargs*)

Create a model of a thin disk.

This will create an infintely thin disk model from the stored parameters specified in kwargs. The thin disk model is described in detail in the online documentation and uses and limitations to this model are discussed there and in qubefit’s reference paper.

Parameters
****kwargs** (*Dictionary*) – The kwargs dictionary contains all of the information to run the fitting procedure. It consists out of several nested dictionaries. The first level consists out the keys: {‘mstring’, ‘mcmcmap’, ‘par’, ‘shape’, ‘data’, ‘kernel’, ‘variance’, ‘maskarray’, ‘initpar’, ‘probmethod’, ‘kernelarea’, ‘convolve’}. The kwargs dictionary should be generated directly from a fully populated qubefit instance using the function kwargs = self.__define_kwargs__(). This extra step is necessary for the input structure of emcee. To create the model, the important keys are:

’mstring’: Dictionary
Contains the model name and profiles used for the model. That is the ‘modelname’, ‘intensityprofile’, ‘velocityprofile’, and ‘dispersionprofile’. The first is a string with the model name, the latter three are list of strings with the name of the profiles in each of the three dimensions, e.g., [‘Exponential’, None, ‘Exponential’].

’par’: Dictionary
Contains the parameters needed to successfully create the model. These are given in the online documentation, but in short they are ‘Xcen’, ‘Ycen’, ‘PA’, ‘Incl’, ‘Rd’, ‘I0’, ‘Rv’, ‘Vmax’, ‘Vcen’, and ‘Disp’. Each should be given in intrinsic units. In addition, some profiles (e.g., Sersic and Power) require an additional parameter, which is given by the optional parameters: ‘IIdx’, ‘VIdx’, and ‘DIdx’.

’shape’: tuple
The shape of the array to be created.

’kernel’: np.ndarray
Array representation of the PSF. Will be used by astropy.convolve to convolve with the model.

'convolve': Boolean
If set to true the model cube will be convolved with the kernel.

Returns
Model – Array of size kwargs['shape'] with the model that was generated from the parameters in kwargs['par'].

Return type
np.ndarray

qubefit.qfmodels.**ThinSpiral**(***kwargs*)

Create a model of a thin spiral disk.

This will create an infintely thin disk model with a spiral pattern from the stored parameters specified in kwargs. The thin spiral model is described in detail in the online documentation and uses and limitations to this model are discussed there. It has been used in Chittidi et al. 2020 arXiv200513158

Parameters
****kwargs** (*Dictionary*) – The kwargs dictionary contains all of the information to run the fitting procedure. It consists out of several nested dictionaries. The first level consists out the keys: {'mstring', 'mcmcmap', 'par', 'shape', 'data', 'kernel', 'variance', 'maskarray', 'initpar', 'probmethod', 'kernelarea', 'convolve'}. The kwargs dictionary should be generated directly from a fully populated qubefit instance using the function kwargs = self.__define_kwargs__(). This extra step is necessary for the input structure of emcee. To create the model, the important keys are:

'mstring': Dictionary
Contains the model name and profiles used for the model. That is the 'modelname', 'intensityprofile', 'velocityprofile', and 'dispersionprofile'. The first is a string with the model name, the latter three are list of strings with the name of the profiles in each of the three dimensions, e.g., ['Exponential', None, 'Exponential'].

'par': Dictionary
Contains the parameters needed to successfully create the model. These are given in the online documentation, but in short they are 'Xcen', 'Ycen', 'PA', 'Incl', 'Rd', 'I0', 'Rv', 'Vmax', 'Vcen', and 'Disp'. Each should be given in intrinsic units. In addition, some profiles (e.g., Sersic and Power) require an additional parameter, which is given by the optional parameters: 'IIdx', 'VIdx', and 'DIdx'. The spiral pattern is described by an additonal six parameters, 'NSpiral', 'Phi0', 'Spcoef', 'Dphi', 'Ispf', and 'Rs'

'shape': tuple
The shape of the array to be created.

'kernel': np.ndarray
Array representation of the PSF. Will be used by astropy.convolve to convolve with the model.

'convolve': Boolean
If set to true the model cube will be convolved with the kernel.

Returns
Model – Array of size kwargs['shape'] with the model that was generated from the parameters in kwargs['par'].

Return type
np.ndarray

qubefit.qfmodels.**TwoClumps**(***kwargs*)

Create a model of two bulges.

This model combines two instances of bulges into a single model. This very basic model is useful to see if the velocity gradient seen in low spatial resolution data can be reproduced using a pair of merging clumps that are not rotating themselves.

Parameters

****kwargs** (*Dictionary*) – The kwargs dictionary contains all of the information to run the fitting procedure. It consists out of several nested dictionaries. The first level consists out the keys: {'mstring', 'mcmcmap', 'par', 'shape', 'data', 'kernel', 'variance', 'maskarray', 'initpar', 'probmethod', 'kernelarea', 'convolve'}. The kwargs dictionary should be generated directly from a fully populated qubefit instance using the function kwargs = self.__define_kwargs__(). This extra step is necessary for the input structure of emcee. To create the model, the important keys are:

'mstring': Dictionary
Contains the model name and profiles used for the model. That is the 'modelname', 'intensityprofile', 'velocityprofile', and 'dispersionprofile'. The first is a string with the model name, the latter three are list of strings with the name of the profiles in each of the three dimensions for **both** clumps, e.g., [['Exponential', None, 'None'], ['Exponential', None, 'None']]

'par': Dictionary
Contains the parameters needed to successfully create the model. These are given in the online documentation, but in short they are 'Xcen1', 'Ycen1', 'Rd1', 'I01', 'Rv1', 'Vcen1' and 'Disp1' for the first clump and similarly for the second clump, 'Xcen2', 'Ycen2', 'Rd2', 'I02', 'Rv2', 'Vcen2' and 'Disp2'. Each should be given in intrinsic units. In addition, some profiles (e.g., Sersic and Power) require an additional parameter, which is given by the optional parameters, 'IIdx1', 'DIdx1', 'IIdx2' and 'DIdx2'.

'shape': tuple
The shape of the array to be created.

'kernel': np.ndarray
Array representation of the PSF. Will be used by astropy.convolve to convolve with the model.

'convolve': Boolean
If set to true the model cube will be convolved with the kernel.

Returns

Model – Array of size kwargs['shape'] with the model that was generated from the parameters in kwargs['par'].

Return type

np.ndarray

PYTHON MODULE INDEX

INDEX