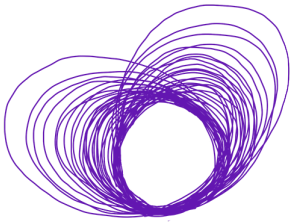
orbitize Documentation

Sarah Blunt, Jason Wang, Henry Ngo, et al.

Apr 16, 2024

CONTENTS



Hello world! Welcome to the documentation for `orbitize`, a Python package for fitting orbits of directly imaged planets.

`orbitize` packages two back-end algorithms into a consistent API. It's written to be fast, extensible, and easy-to-use. The tutorials below will walk you through the code and introduce some technical stuff, but we suggest learning about the Orbits for the Impatient (OFTI) algorithm and MCMC algorithms (we use this one) before diving in. Our contributor guidelines document will point you to more useful resources.

`orbitize` is designed to meet the needs of the exoplanet imaging community, and we encourage community involvement. If you find a bug, want to request a feature, etc. please create an issue on GitHub.

`orbitize` is patterned after and inspired by radvel.

CHAPTER

ONE

ATTRIBUTION:

- If you use `orbitize` in your work, please cite Blunt et al (2019).
- If you use the OFTI algorithm, please also cite Blunt et al (2017).
- If you use the Affine-invariant MCMC algorithm from `emcee`, please also cite Foreman-Mackey et al (2013).
- If you use the parallel-tempered Affine-invariant MCMC algorithm from `ptemcee`, please also cite Vousden et al (2016).
- If you use the Hipparcos intermediate astrometric data (IAD) fitting capability, please also cite Nielsen et al (2020) and van Leeuwen et al (2007).
- If you use Gaia data, please also cite Gaia Collaboration et al (2018; for DR2), or Gaia Collaboration et al (2021; for eDR3).

CHAPTER

TWO

USER GUIDE:

2.1 Installation

2.1.1 For Users

Parts of `orbitize` are written in C, so you'll need `gcc` (a C compiler) to install properly. Most Linux and Windows computers come with `gcc` built in, but Mac computers don't. If you haven't before, you'll need to download Xcode command line tools. There are several helpful guides online that teach you how to do this. Let us know if you have trouble!

`orbitize` is registered on `pip`, and works in Python>3.6. To install `orbitize`, first make sure you have the latest versions of `numpy` and `cython` installed. With `pip`, you can do this with the command:

```
$ pip install numpy cython --upgrade
```

Next, install `orbitize`:

```
$ pip install orbitize
```

We recommend installing and running `orbitize` in a `conda` virtual environment. Install `anaconda` or `miniconda` here, then see instructions here to learn more about `conda` virtual environments.

2.1.2 For Windows Users

Many of the packages that we use in `orbitize` were originally written for Linux or macOS. For that reason, we highly recommend installing the Windows Subsystem for Linux (WSL) which is an entire Linux development environment within Windows. See here for a handy getting started guide.

If you don't want to use WSL, there are a few extra steps you'll need to follow to get `orbitize` running:

1. There is a bug with the `ptemcee` installation that, as far as we know, only affects Windows users. To work around this, download `ptemcee` from its pypi page. Navigate to the root `ptemcee` folder, remove the `README.md` file, then install:

```
$ cd ptemcee
$ rm README.md
$ pip install . --upgrade
```

2. Some users have reported issues with installing `curses`. If this happens to you, you can install `windows-curses` which should work as a replacement.

```
$ pip install windows-curses
```

3. Finally, `rebound` is not compatible with windows, so you'll need to git clone orbitize, remove rebound from orbitize/requirements.txt, then install from the command line.

```
$ git clone https://github.com/sblunt/orbitize.git
$ cd orbitize
```

Open up orbitize/requirements.txt, remove `rebound`, and save.

```
$ pip install . --upgrade
```

2.1.3 For Developers

`orbitize` is actively being developed. The following method for installing `orbitize` will allow you to use it and make changes to it. After cloning the Git repository, run the following command in the top level of the repo:

```
$ pip install -r requirements.txt -e .
```

2.1.4 Issues?

If you run into any issues installing `orbitize`, please create an issue on GitHub.

If you are specifically having difficulties using `cython` to install `orbitize`, we suggest first trying to install `wheel`, then installing all of the `orbitize` dependencies (listed in `requirements.txt`).

If that doesn't work, we suggest disabling compilation of the C-based Kepler module with the following alternative installation command:

```
$ pip install orbitize --install-option="--disable-cython"
```

2.2 Tutorials

The following tutorials walk you through performing orbit fits with `orbitize`. To get started, read through "Formatting Input," "OFTI Introduction," and "MCMC Introduction." To learn more about the `orbitize` API, check out "Modifying Priors" and "Modifying MCMC Initial Positions." For an advanced plotting demo, see "Advanced Plotting," and to learn about the differences between OFTI and MCMC algorithms, we suggest "MCMC vs OFTI Comparison."

We also have a bunch of tutorials designed to introduce you to specific features of our code, listed below.

Many of these tutorials are also available as jupyter notebooks here.

If you find a bug, or if something is unclear, please create an issue on GitHub! We'd love any feedback on how to make `orbitize` more accessible.

A note about the tutorials: There are many ways to interact with the `orbitize` code base, and each person on our team uses the code differently. Each tutorial has a different author, and correspondingly a different style of using and explaining the code. If you are confused by part of one tutorial, we suggest looking at some of the others (and then contacting us if you are still confused).

2.2.1 Quick Start

This brief tutorial goes through the most minimal code you could write to do an orbit fit with `orbitize!`. It uses an input .csv that was placed on your computer when you installed `orbitize!`. The file lives here:

[1]:
```
import orbitize

path_to_file = "{}GJ504.csv".format(orbitize.DATADIR)

print(path_to_file)
```

```
/home/sblunt/Projects/orbitize/orbitize/example_data/GJ504.csv
```

The input .csv file looks like this:

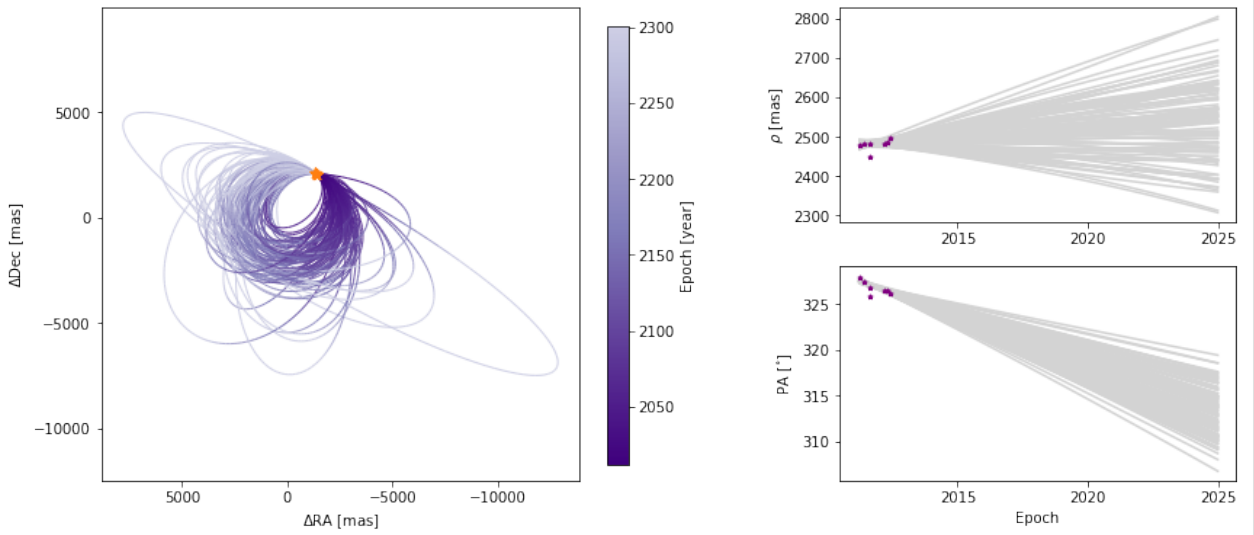
[2]:
```
from orbitize import read_input

read_input.read_file(path_to_file)
```

[2]:
```
<Table length=7>
    epoch      object  quant1 quant1_err ... quant12_corr quant_type instrument
   float64     int64  float64  float64   ...   float64      bytes5     bytes5
-------------- ------ ------- ---------- ... ------------ ---------- ----------
      55645.95      1  2479.0       16.0 ...          nan      seppa      defsp
      55702.89      1  2483.0        8.0 ...          nan      seppa      defsp
     55785.015      1  2481.0       33.0 ...          nan      seppa      defsp
     55787.935      1  2448.0       24.0 ...          nan      seppa      defsp
55985.19400184      1  2483.0       15.0 ...          nan      seppa      defsp
56029.11400323      1  2487.0        8.0 ...          nan      seppa      defsp
56072.30200459      1  2499.0       26.0 ...          nan      seppa      defsp
```

[3]:
```
%matplotlib inline

from orbitize import driver

myDriver = driver.Driver(
    '{}/GJ504.csv'.format(orbitize.DATADIR), # data file
    'OFTI',         # choose from: ['OFTI', 'MCMC']
    1,              # number of planets in system
    1.22,           # total mass [M_sun]
    56.95,          # system parallax [mas]
    mass_err=0.08, # mass error [M_sun]
    plx_err=0.26   # parallax error [mas]
)
orbits = myDriver.sampler.run_sampler(1000)

# plot the results
myResults = myDriver.sampler.results
orbit_figure = myResults.plot_orbits(
    start_mjd=myDriver.system.data_table['epoch'][0] # minimum MJD for colorbar (choose␣
↪first data epoch)
)
```

```
WARNING: ErfaWarning: ERFA function "d2dtf" yielded 1 of "dubious year (Note 5)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "dtf2d" yielded 1 of "dubious year (Note 6)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "utctai" yielded 1 of "dubious year (Note 3)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "taiutc" yielded 1 of "dubious year (Note 4)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "dtf2d" yielded 8 of "dubious year (Note 6)"␣
↪[astropy._erfa.core]
```

```
<Figure size 1008x432 with 0 Axes>
```



2.2.2 Formatting Input

Use `orbitize.read_input.read_file()` to read your astrometric data into orbitize. This method takes one argument, a string to the path of the file containing your input.

This method can read any file format supported by `astropy.io.ascii.read()`, including csv format. See the astropy docs.

There are two ways to provide input data to orbitize, either as observations or as an `orbitize!`-formatted input table.

Option 1

You can provide your observations in one of the following valid sets of measurements using the corresponding column names:

- RA and DEC offsets [milliarcseconds], using column names `raoff`, `raoff_err`, `decoff`, and `decoff_err`; or
- sep [milliarcseconds] and PA [degrees East of NCP], using column names `sep`, `sep_err`, `pa`, and `pa_err`; or
- RV measurement [km/s] using column names `rv` and `rv_err`.

Each row must also have a column for `epoch` and `object`. Epoch is the date of the observation, in MJD (JD-2400000.5). If this method thinks you have provided a date in JD, it will print a warning and attempt to convert to MJD. Objects are numbered with integers, where the primary/central object is `0`.

You may mix and match these three valid measurement formats in the same input file. So, you can have some epochs with RA/DEC offsets and others in separation/PA measurements.

If you have, for example, one RV measurement of a star and three astrometric measurements of an orbiting planet, you should put `0` in the `object` column for the RV point, and `1` in the columns for the astrometric measurements.

This method will look for columns with the above labels in whatever file format you choose so if you encounter errors, be sure to double check the column labels in your input file.

Putting it all together, here an example of a valid .csv input file:

```
epoch,object,raoff,raoff_err,decoff,decoff_err,radec_corr,sep,sep_err,pa,pa_err,rv,rv_err
1234,1,0.010,0.005,0.50,0.05,0.025,,,,,,
1235,1,,,,,,1.0,0.005,89.0,0.1,,
1236,1,,,,,,1.0,0.005,89.3,0.3,,
1237,0,,,,,,,,,,10,0.1
```

Note: Columns with no data can be omitted (e.g. if only separation and PA are given, the raoff, deoff, and rv columns can be excluded).

If more than one valid set is given (e.g. RV measurement and astrometric measurement taken at the same epoch), `read_file()` will generate a separate output row for each valid set.

Whatever file format you choose, this method will read your input into an `orbitize!`-formatted input table. This is an `astropy.Table.table` object that looks like this (for the example input given above):

```
epoch  object  quant1 quant1_err  quant2 quant2_err quant12_corr quant_type
float64  int   float64  float64   float64  float64     float64       str5
------- ------ ------- ---------- ------- ---------- ------------ ----------
1234.0      1    0.01      0.005     0.5       0.05      0.025        radec
1235.0      1     1.0      0.005    89.0        0.1        nan        seppa
1236.0      1     1.0      0.005    89.3        0.3        nan        seppa
1237.0      0    10.0        0.1     nan        nan        nan           rv
```

where `quant_type` is one of "radec", "seppa", or "rv".

If `quant_type` is "radec" or "seppa", the units of quant are mas and degrees, if `quant_type` is "rv", the units of quant are km/s.

Covariances

For RA/Dec and Sep/PA, you can optionally specify a correlation term. This is useful when your error ellipse is tilted with respect to the RA/Dec or Sep/PA. The correlation term is the Pearson correlation coefficient (), which corresponds to the normalized off diagonal term of the covariance matrix (C):

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{12} & C_{22} \end{bmatrix}.$$

Here C_11 = quant1_err^2 and C_22 = quant2_err^2 and C_12 is the off diagonal term (note that by definition both off-diagonal terms of the covariance matrix are the same). Then, $\rho = C_{12}/\sqrt{C_{11}C_{22}}$. Essentially it is the covariance normalized by the variance. As such, -1 1. You can specify either as radec_corr or seppa_corr to include a correlation

in the errors. By definition, both are dimensionless, but one will correspond to RA/Dec and the other to Sep/PA. If no correlations are specified, it will assume the errors are uncorrelated (= 0). In many papers, the errors are assumed to be uncorrelated. An example of real world data that reports correlations is this GRAVITY paper where table 2 reports the correlation values and figure 4 shows how the error ellipses are tilted.

In the example above, we specify the first epoch has a positive correlation between the uncertainties in RA and Dec using the `radec_corr` column in the input data. This gets translated into the `quant12_corr` field in `orbitize!`-format. No correlations are specified for the other entries, and so we will assume those errors are uncorrelated. After this is specified, handling of the correlations will be done automatically when computing model likelihoods. There's nothing else you have to do after this step!

Option 2

Alternatively, you can also supply a data file with the columns already corresponding to the `orbitize!`-formatted input table (see above for column names). This may be useful if you are wanting to use the output of the `write_orbitize_input` method (e.g. using some input prepared by another `orbitize!` user).

Note: When providing data with columns in the orbitize format, there should be no empty cells. As in the example below, when quant2 is not applicable, the cell should contain nan.

2.2.3 OFTI Introduction

by Isabel Angelo and Sarah Blunt (2018)

OFTI (Orbits For The Impatient) is an orbit-generating algorithm designed specifically to handle data covering short fractions of long-period exoplanets (Blunt et al. 2017). Here we go through steps of using OFTI within `orbitize!`

[1]:
```
import orbitize
```

Basic Orbit Generating

Orbits are generated in OFTI through a `Driver` class within orbitize. Once we have imported this class:

[2]:
```
import orbitize.driver
```

we can initialize a `Driver` object specific to our data:

[3]:
```
myDriver = orbitize.driver.Driver('{}/GJ504.csv'.format(orbitize.DATADIR), # path to␣
↪data file
                                  'OFTI', # name of algorithm for orbit-fitting
                                  1, # number of secondary bodies in system
                                  1.22, # total mass [M_sun]
                                  56.95, # total parallax of system [mas]
                                  mass_err=0.08, # mass error [M_sun]
                                  plx_err=0.26) # parallax error [mas]
```

Because OFTI is an object class within orbitize, we can assign all of the OFTI attributes onto a variable (`s`). We can then generate orbits for `s` using a function called `run_sampler`, a method of the `OFTI` class. The `run_sampler` method takes in the desired number of accepted orbits as an input.

Here we use run OFTI to randomly generate orbits until 1000 are accepted:

[4]:
```
s = myDriver.sampler
orbits = s.run_sampler(1000)
```

We have now generated 1000 possible orbits for our system. Here, `orbits` is a (1000 x 8) array, where each of the 1000 elements corresponds to a single orbit. An orbit is represented by 8 orbital elements.

Here is an example of what an accepted orbit looks like from orbitize:

[5]:
```
orbits[0]
```

[5]:
```
array([4.93916907e+01, 8.90197501e-03, 2.63925411e+00, 2.44962990e+00,
       9.31508665e-01, 1.20302112e-01, 5.74242058e+01, 1.22728974e+00])
```

To further inspect what each of the 8 elements in your orbit represents, you can view the system.param_idx variable. This is a dictionary that tells you the indices of your orbit that correspond to semi-major axis (a), eccentricity (e), inclination (i), argument of periastron (aop), position angle of nodes (pan), and epoch of periastron passage (epp). The last two indices are the parallax and system mass, and the number following the parameter name indicates the number of the body in the system.

[6]:
```
s.system.param_idx
```

[6]:
```
{'sma1': 0,
 'ecc1': 1,
 'inc1': 2,
 'aop1': 3,
 'pan1': 4,
 'tau1': 5,
 'plx': 6,
 'mtot': 7}
```

Plotting

Now that we can generate possible orbits for our system, we want to plot the data to interpret our results. Here we will go through a brief overview on ways to visualize your data within orbitize. For a more detailed guide on data visualization capabilities within orbitize, see the Orbitize plotting tutorial.

Histogram

One way to visualize our results is through histograms of our computed orbital parameters. Our orbits are outputted from `run_sampler` as an array of orbits, where each orbit is represented by a set of orbital elements:

[7]:
```
print(orbits.shape)
orbits[:5]
```

```
(1000, 8)
```

[7]:
```
array([[4.93916907e+01, 8.90197501e-03, 2.63925411e+00, 2.44962990e+00,
        9.31508665e-01, 1.20302112e-01, 5.74242058e+01, 1.22728974e+00],
       [4.69543031e+01, 1.31571508e-01, 2.52917998e+00, 1.34963602e+00,
        4.18692436e+00, 4.17659289e-01, 5.73207900e+01, 1.23162413e+00],
       [5.15848551e+01, 1.18074455e-01, 2.26110475e+00, 2.98346893e+00,
        2.31713931e+00, 3.94202277e-03, 5.69191065e+01, 1.15389146e+00],
```

(continues on next page)

(continued from previous page)

```
       [3.89558225e+01, 3.71357464e-01, 2.94801131e+00, 3.08542398e+00,
        4.77645562e+00, 7.15043369e-01, 5.72827098e+01, 1.05721164e+00],
       [8.19463988e+01, 1.45955646e-02, 2.11512811e+00, 4.52064036e+00,
        4.44802306e+00, 9.32004660e-01, 5.72429430e+01, 1.35323242e+00]])
```

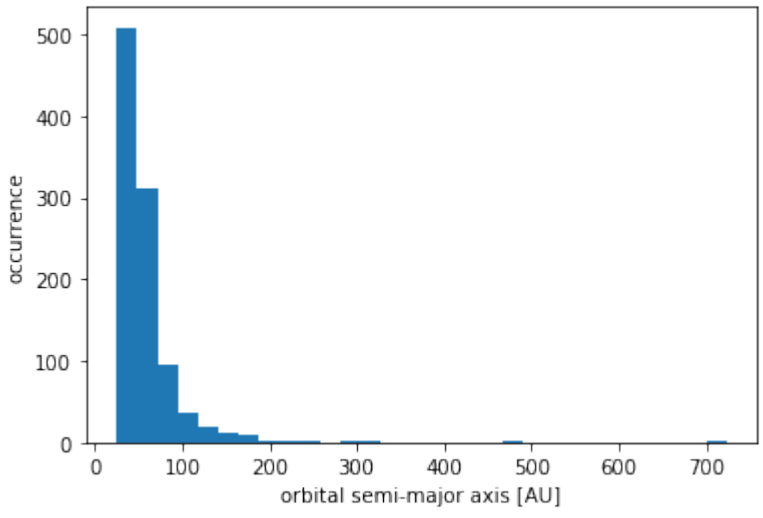
We can effectively view outputs from `run_sampler` by creating a histogram of a given orbit element to see its distribution of possible values. Our system.param_idx dictionary is useful here. We can use it to determine the index of a given orbit that corresponds to the orbital element we are interested in:

[8]:
```
s.system.param_idx
```

[8]:
```
{'sma1': 0,
 'ecc1': 1,
 'inc1': 2,
 'aop1': 3,
 'pan1': 4,
 'tau1': 5,
 'plx': 6,
 'mtot': 7}
```

If we want to plot the distribution of orbital semi-major axes (a) in our generated orbits, we would use the index dictionary `s.system.param_idx` to index the semi-major axis element from each orbit:

[9]:
```
sma = [x[s.system.param_idx['sma1']] for x in orbits]

%matplotlib inline
import matplotlib.pyplot as plt
plt.hist(sma, bins=30)
plt.xlabel('orbital semi-major axis [AU]')
plt.ylabel('occurrence')
plt.show()
```



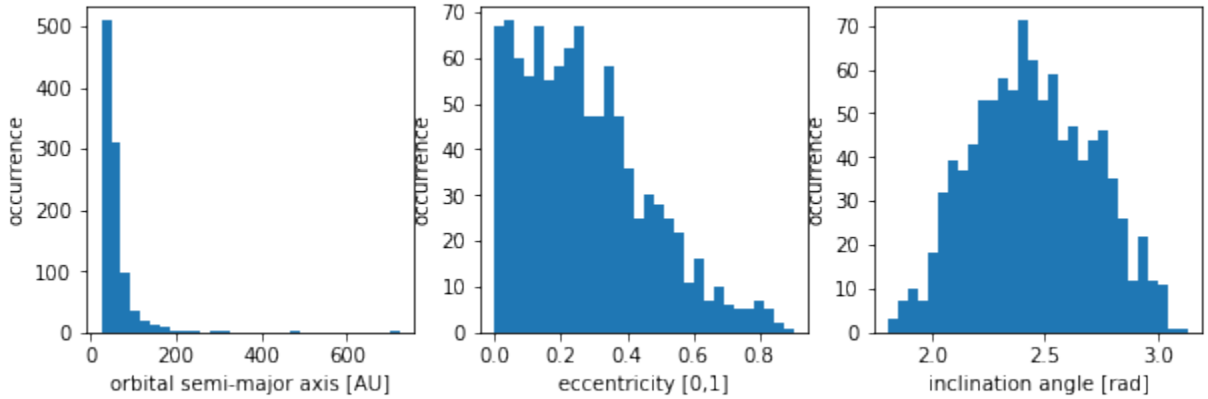
You can use this method to create histograms of any orbital element you are interested in:

[10]:
```
ecc = [x[s.system.param_idx['ecc1']] for x in orbits]
i = [x[s.system.param_idx['inc1']] for x in orbits]
```

(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(10,3))
plt.subplot(131)
plt.hist(sma, bins=30)
plt.xlabel('orbital semi-major axis [AU]')
plt.ylabel('occurrence')

plt.subplot(132)
plt.hist(ecc, bins=30)
plt.xlabel('eccentricity [0,1]')
plt.ylabel('occurrence')

plt.subplot(133)
plt.hist(i, bins=30)
plt.xlabel('inclination angle [rad]')
plt.ylabel('occurrence')

plt.show()
```



In addition to our `orbits` array, Orbitize also creates a `Results` class that contains built-in plotting capabilities for two types of plots: corner plots and orbit plots.

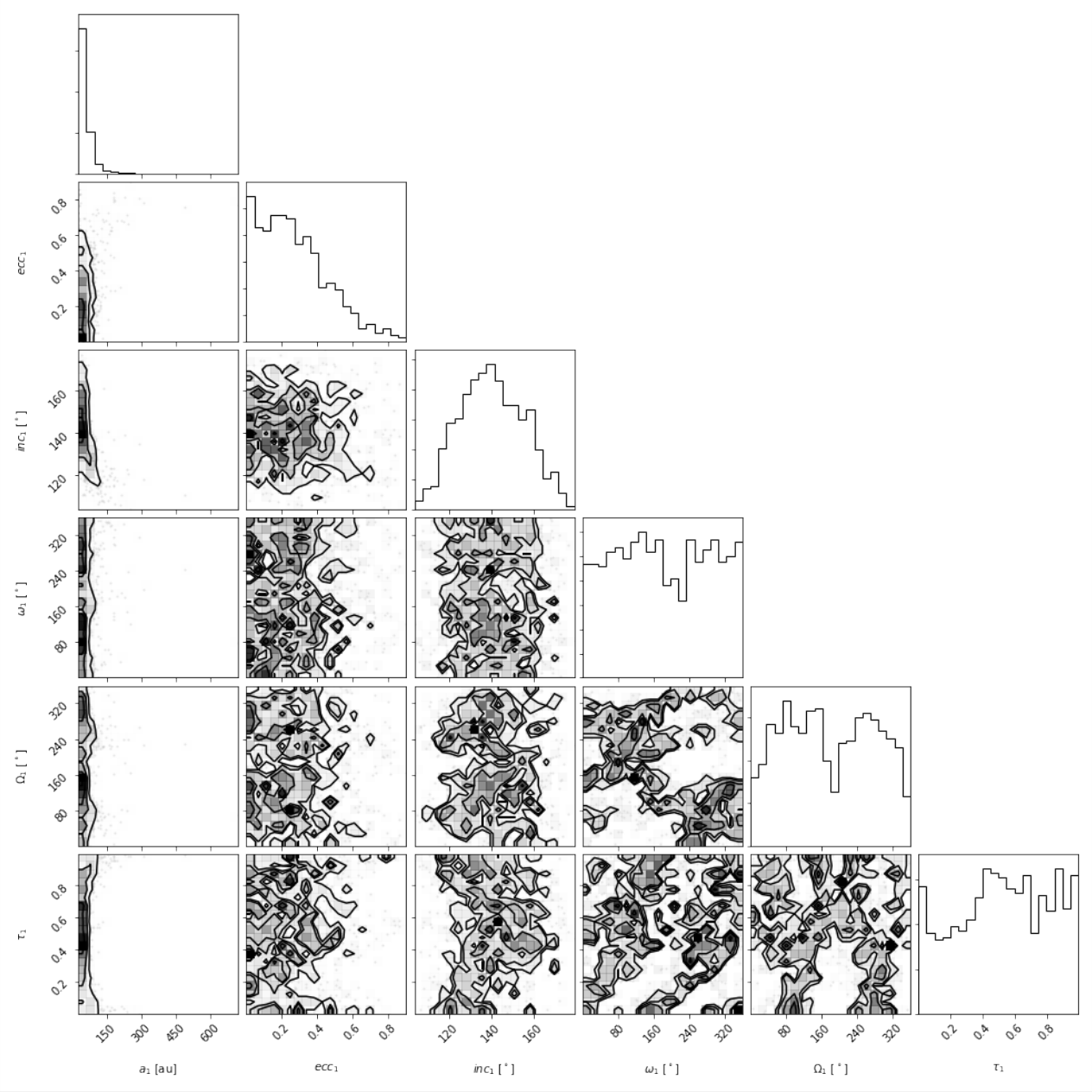
Corner Plot

After generating the samples, the `run_sampler` method also creates a `Results` object that can be accessed with `s.results`:

```
[11]: myResults = s.results
```

We can now create a corner plot using the function `plot_corner` within the `Results` class. This function requires an input list of the parameters, in string format, that you wish to include in your corner plot. We can even plot all of the orbital parameters at once! As shown below:

```
[12]: corner_figure = myResults.plot_corner(param_list=['sma1', 'ecc1', 'inc1', 'aop1', 'pan1',
      ↪'tau1'])
```



A Note about Convergence

Those of you with experience looking at corner plots will note that the result here does not look converged (i.e. we need more samples for our results to be statistically significant). Because this is a tutorial, we didn't want you to have to wait around for a while for the OFTI results to converge.

It's safe to say that OFTI should accept a minimum of 10,000 orbit for convergence. For pretty plots to go in publications, we recommend at least 1,000,000 accepted orbits.

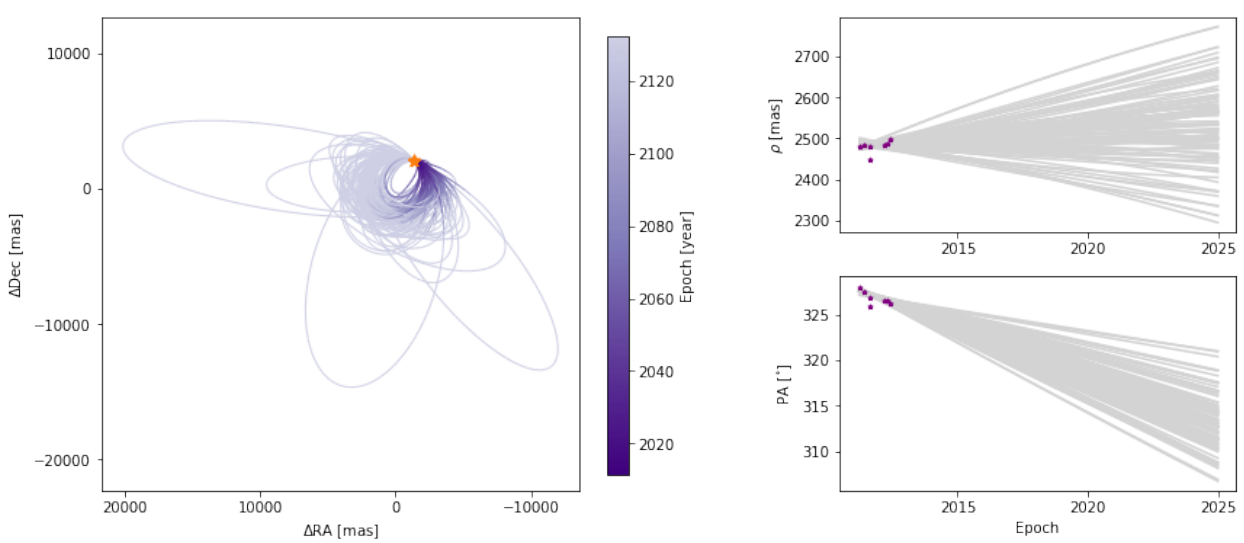
Orbit Plot

What about if we want to see how the orbits look in the sky? Don't worry, the `Results` class has a command for that too! It's called `plot_orbits`. We can create a simple orbit plot by running the command as follows:

[13]:
```
epochs = myDriver.system.data_table['epoch']

orbit_figure = myResults.plot_orbits(
    start_mjd=epochs[0] # Minimum MJD for colorbar (here we choose first data epoch)
)
```

```
WARNING: ErfaWarning: ERFA function "d2dtf" yielded 1 of "dubious year (Note 5)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "dtf2d" yielded 1 of "dubious year (Note 6)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "utctai" yielded 1 of "dubious year (Note 3)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "taiutc" yielded 1 of "dubious year (Note 4)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "dtf2d" yielded 8 of "dubious year (Note 6)"␣
↪[astropy._erfa.core]
```

```
<Figure size 1008x432 with 0 Axes>
```



Advanced OFTI and API Interaction

We've seen how the `run_sampler` command is the fastest way to generate orbits within OFTI. For users interested in what's going on under-the-hood, this part of the tutorial takes us each step of `run_sampler`. Understanding the intermediate stages of orbit-fitting can allow for more customization that goes beyond Orbitize's default parameters.

We begin again by intializing a `sampler` object on which we can run OFTI:

[14]:
```
myDriver = orbitize.driver.Driver('{}/GJ504.csv'.format(orbitize.DATADIR), # path to␣
↪data file
                                  'OFTI', # name of algorith for orbit-fitting
                                  1, # number of secondary bodies in system
```

(continues on next page)

(continued from previous page)

```
                                    1.22, # total mass [M_sun]
                                    56.95, # total parallax of system [mas]
                                    mass_err=0.08, # mass error [M_sun]
                                    plx_err=0.26) # parallax error [mas]
```

[15]:
```
s = myDriver.sampler
```

In orbitize, the first thing that OFTI does is prepare an initial set of possible orbits for our object through a function called `prepare_samples`, which takes in the number of orbits to generate as an input. For example, we can generate 100,000 orbits as follows:

[16]:
```
samples = s.prepare_samples(100000)
```

Here, `samples` is an array of randomly generated orbits that have been scaled-and-rotated to fit our astrometric observations. The first and second dimension of this array are the number of orbital elements and total orbits generated, respectively. In other words, each element in `samples` represents the value of a particular orbital element for each generated orbit:

[17]:
```
print('samples: ', samples.shape)
print('first element of samples: ', samples[0].shape)
```

```
samples:  (8, 100000)
first element of samples:  (100000,)
```

Once our initial set of orbits is generated, the orbits are vetted for likelihood in a function called `reject`. This function computes the probability of an orbit based on its associated chi squared. It then rejects orbits with lower likelihoods and accepts the orbits that are more probable. The output of this function is an array of possible orbits for our input system.

[18]:
```
orbits, lnlikes = s.reject(samples)
```

Our `orbits` array represents the final orbits that are output by OFTI. Each element in this array contains the 8 orbital elements that are computed by orbitize:

[19]:
```
orbits.shape
```

[19]:
```
(1, 8)
```

We can synthesize this sequence with the `run_sampler()` command, which runs through the steps above until the input number of orbits has been accepted. Additionally, we can specify the number of orbits generated by `prepare_samples` each time the sequence is initiated with an argument called `num_samples`. Higher values for `num_samples` will output more accepted orbits, but may take longer to run since all initially prepared orbits will be run through the rejection step.

[20]:
```
orbits = s.run_sampler(100, num_samples=1000)
```

Saving and Loading Results

Finally, we can save our generated orbits in a file that can be easily read for future use and analysis. Here we will walk through the steps of saving a set of orbits to a file in hdf5 format. The easiest way to do this is using `orbitize.Results.save_results()`:

[21]:
```
s.results.save_results('orbits.hdf5')
```

Now when you are ready to use your orbits data, it is easily accessible through the file we've created. One way to do this is to load the data into a new `results` object; in this way you can make use of the functions that we learn before, like `plot_corner` and `plot_orbits`. To do this, use the `results` module:

[22]:
```
import orbitize.results
loaded_results = orbitize.results.Results() # create a blank results object to load the␣
↪data
loaded_results.load_results('orbits.hdf5')
```

Alternatively, you can directly access the saved data using the `h5py` module:

[23]:
```
import h5py
f = h5py.File('orbits.hdf5', 'r')
orbits = f['post']

print('orbits array dimensions: ', orbits.shape)
print('orbital elements for first orbit: ', orbits[0])

f.close()
```

```
orbits array dimensions:  (100, 8)
orbital elements for first orbit:  [42.56737306  0.18520168  2.50497349  1.46225095  3.
↪29419715  0.60649612
 57.26589357  1.1478072 ]
```

And now we can easily work with the saved orbits that were generated by `orbitize!` Find out more about generating orbits in `orbitize!` with tutorials here.

2.2.4 MCMC Introduction

by Jason Wang and Henry Ngo (2018)

Here, we will explain how to sample an orbit posterior using MCMC techniques. MCMC samplers take some time to fully converge on the complex posterior, but should be able to explore all posteriors in roughly the same amount of time (unlike OFTI). We will use the parallel-tempered version of the Affine-invariant sample from the `ptemcee` package, as the parallel tempering helps the walkers get out of local minima. Parallel-tempering can be disabled by setting the number of temperatures to 1, and will revert back to using the regular ensemble sampler from emcee.

Read in Data and Set up Sampler

We use `orbitize.driver.Driver` to streamline the processes of reading in data, setting up the two-body interaction, and setting up the MCMC sampler.

When setting up the sampler, we need to decide how many temperatures and how many walkers per temperature to use. Increasing the number of temperatures further ensures your walkers will explore all of parameter space and will not get stuck in local minima. Increasing the number of walkers gives you more samples to use, and, for the Affine-invariant sampler, a minimum number is required for good convergence. Of course, the tradeoff is that more samplers means more computation time. We find 20 temperatures and 1000 walkers to be reliable for convergence. Since this is a tutorial meant to be run quickly, we use fewer walkers and temperatures here.

Note that we will only use the samples from the lowest-temperature walkers. We also assume that our astrometric measurements follow a Gaussian distribution.

`orbitize` can also fit for the total mass of the system and system parallax, including marginalizing over the uncertainties in those parameters.

[2]:
```
import numpy as np

import orbitize
from orbitize import driver
import multiprocessing as mp

filename = "{}/GJ504.csv".format(orbitize.DATADIR)

# system parameters
num_secondary_bodies = 1
total_mass = 1.75  # [Msol]
plx = 51.44  # [mas]
mass_err = 0.05  # [Msol]
plx_err = 0.12  # [mas]

# MCMC parameters
num_temps = 5
num_walkers = 20
num_threads = 2  # or a different number if you prefer, mp.cpu_count() for example


my_driver = driver.Driver(
    filename,
    "MCMC",
    num_secondary_bodies,
    total_mass,
    plx,
    mass_err=mass_err,
    plx_err=plx_err,
    mcmc_kwargs={
        "num_temps": num_temps,
        "num_walkers": num_walkers,
        "num_threads": num_threads,
    },
)
```

Running the MCMC Sampler

We need to pick how many steps the MCMC sampler should sample. Additionally, because the samples are correlated, we often only save every nth sample. This helps when we run a lot of samples, because saving all the samples requires too much disk space and many samples are unnecessary because they are correlated.

[3]:
```
total_orbits = 6000  # number of steps x number of walkers (at lowest temperature)
burn_steps = 10  # steps to burn in per walker
thin = 2  # only save every 2nd step

my_driver.sampler.run_sampler(total_orbits, burn_steps=burn_steps, thin=thin)
```

```
/home/sblunt/Projects/orbitize/orbitize/priors.py:354: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = -np.log((element_array*normalizer))
/home/sblunt/Projects/orbitize/orbitize/priors.py:463: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = np.log(np.sin(element_array)/normalization)
```

```
Starting Burn in
```

```
/home/sblunt/Projects/orbitize/orbitize/priors.py:354: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = -np.log((element_array*normalizer))
/home/sblunt/Projects/orbitize/orbitize/priors.py:463: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = np.log(np.sin(element_array)/normalization)
```

```
10/10 steps of burn-in complete
Burn in complete. Sampling posterior now.
300/300 steps completed
Run complete
```

[3]:
```
<ptemcee.sampler.Sampler at 0x7f5ef52163d0>
```

After completing the samples, the `'run_sampler'` method also creates a `'Results'` object that can be accessed with `'my_sampler.results'`.

MCMC Convergence

We want our walkers to be "converged" before they accurately sample the full posterior of our fitted parameters. Formal proofs of MCMC convergence are difficult or impossible in some cases. Many tests of convergence are necesary but not sufficient proofs of convergence. Here, we provide some convience functions to help assess convergence, but we caution they are not foolproof. A more detailed description of convergence analysis for affine-invariant samples (which are the ones used in `orbitize!`) is available *in the ``emcee`* docs <https://emcee.readthedocs.io/en/stable/tutorials/autocorr/>`__.

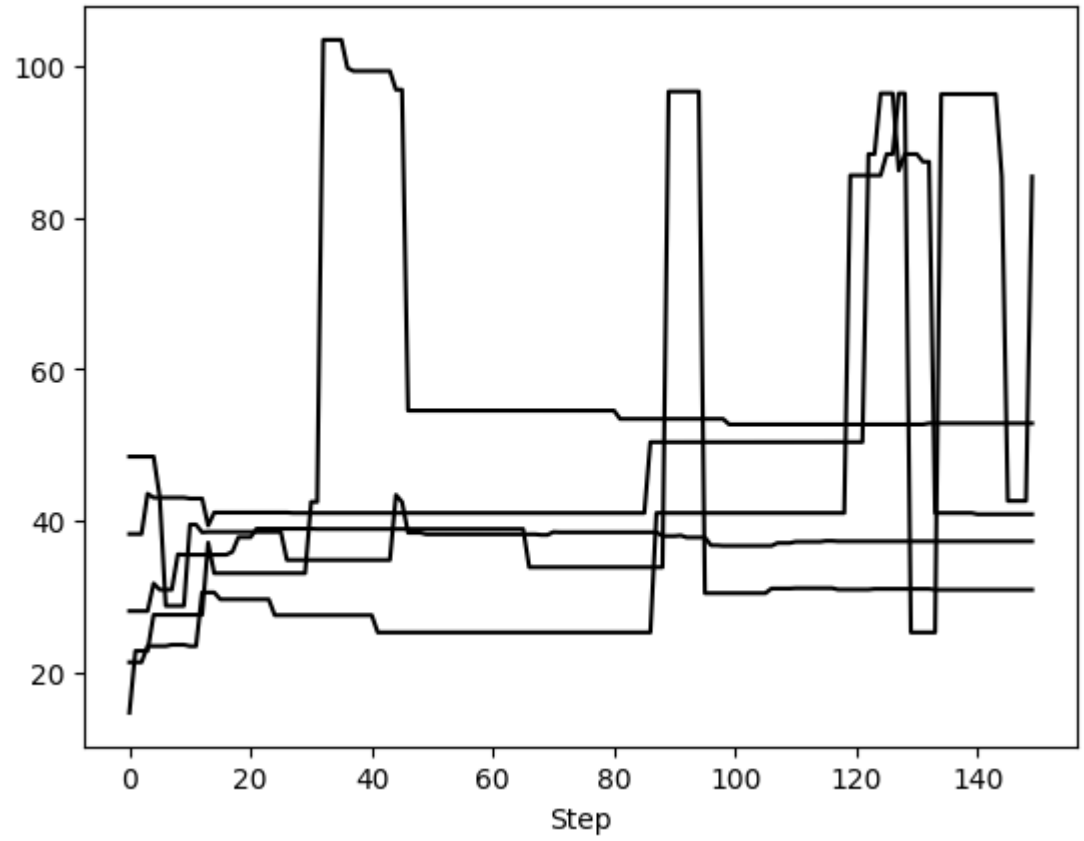
One of the primary ways we assess convergence for orbit fits for MCMC is the visual inspection of the chains. This is done by looking at some key parameters such as semi-major axis and eccentricity and plotting the value sampled for each chain as a function of step number. At the beginning, the ensemble of walkers is going to expand/contract/wiggle around to figure out where the posterior space is. Eventually, the walkers will appear to reach some "thermal equilibrium" beyond which the values sampled by the ensemble of walkers appear to not change with time. Below is how to use some built-in diagnostic functions for this.

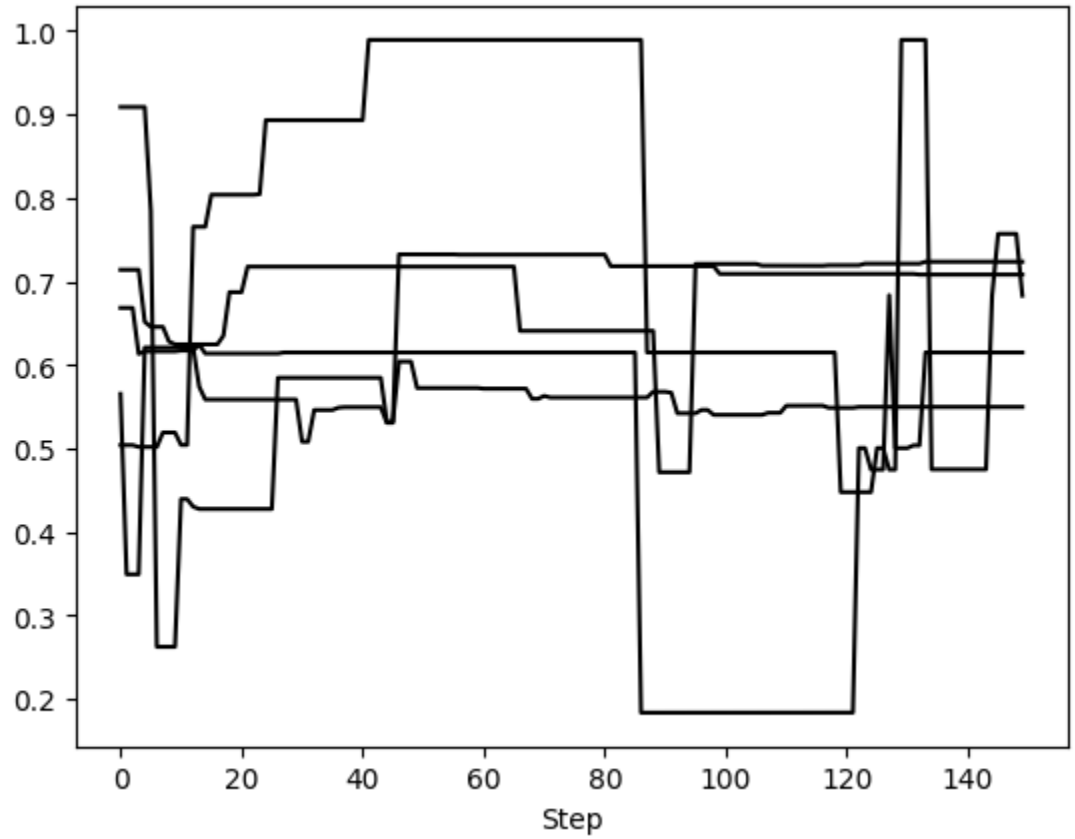
Diagnostic Functions

The `Sampler` object also has two convenience functions to examine and modify the walkers in order to diagnose MCMC performance. Note that in this example we have not run things for long enough with enough walkers to actually see convergence, so this is merely a demo of the API.

First, we can examine 5 randomly selected walkers for two parameters: semimajor axis and eccentricity. We expect 150 steps per walker since there were 6,000 orbits requested with 20 walkers, so that's 300 orbits per walker. However, we have thinned by a factor of 2, so there are 150 saved steps.

[4]:
```
sma_chains, ecc_chains = my_driver.sampler.examine_chains(
    param_list=["sma1", "ecc1"], n_walkers=5
)
```





This method returns one matplotlib `Figure` object for each parameter. If no `param_list` given, all parameters are returned. Here, we told it to plot 5 randomly selected walkers but we could have specified exactly which walkers with the `walker_list` keyword. The `step_range` keyword also determines which steps in the chain are plotted (when nothing is given, the default is to plot all steps). We can also have these plots automatically generate if we called `run_sampler` with `examine_chains=True`.

Note that this is just a convenience function. It is possible to recreate these chains from reshaping the posterior samples and selecting the correct entries.

The second diagnostic tool is the `chop_chains`, which allows us to remove entries from the beginning and/or end of a chain. This updates the corresponding `Results` object stored in `sampler` (in this case, `my_driver.sampler.results`). The `burn` parameter specifies the number of steps to remove from the beginning (i.e. to add a burn-in to your chain) and the `trim` parameter specifies the number of steps to remove from the end. If only one parameter is given, it is assumed to be a `burn` value. If `trim` is not zero, the `sampler` object is also updated so that the current position (`sampler.curr_pos`) matches the new end point. This allows us to continue MCMC runs at the correct position, even if we have removed the last few steps of the chain.

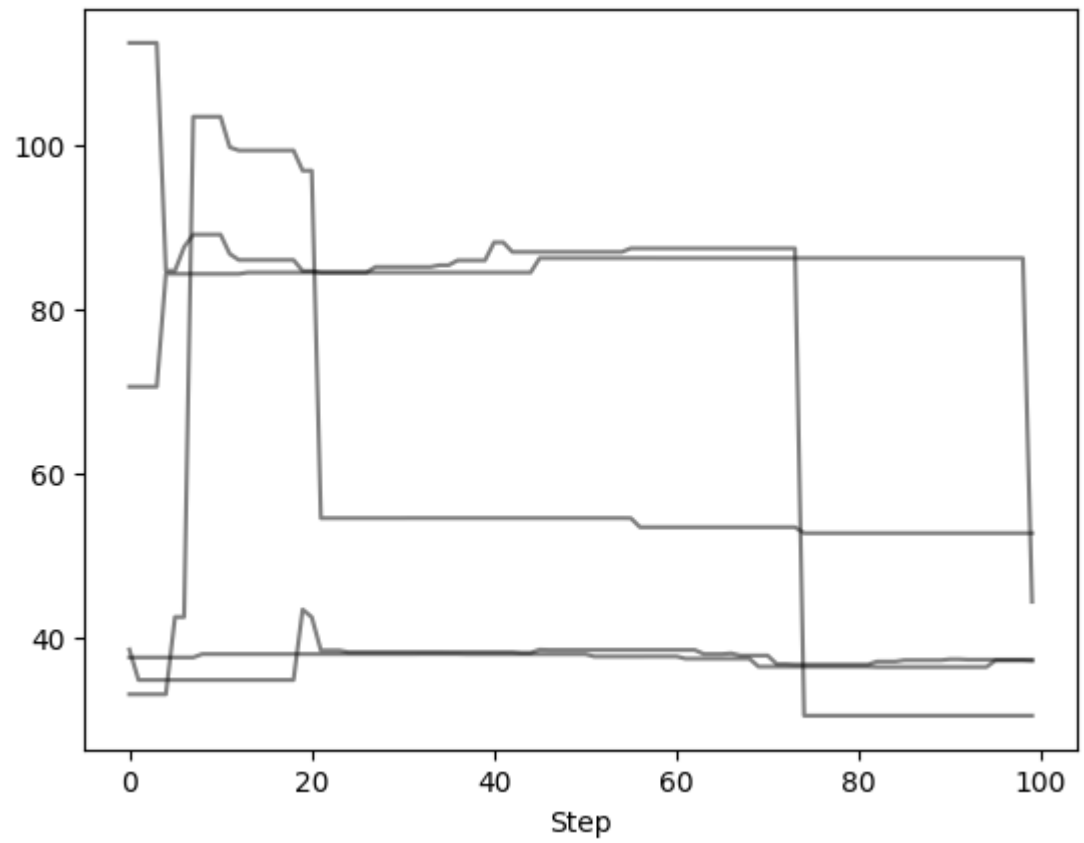
Let's remove the first and last 25 steps, leaving 100 orbits (or steps) per walker

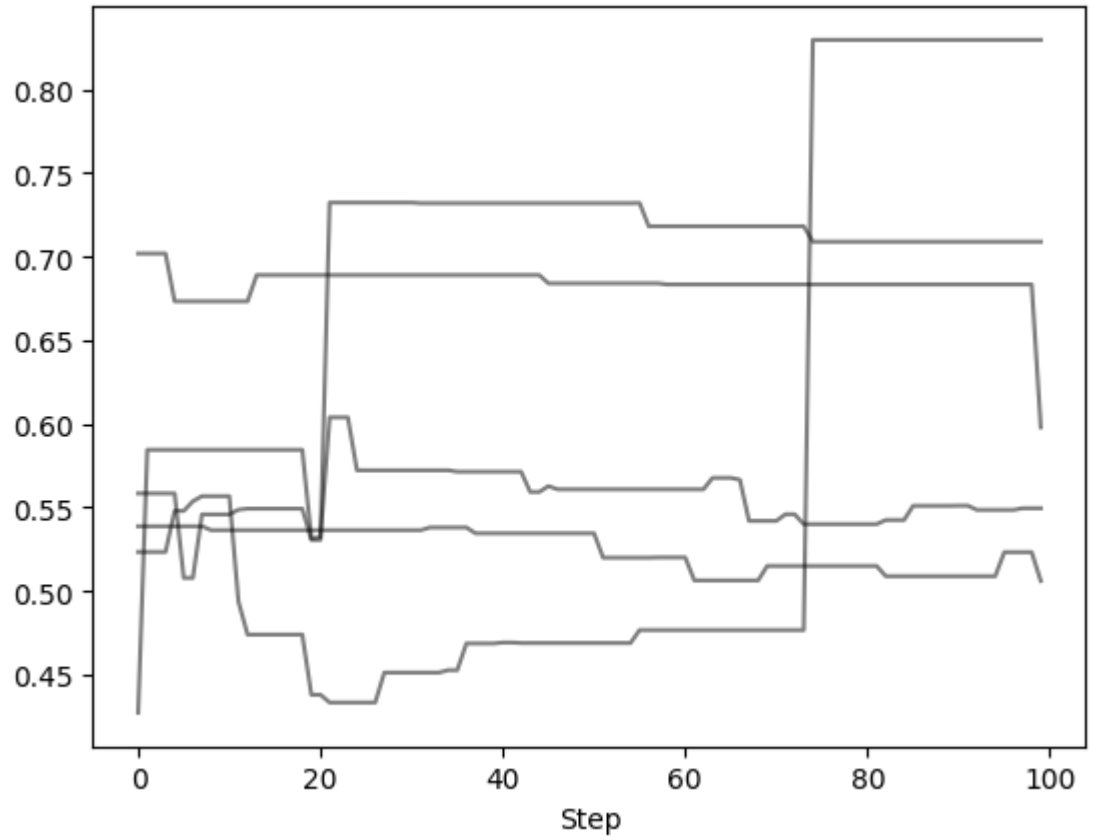
[5]:
```
my_driver.sampler.chop_chains(burn=25, trim=25)
```

```
Chains successfully chopped. Results object updated.
```

Now we can examine the chains again to verify what we did. Note that the number of steps removed from either end of the chain is uniform across all walkers.

[6]:
```
sma_chains, ecc_chains = my_driver.sampler.examine_chains(
    param_list=["sma1", "ecc1"], n_walkers=5, transparency=0.5
)
```





Autocorrelation Time Estimation

We can use the `emcee` package to estimate the autocorrelation time from our chains.

The integrated autocorrelation time (τ) for each parameter is returned, with an estimate of $\sim$ 10-15 steps needed for the chain to "forget" where it started. Notice an `AutocorrError` is returned since we have not run our chain long enough. Therefore, we should treat the integrated autocorrelation time as a lower bound, and run the MCMC for more steps. Here is a tutorial from `emcee` that estimates a more accurate integrated autocorrelation time when the chains are properly converged.

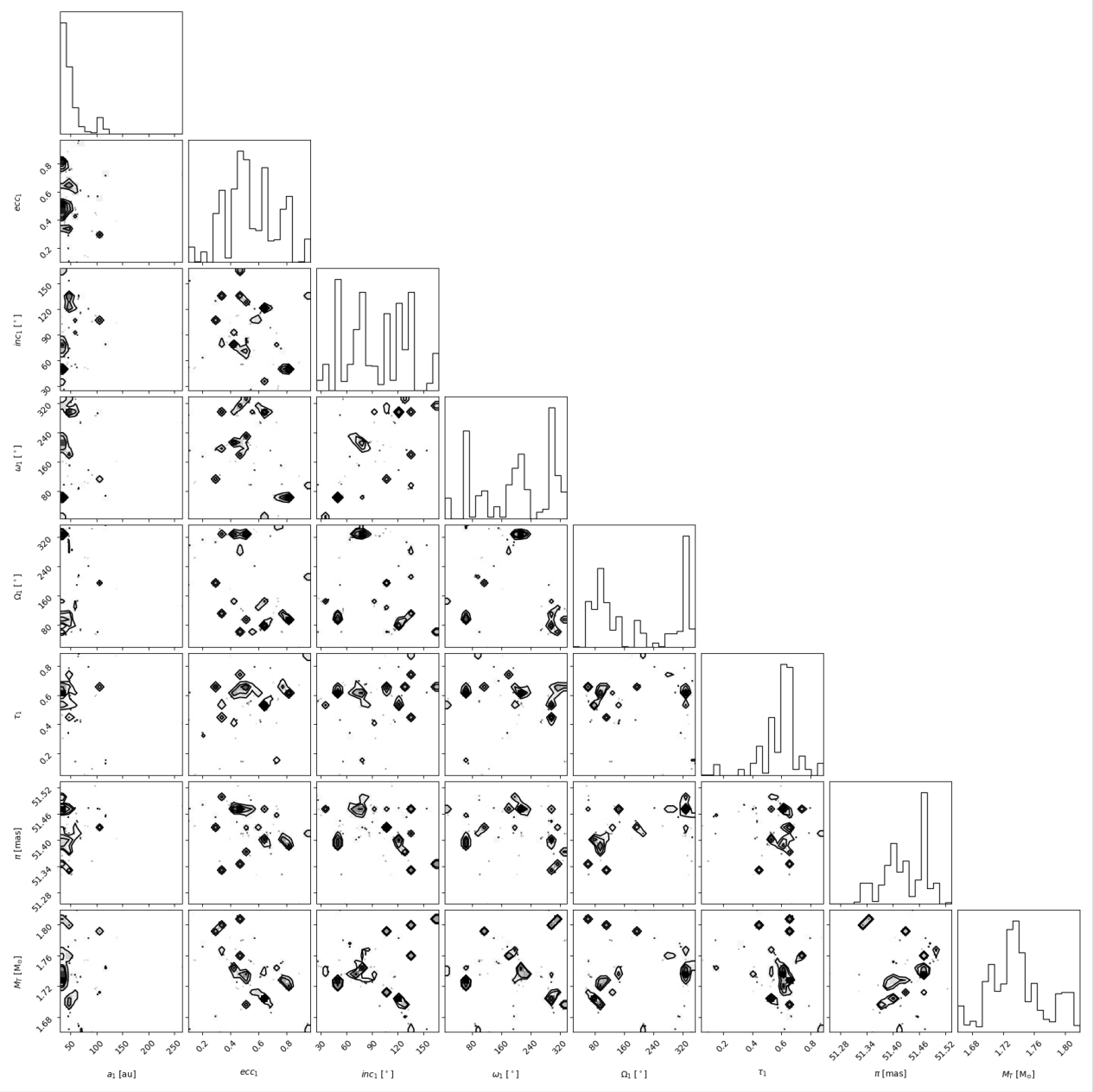
[ ]:
```
import emcee

flatchain = my_driver.sampler.post
total_samples, n_params = flatchain.shape
n_steps = int(total_samples / num_walkers)
chn = flatchain.reshape(num_walkers, n_steps, n_params)
# For emcee, reshape to (n_steps, num_walkers, n_params)
chn = np.transpose(chn, axes=(1, 0, 2))

try:
    tau = emcee.autocorr.integrated_time(chn)
except Exception as e:
    print("Exception was raised! The error message is: \n \n{}".format(e))
```

```
Exception was raised! The error message is:

 The chain is shorter than 50 times the integrated autocorrelation time for 8␣
↪parameter(s). Use this estimate with caution and run a longer chain!
N/50 = 3;
tau: [11.72814516 12.47675593 13.46492791 13.94748748 12.62859548 12.64879897
 12.94860228 13.10522437]
```

Plotting Basics

We will make some basic plots to visualize the samples in `'my_driver.sampler.results'`. Orbitize currently has two basic plotting functions which return matplotlib `Figure` objects. First, we can make a corner plot (also known as triangle plot, scatterplot matrix, or pairs plot) to visualize correlations between pairs of orbit parameters:

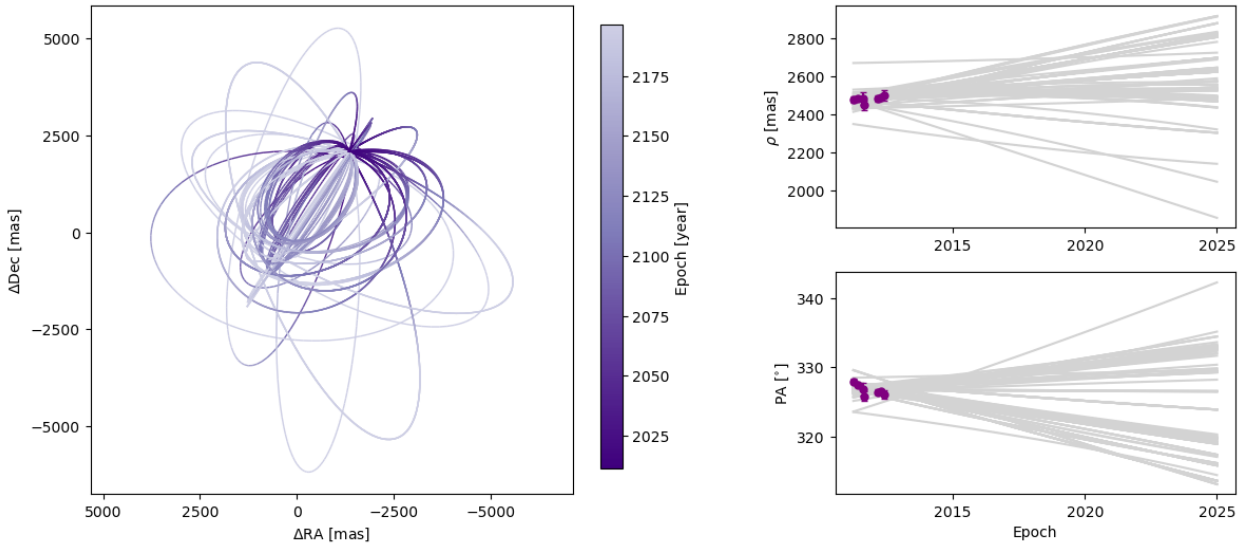
[7]:
```
corner_plot_fig = (
    my_driver.sampler.results.plot_corner()
)  # Creates a corner plot and returns Figure object
corner_plot_fig.savefig(
    "my_corner_plot.png"
)  # This is matplotlib.figure.Figure.savefig()
```



Next, we can plot a visualization of a selection of orbits sampled by our sampler. By default, the first epoch plotted is the year 2000 and 100 sampled orbits are displayed.

[8]:
```
epochs = my_driver.system.data_table["epoch"]

orbit_plot_fig = my_driver.sampler.results.plot_orbits(
    object_to_plot=1,  # Plot orbits for the first (and only, in this case) companion
    num_orbits_to_plot=100,  # Will plot 100 randomly selected orbits of this companion
    start_mjd=epochs[0],  # Minimum MJD for colorbar (here we choose first data epoch)
)
orbit_plot_fig.savefig(
    "my_orbit_plot.png"
)  # This is matplotlib.figure.Figure.savefig()
```

```
<Figure size 1400x600 with 0 Axes>
```



For more advanced plotting options and suggestions on what to do with the returned matplotlib Figure objects, see the dedicated Plotting tutorial.

Saving and Loading Results

We will save the results in the HDF5 format. It will save two datasets: `'post'` which will contain the posterior (the chains of the lowest temperature walkers) and `'lnlike'` which has the corresponding probabilities. In addition, it saves `'sampler_name'` and the orbitize version number as attributes of the HDF5 root group, and the `orbitize.system.System` object used to generate the orbit fit.

[9]:
```
hdf5_filename = "my_posterior.hdf5"
import os

# To avoid weird behaviours, delete saved file if it already exists from a previous run␣
↪of this notebook
if os.path.isfile(hdf5_filename):
    os.remove(hdf5_filename)
my_driver.sampler.results.save_results(hdf5_filename)
```

Saving sampler results is a good idea when we want to analyze the results in a different script or when we want to save the output of a long MCMC run to avoid having to re-run it in the future. We can then load the saved results into a new blank results object.

[10]:
```
from orbitize import results

loaded_results = results.Results()  # Create blank results object for loading
loaded_results.load_results(hdf5_filename)
```

Instead of loading results into an orbitize.results.Results object, we can also directly access the saved data using the `'h5py'` python module.

[11]:
```
import h5py
```

(continues on next page)

(continued from previous page)

```
filename = "my_posterior.hdf5"
hf = h5py.File(filename, "r")  # Opens file for reading
# Load up each dataset from hdf5 file
sampler_name = str(hf.attrs["sampler_name"])
post = np.array(hf.get("post"))
lnlike = np.array(hf.get("lnlike"))
hf.close()  # Don't forget to close the file
```

2.2.5 Modifying Priors

by Sarah Blunt (2018)

Most often, you will use the `Driver` class to interact with `orbitize`. This class automatically reads your input file, creates all of the `orbitize` objects you need to run an orbit fit, and allows you to run the orbit fit. See the introductory OFTI and MCMC tutorials for examples of working with this class.

However, sometimes you will want to work with the underlying methods directly. Doing this gives you control over the functionality `Driver` executes automatically, and allows you more flexibility.

Modifying priors is an example of something you might want to use the underlying API for. This tutorial walks you through how to do that.

Goals of this tutorial: - Learn to modify priors in `orbitize` - Learn how to fix a parameter at a specific value - Learn about the structure of the `orbitize` code base

[20]:
```
import numpy as np
from matplotlib import pyplot as plt
import orbitize
from orbitize import read_input, system, priors, sampler
```

Read in Data

First, let's read in our data table. This is accomplished with `orbitize.read_input`:

[21]:
```
data_table = read_input.read_file('{}/GJ504.csv'.format(orbitize.DATADIR))

print(data_table)
```

```
    epoch      object quant1 quant1_err quant2 quant2_err quant12_corr quant_type␣
↪instrument
-------------- ------ ------ ---------- ------ ---------- ------------ ---------- -------
↪---
      55645.95      1 2479.0       16.0 327.94       0.39          nan      seppa     ␣
↪defsp
      55702.89      1 2483.0        8.0 327.45       0.19          nan      seppa     ␣
↪defsp
     55785.015      1 2481.0       33.0 326.84       0.94          nan      seppa     ␣
↪defsp
     55787.935      1 2448.0       24.0 325.82       0.66          nan      seppa     ␣
↪defsp
55985.19400184      1 2483.0       15.0 326.46       0.36          nan      seppa     ␣
↪defsp
```

(continues on next page)

(continued from previous page)

```
56029.11400323      1 2487.0        8.0 326.54       0.18          nan       seppa      ␣
↪defsp
56072.30200459      1 2499.0       26.0 326.14       0.61          nan       seppa      ␣
↪defsp
```

Initialize `System` Object

Next, we initialize an `orbitize.system.System` object. This object stores information about the system you're fitting, such as your data, the total mass, and the parallax.

[22]:
```
# number of secondary bodies in system
num_planets = 1

# total mass & error [msol]
total_mass = 1.22
mass_err = 0.08

# parallax & error[mas]
plx = 56.95
plx_err = 0

sys = system.System(
    num_planets, data_table, total_mass,
    plx, mass_err=mass_err, plx_err=plx_err
)
```

The `System` object has a few handy attributes to help you keep track of your fitting parameters. `System.labels` is a list of the names of your fit parameters, and `System.sys_priors` is a list of the priors on each parameter. Notice that the "prior" on parallax (`plx`) is just a float. That's because we fixed this parameter at the printed value by specifying that `plx_err`=0.

Finally, `System.param_idx` is a dictionary that maps the parameter names from `System.labels` to their indices in `System.sys_priors`.

[23]:
```
print(sys.labels)
print(sys.sys_priors)
print(sys.param_idx)

# alias for convenience
lab = sys.param_idx
```

```
['sma1', 'ecc1', 'inc1', 'aop1', 'pan1', 'tau1', 'plx', 'mtot']
[Log Uniform, Uniform, Sine, Uniform, Uniform, Uniform, 56.95, Gaussian]
{'sma1': 0, 'ecc1': 1, 'inc1': 2, 'aop1': 3, 'pan1': 4, 'tau1': 5, 'plx': 6, 'mtot': 7}
```

Explore & Modify Priors

Priors in `orbitize` are Python objects. You can view an exhaustive list here. Let's print out the attributes of some of our priors:

[24]:
```
print(vars(sys.sys_priors[lab['ecc1']]))
print(vars(sys.sys_priors[lab['sma1']]))
```

```
{'minval': 0.0, 'maxval': 1.0}
{'minval': 0.001, 'maxval': 10000.0, 'logmin': -6.907755278982137, 'logmax': 9.
↪210340371976184}
```

Check out the priors documentation (linked above) for more info about the attributes of each of these priors.

Now that we understand how priors are represented and where they are stored, we can modify them! Here's an example of changing the prior on eccentricity from the current uniform prior to a Gaussian prior:

[32]:
```
mu = 0.2
sigma = 0.05

sys.sys_priors[lab['ecc1']] = priors.GaussianPrior(mu, sigma)

print(sys.labels)
print(sys.sys_priors)
print(vars(sys.sys_priors[lab['ecc1']]))
```

```
['sma1', 'ecc1', 'inc1', 'aop1', 'pan1', 'tau1', 'plx', 'mtot']
[Log Uniform, Gaussian, Sine, Uniform, Uniform, Uniform, 56.95, Gaussian]
{'mu': 0.2, 'sigma': 0.05, 'no_negatives': True}
```

Let's do one more example. Say we want to fix the inclination to a particular value (i.e. not allow it to vary in the fit at all), perhaps the known inclination value of a disk in the system. We can do that as follows:

[8]:
```
sys.sys_priors[lab['inc1']] = 2.5

print(sys.labels)
print(sys.sys_priors)
print('Inclination "prior:" {}'.format(sys.sys_priors[sys.param_idx['inc1']]))
print('Eccentricity prior: {}'.format(sys.sys_priors[sys.param_idx['ecc1']]))
```

```
['sma1', 'ecc1', 'inc1', 'aop1', 'pan1', 'tau1', 'plx', 'mtot']
[Log Uniform, Gaussian, 2.5, Uniform, Uniform, Uniform, 56.95, Gaussian]
Inclination "prior:" 2.5
Eccentricity prior: Gaussian
```

Run OFTI

All right! We're in business. To finish up, I'll demonstrate how to run an orbit fit with our modified `System` object, first with OFTI, then with MCMC.

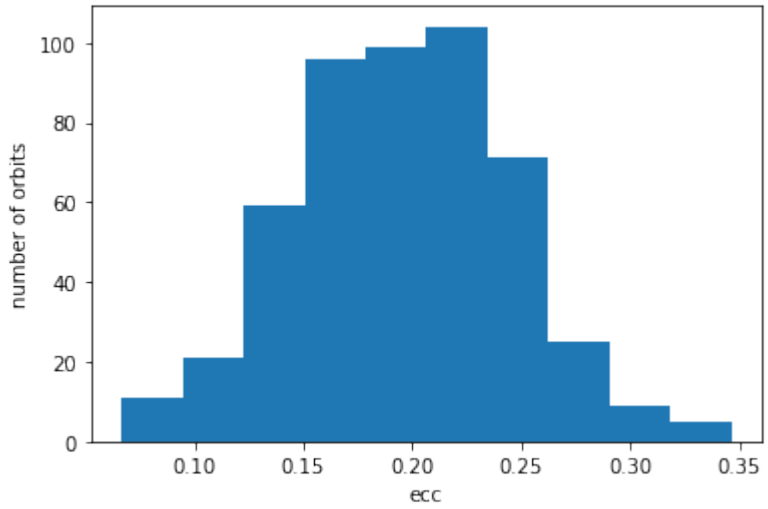
[8]:
```
ofti_sampler = sampler.OFTI(sys)

# number of orbits to accept
n_orbs = 500
```

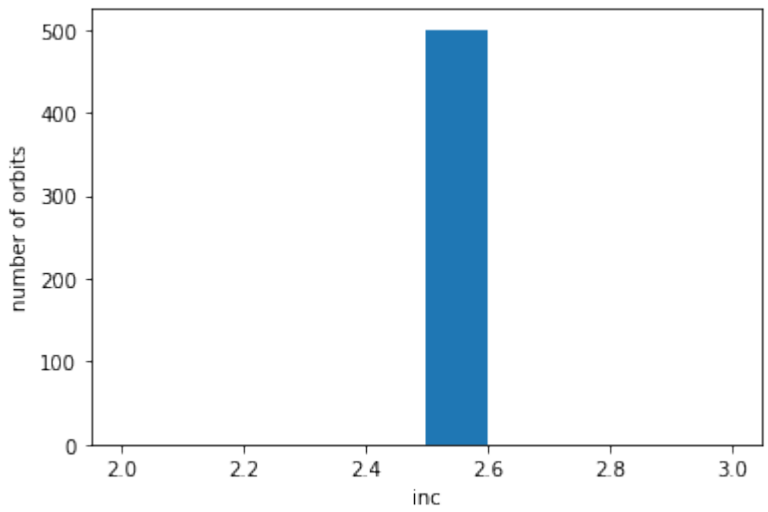
(continues on next page)

(continued from previous page)

```
_ = ofti_sampler.run_sampler(n_orbs)

plt.figure()
accepted_eccentricities = ofti_sampler.results.post[:, lab['ecc1']]
plt.hist(accepted_eccentricities)
plt.xlabel('ecc'); plt.ylabel('number of orbits')

plt.figure()
accepted_inclinations = ofti_sampler.results.post[:, lab['inc1']]
plt.hist(accepted_inclinations)
plt.xlabel('inc'); plt.ylabel('number of orbits')
```

[8]:
```
Text(0, 0.5, 'number of orbits')
```



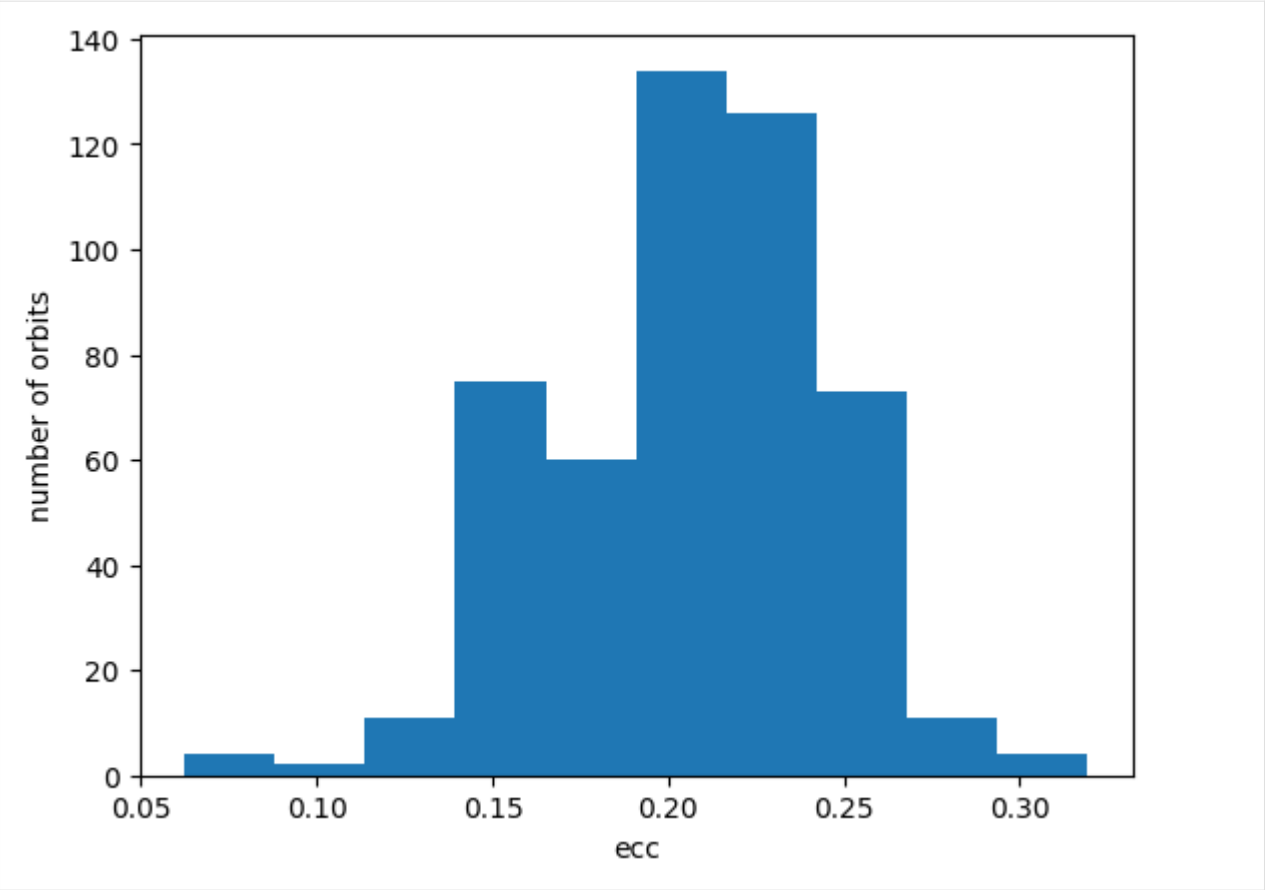


Run MCMC

[33]:
```
# number of temperatures & walkers for MCMC
num_temps = 3
num_walkers = 50

# number of steps to take
n_orbs = 500

mcmc_sampler = sampler.MCMC(sys, num_temps, num_walkers)

# number of orbits to accept
n_orbs = 500

_ = mcmc_sampler.run_sampler(n_orbs)

accepted_eccentricities = mcmc_sampler.results.post[:, lab['ecc1']]
plt.hist(accepted_eccentricities)
plt.xlabel('ecc'); plt.ylabel('number of orbits')
```

```
Starting Burn in

Burn in complete. Sampling posterior now.
10/10 steps completed
Run complete
```

[33]:
```
Text(0, 0.5, 'number of orbits')
```



The MCMC sampler will automatically take the priors from the system used to instantiate it. However, updates to the system priors made after the sampler instantiation will not be propogated to the sampler, and thus will not be used when running MCMC. The recommended practice when modifying priors is to first instantiate and modify system priors, then instantiate the sampler. When modifying priors we also recommend instantiating the system and sampler objects individually, rather than instantiating a driver object.

2.2.6 Advanced Plotting

by Henry Ngo (2018)

The `results.py` module contains several plotting functions to visualize the results of your `orbitize` orbit fit. Basic uses of these functions are covered in the OFTI and MCMC tutorials. Here, we will examine these plotting functions more deeply. This tutorial will be updated as more features are added to `orbitize`.

1. Test orbit generation with OFTI

In order to have sample data for this tutorial, we will use OFTI to generate some orbits for a published dataset on the GJ 504 system. The following code block is from the OFTI Tutorial , with 10000 orbits generated. Please see that tutorial for details.

Note: If you have already run this tutorial and saved the computed orbits, you may skip to Section 3 and load up your previously computed orbits instead of running this block below.

[1]:
```
import orbitize.driver

myDriver = orbitize.driver.Driver(
    "{}/GJ504.csv".format(orbitize.DATADIR),  # relative or absolute path to data file
    "OFTI",  # name of algorithm for orbit-fitting
    1,  # number of secondary bodies in system
    1.22,  # total mass [M_sun]
    56.95,  # parallax of system [mas]
    mass_err=0.08,  # mass error [M_sun]
    plx_err=0.26,  # parallax error [mas]
)
s = myDriver.sampler
orbits = s.run_sampler(1000)
```

```
1000/1000 orbits found
```

2. Accessing a Results object with computed orbits

After computing your orbits from either OFTI or MCMC, they are accessible as a `Results` object for further analysis and plotting. This object is an attribute of `s`, the `sampler` object defined above.

[2]:
```
myResults = (
    s.results
)  # array of MxN array of orbital parameters (M orbits with N parameters per orbit)
```

It is also useful to save this Results object to a file if we want to load up the same data later without re-computing the orbits.

[3]:
```
myResults.save_results("plotting_tutorial_GJ504_results.hdf5")
```

For more information on the Results object, see below.

[4]:
```
myResults?
```

```
Type:        Results
String form: <orbitize.results.Results object at 0x7fcd20f3c2e0>
File:        ~/Documents/GitHub/orbitize/orbitize/results.py
Docstring:
A class to store accepted orbital configurations from the sampler

Args:
    system (orbitize.system.System): System object used to do the fit.
    sampler_name (string): name of sampler class that generated these results
        (default: None).
    post (np.array of float): MxN array of orbital parameters
        (posterior output from orbit-fitting process), where M is the
        number of orbits generated, and N is the number of varying orbital
        parameters in the fit (default: None).
    lnlike (np.array of float): M array of log-likelihoods corresponding to
        the orbits described in ``post`` (default: None).
    version_number (str): version of orbitize that produced these results.
    data (astropy.table.Table): output from ``orbitize.read_input.read_file()``
    curr_pos (np.array of float): for MCMC only. A multi-D array of the
```

(continues on next page)

(continued from previous page)

```
        current walker positions that is used for restarting a MCMC sampler.

Written: Henry Ngo, Sarah Blunt, 2018

API Update: Sarah Blunt, 2021
```

Note that you can also add more computed orbits to a results object with `myResults.add_samples()`:

[5]:
```
myResults.add_samples?
```

```
Signature: myResults.add_samples(orbital_params, lnlikes, curr_pos=None)
Docstring:
Add accepted orbits, their likelihoods, and the orbitize version number
to the results

Args:
    orbital_params (np.array): add sets of orbital params (could be multiple)
        to results
    lnlike (np.array): add corresponding lnlike values to results
    curr_pos (np.array of float): for MCMC only. A multi-D array of the
        current walker positions

Written: Henry Ngo, 2018

API Update: Sarah Blunt, 2021
File:      ~/Documents/GitHub/orbitize/orbitize/results.py
Type:      method
```

3. (Optional) Load up saved results object

If you are skipping the generation of all orbits because you would rather load from a file that saved the Results object generated above, then execute this block to load it up. Otherwise, skip this block (however, nothing bad will happen if you run it even if you generated orbits above).

[6]:
```
import orbitize.results

if "myResults" in locals():
    del myResults  # delete existing Results object
myResults = orbitize.results.Results()  # create empty Results object
myResults.load_results("plotting_tutorial_GJ504_results.hdf5")  # load from file
```

4. Using our Results object to make plots

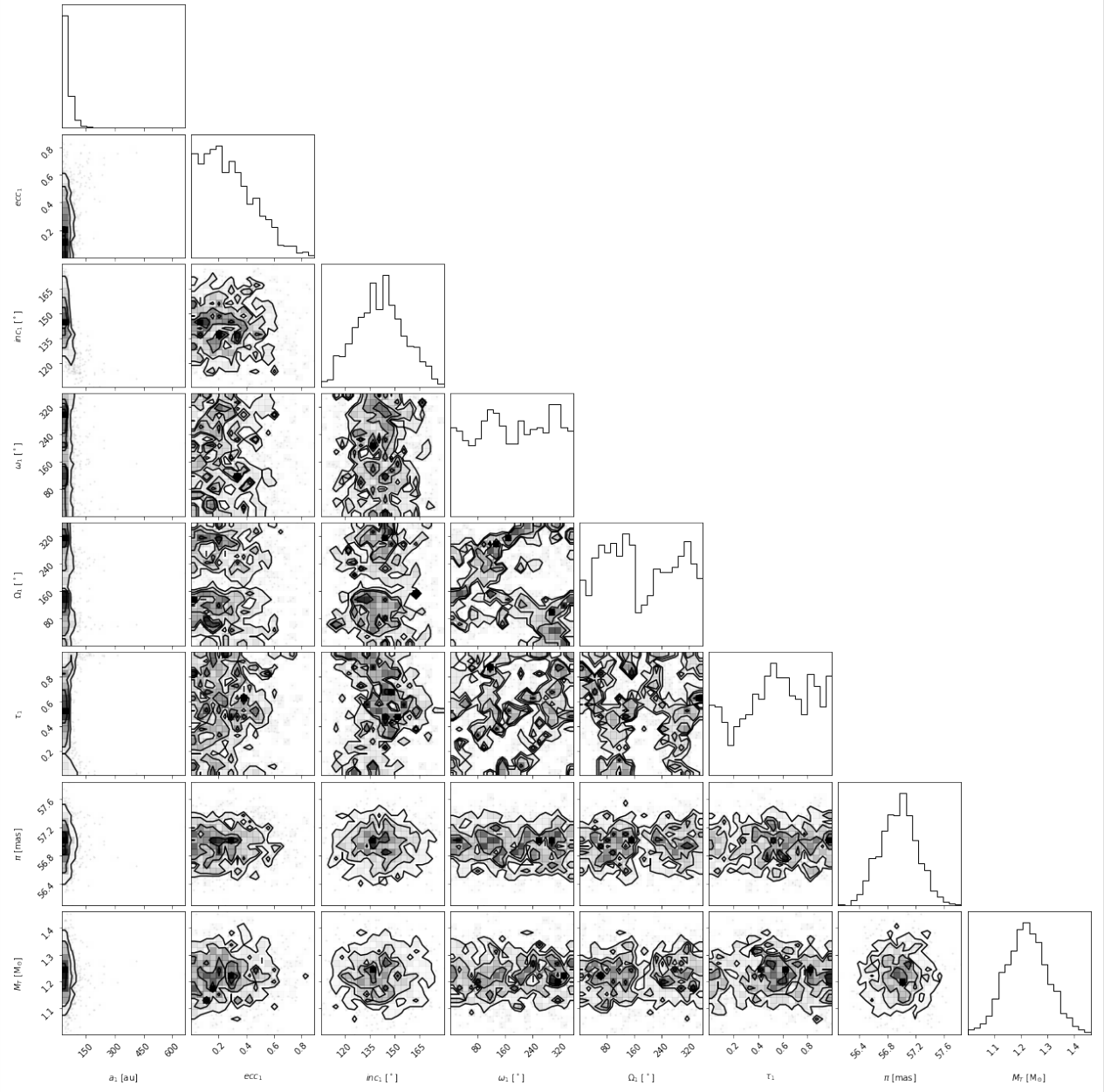
In this tutorial, we'll work with two plotting functions: `plot_corner()` makes a corner plot and `plot_orbits()` displays some or all of the computed orbits. Both plotting functions return matplotlib.pyplot.figure objects, which can be displayed, further manipulated with matplotlib.pyplot functions, and saved.

[7]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

4.1 Corner plots

This function is a wrapper for `corner.py` and creates a display of the 2-D covariances between each pair of parameters as well as histograms for each parameter. These plots are known as "corner plots", "pairs plots", and "scatterplot matrices", as well as other names.

[8]:
```
corner_figure = myResults.plot_corner()
```



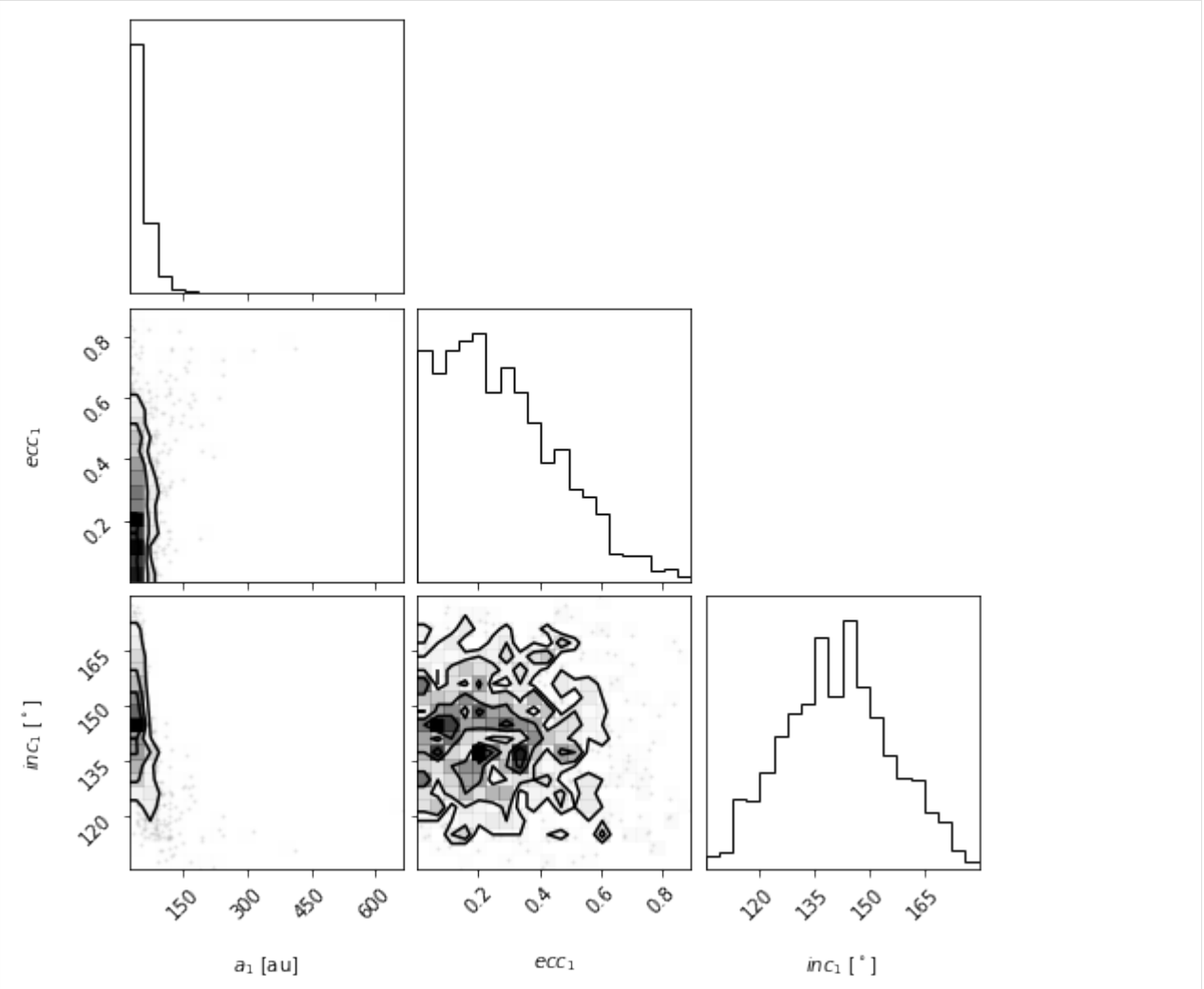
Choosing which parameters to plot

Sometimes, the full plot with all parameters is not what we want. Let's use the `param_list` keyword argument to plot only semimajor axis, eccentricity and inclination. Here are the possible string labels for this fit that you can enter for `param_list` and the corresponding orbit fit parameter:

Label	Parameter name
sma1	semimajor axis
ecc1	eccentricity
inc1	inclination
aop1	argument of periastron
pan1	position angle of nodes (aka longitude of ascending node)
tau1	epoch of periastron passage (expressed as a fraction of orbital period past a specified offset)
mtot	system mass
plx	system parallax

Note: for labels with numbers, the number corresponds to the companion (sma1 is the first object's semimajor axis, sma2 would be the second object, etc)

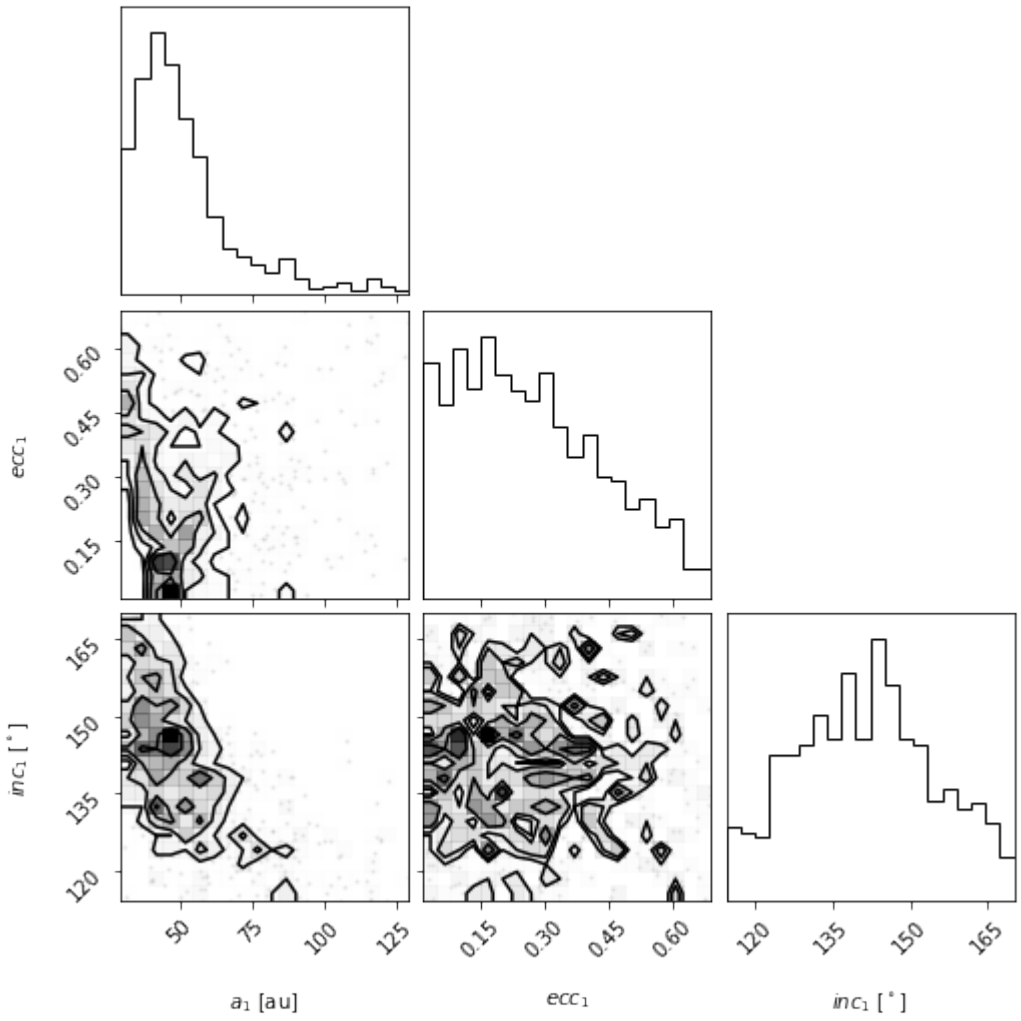
[9]:
```
corner_figure_aei = myResults.plot_corner(param_list=["sma1", "ecc1", "inc1"])
```



Limiting which samples to display

By picking out the panels we show, the plot can be easier to read. But in this case, we see that the plotted x-range on semimajor axis does show the posterior very well. This function will pass on all `corner.corner()` keywords as well. For example, we can use `corner`'s `range` keyword argument to limit the panels to only display 95% of the samples to avoid showing the long tails in the distribution.

[10]:
```
corner_figure_aei_95 = myResults.plot_corner(
    param_list=["sma1", "ecc1", "inc1"], range=(0.95, 0.95, 0.95)
)
```

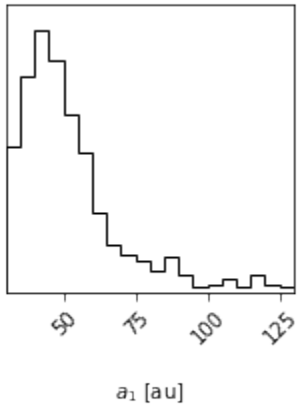


For other keywords you can pass to `corner`, see the `corner.py` API.

Making single variable histogram plots

One use of the `param_list` keyword is to just plot the histogram for the distribution of one single orbit fit parameter. We can do this by just providing one single parameter.

[11]:
```
histogram_figure_sma1 = myResults.plot_corner(param_list=["sma1"], range=(0.95,))
```

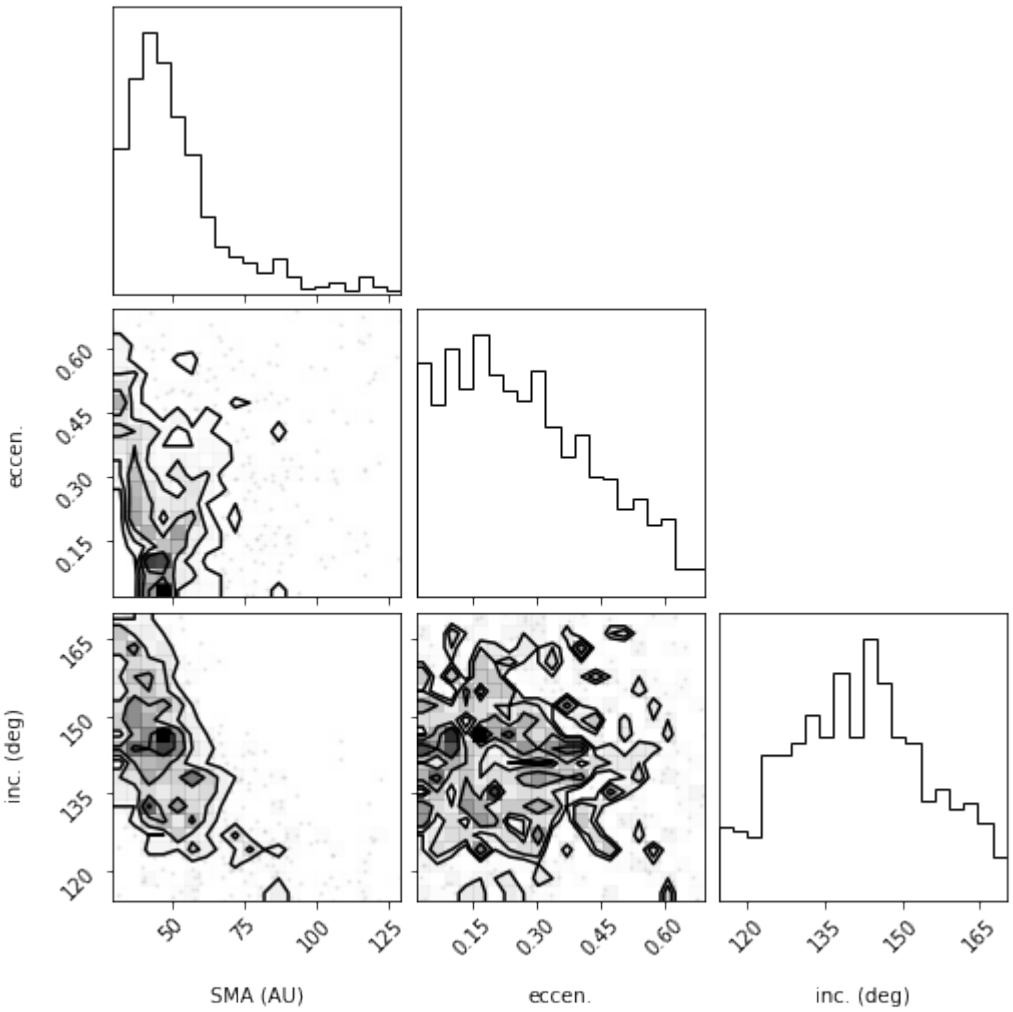


Axis label customization

The axis labels seen on the above plots are the default labels that `orbitize` passes to `corner.py` to make these plots. We can override these defaults by passing our own labels through the `labels` keyword parameter as per the `corner.py` API.

Note that the length of the list of `labels` should match the number of parameters plotted.

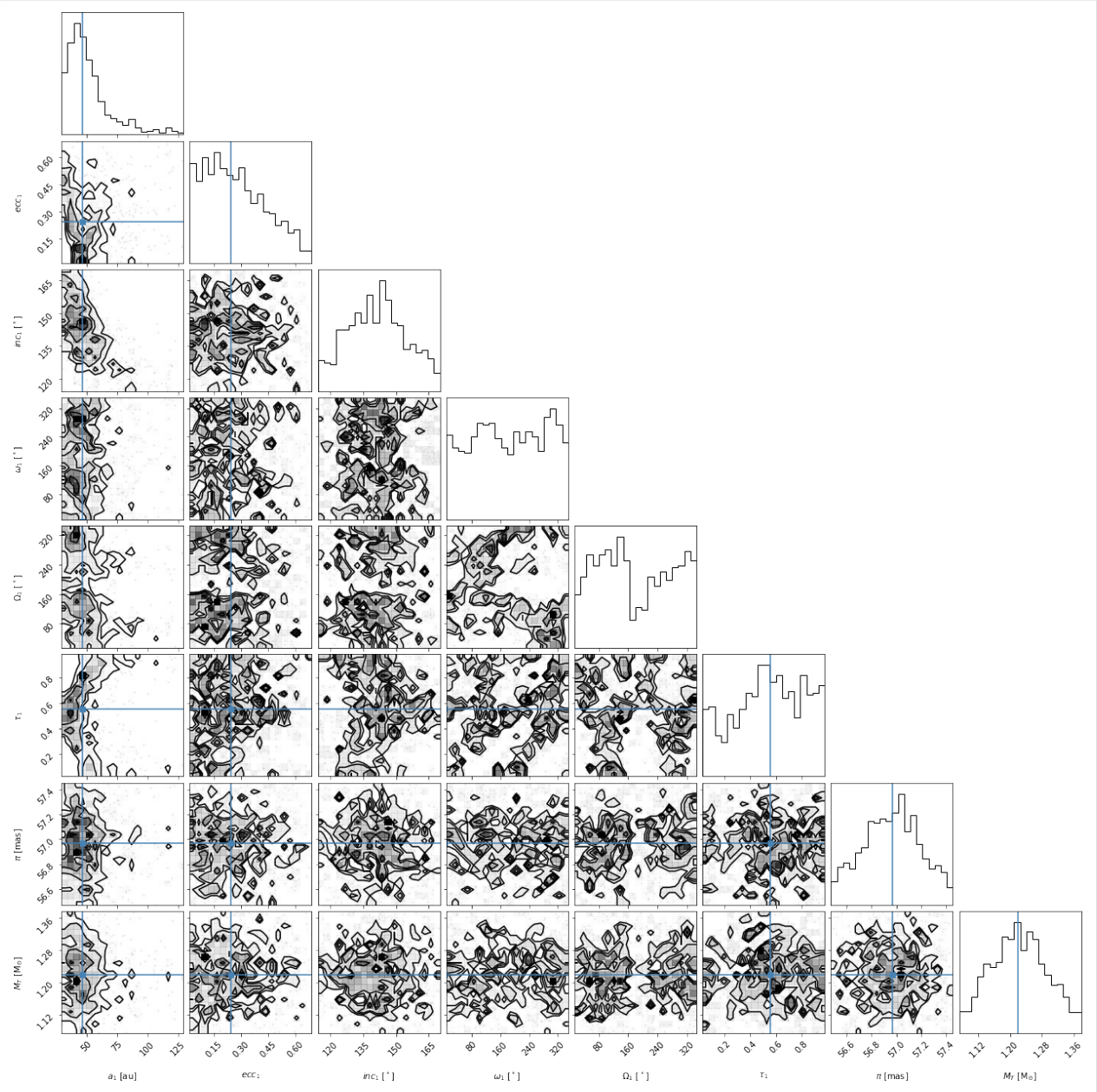
[12]:
```
# Corner plot with alternate labels
corner_figure_aei_95_labels = myResults.plot_corner(
    param_list=["sma1", "ecc1", "inc1"],
    range=(0.95, 0.95, 0.95),
    labels=("SMA (AU)", "eccen.", "inc. (deg)"),
)
```



Overplotting best fit ("truth") values

One feature of `corner.py` is to overplot the contours and histograms with a so-called "truth" value, which we can use for many purposes. For example, if we are sampling from a known distribution, we can use it to plot the true value to compare with our samples. Or, we can use it to mark the location of the mean or median of the distribution (the histogram and contours make it easy to pick out the most likely value or peaks of the distribution but maybe not the mean or median). Here is an example of overplotting the median on top of the distribution for the full corner plot.

[13]:
```
import numpy as np

median_values = np.median(myResults.post, axis=0)  # Compute median of each parameter
range_values = (
    np.ones_like(median_values) * 0.95
)  # Plot only 95% range for each parameter
corner_figure_median_95 = myResults.plot_corner(
    range=range_values, truths=median_values
)
```



Overall, we find that some of the parameters have converged well but others are not well constrained by our fit. As mentioned above, the output of the `plot_corner()` methods are matplotlib Figure objects. So, if we wanted to save this figure, we might use the `savefig()` method.

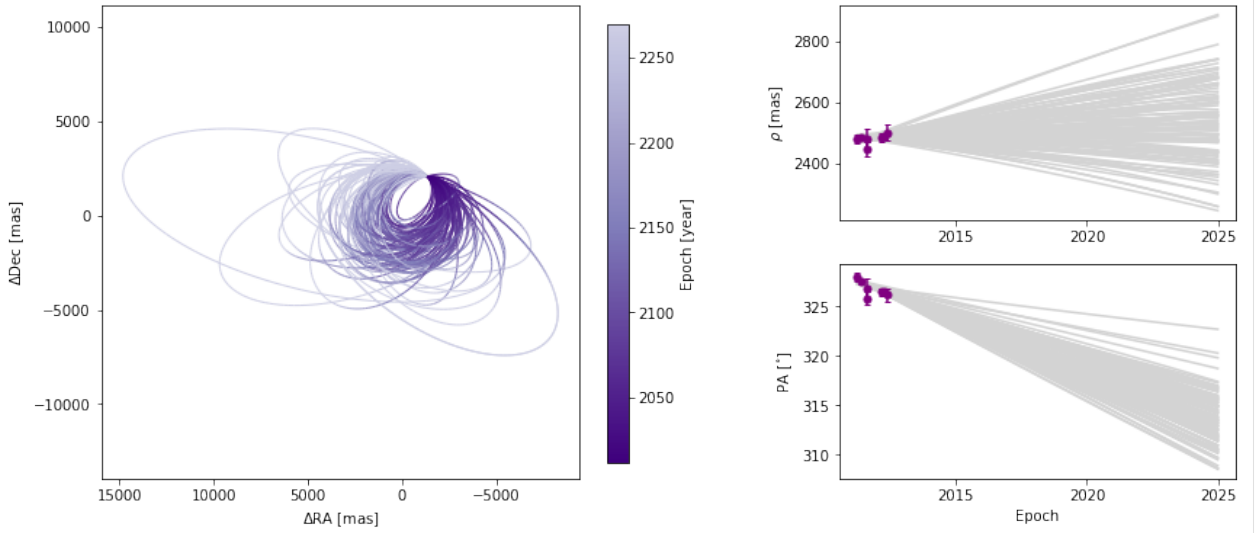
[14]:
```
corner_figure_median_95.savefig("plotting_tutorial_corner_figure_example.png")
```

4.2 Orbit Plot

The `plot_orbits` method in the `Results` module allows us to visualize the orbits sampled by `orbitize!`. The default call to `plot_orbits` will draw 100 orbits randomly chosen out of the total orbits sampled (set by parameter `num_orbits_to_plot`). In addition, to draw each of these orbits, by default, we will sample each orbit at 100 evenly spaced points in time throughout the orbit's orbital period (set by parameter `num_epochs_to_plot`). These points are then connected by coloured line segments corresponding to the date where the object would be at that point in the orbit. By default, orbits are plotted starting in the year 2000 (set by parameter `start_mjd`) and are drawn for one complete orbital period. We usually choose to begin plotting orbits at the first data epoch, using this keyword as illustrated below.

[15]:
```
epochs = myDriver.system.data_table["epoch"]

orbit_figure = myResults.plot_orbits(
    start_mjd=epochs[0]  # Minimum MJD for colorbar (here we choose first data epoch)
)
orbit_figure.savefig("example_orbit_figure.png", dpi=250)
```

```
<Figure size 1008x432 with 0 Axes>
```



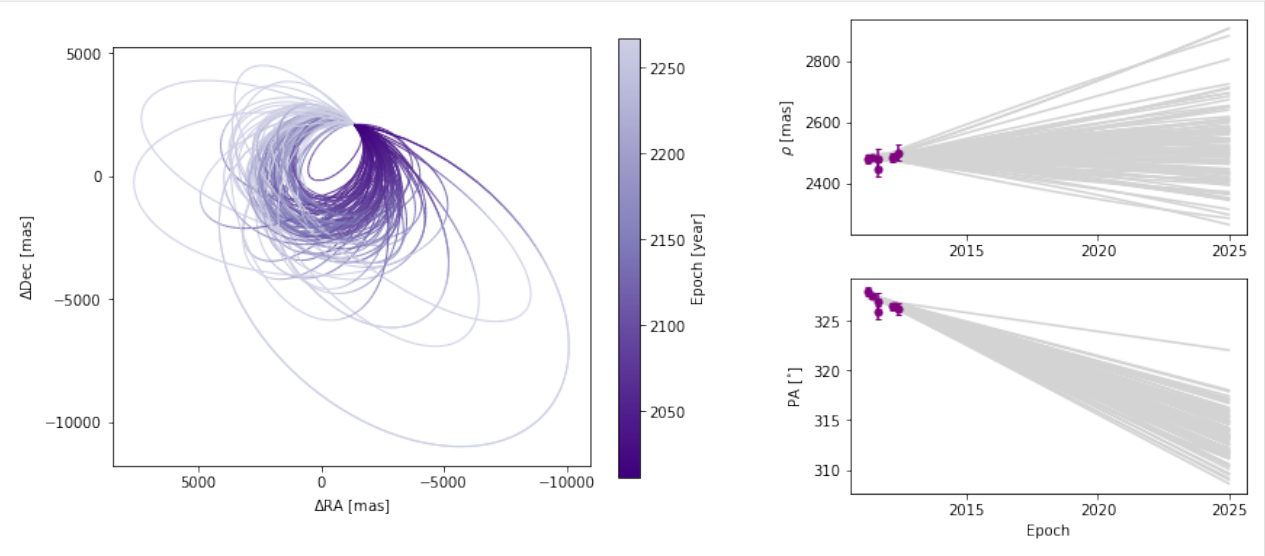
Customizing orbit plot appearence

In the above figure, we see that the x and y axes are RA and Dec offsets from the primary star in milliarcseconds. By default the axes aspect ratio is square, but we can turn that off. This is normally not recommended but for some situations, it may be desirable.

(Note: Each call to `plot_orbits` selects a different random subset of orbits to plot, so the orbits shown in the figures here are not exactly the same each time).

[16]:
```
orbit_figure_non_square = myResults.plot_orbits(start_mjd=epochs[0], square_plot=False)
```

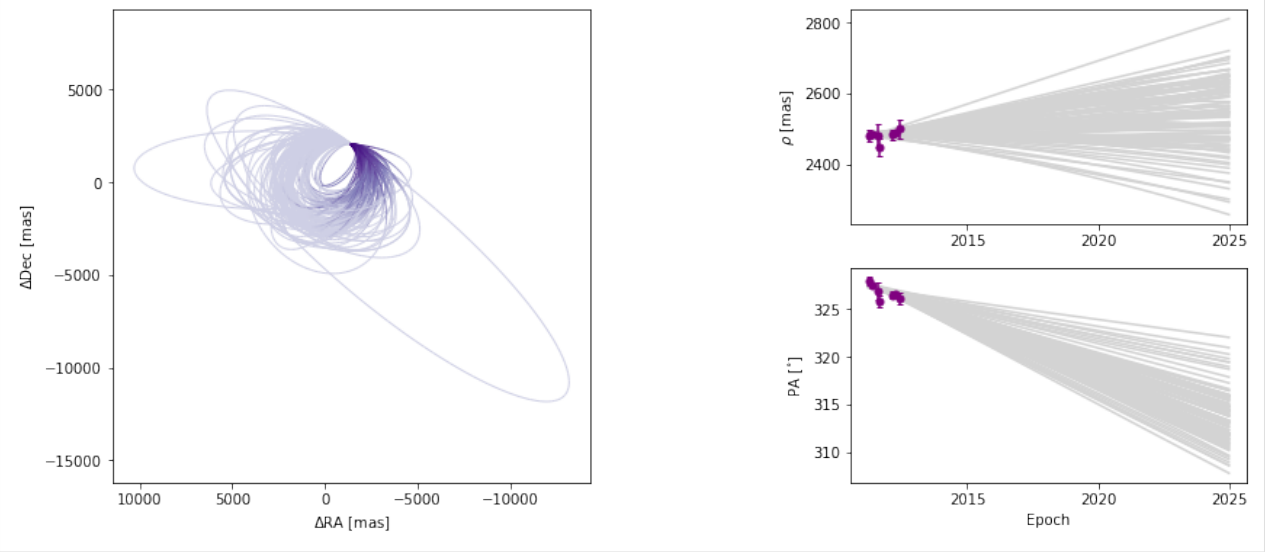
```
<Figure size 1008x432 with 0 Axes>
```



The colorbar shows how the line segment colors correspond to the date, beginning at the first data epoch. We can also turn off the colourbar.

[17]:
```
orbit_figure_no_colorbar = myResults.plot_orbits(
    start_mjd=epochs[0], show_colorbar=False
)
```

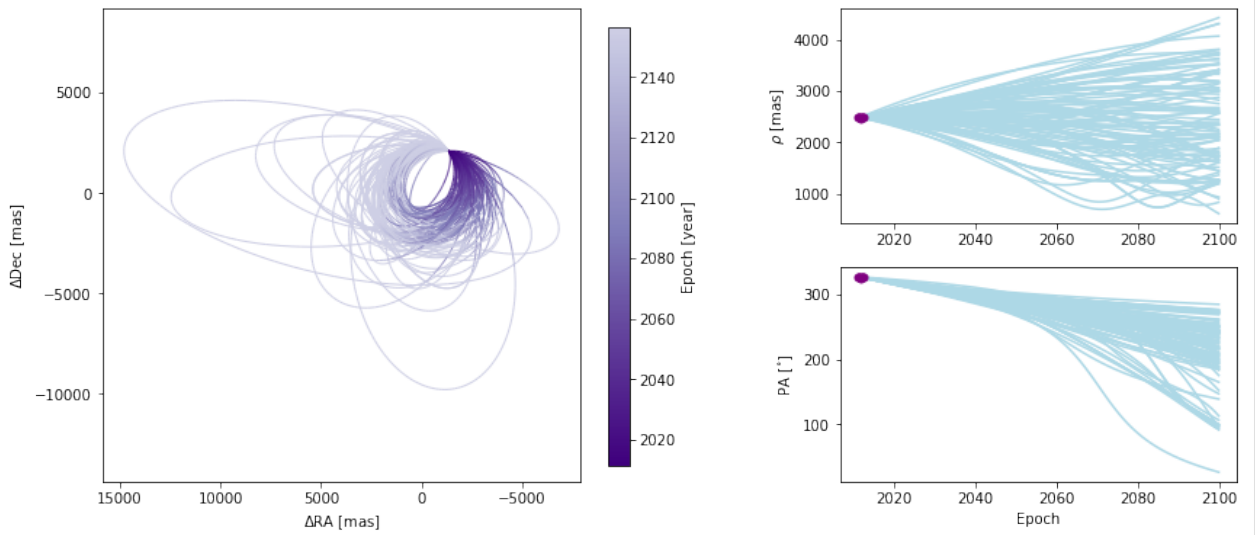
```
<Figure size 1008x432 with 0 Axes>
```



We can also modify the color and ending-epoch of the separation/position angle panels as follows:

[18]:
```
orbit_figure_no_colorbar = myResults.plot_orbits(
    start_mjd=epochs[0], sep_pa_color="lightblue", sep_pa_end_year=2100.0
)
```

```
<Figure size 1008x432 with 0 Axes>
```

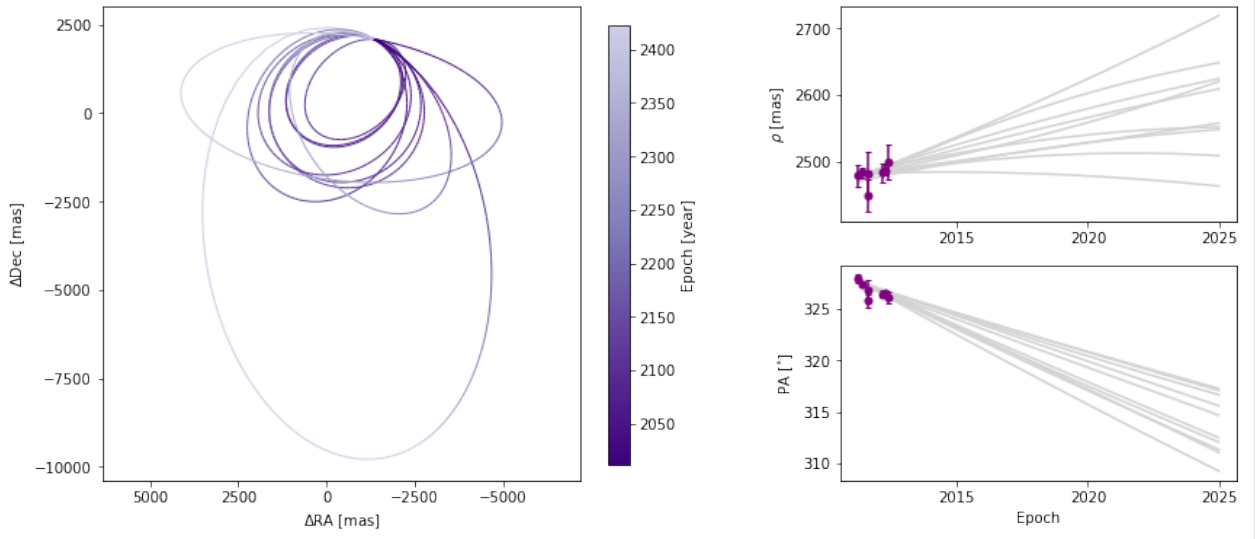


Choosing how orbits are plotted

Plotting one hundred orbits may be too dense. We can set the number of orbits displayed to any other value.

[19]:
```
orbit_figure_plot10 = myResults.plot_orbits(start_mjd=epochs[0], num_orbits_to_plot=10)
```

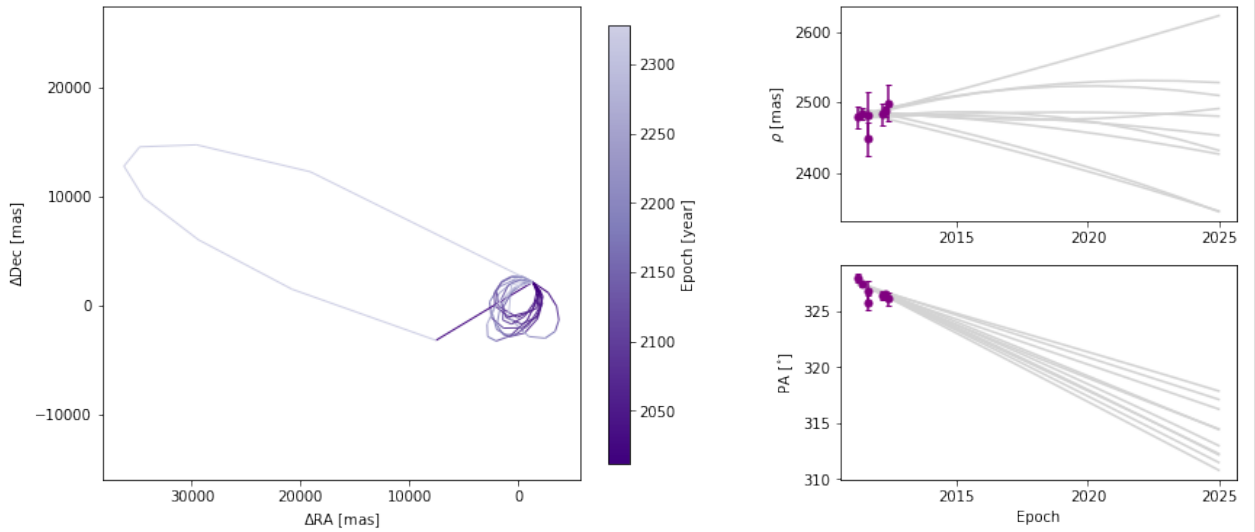
```
<Figure size 1008x432 with 0 Axes>
```



We can also adjust how well we sample each of the orbits. By default, 100 evenly-spaced points are computed per orbit, beginning at `start_mjd` and ending one orbital period later. Decreasing the sampling could lead to faster plot generation but if it is too low, it might not correctly sample the orbit, as shown below.

[20]:
```
orbit_figure_epochs10 = myResults.plot_orbits(
    start_mjd=epochs[0], num_epochs_to_plot=10, num_orbits_to_plot=10
)
```

```
<Figure size 1008x432 with 0 Axes>
```



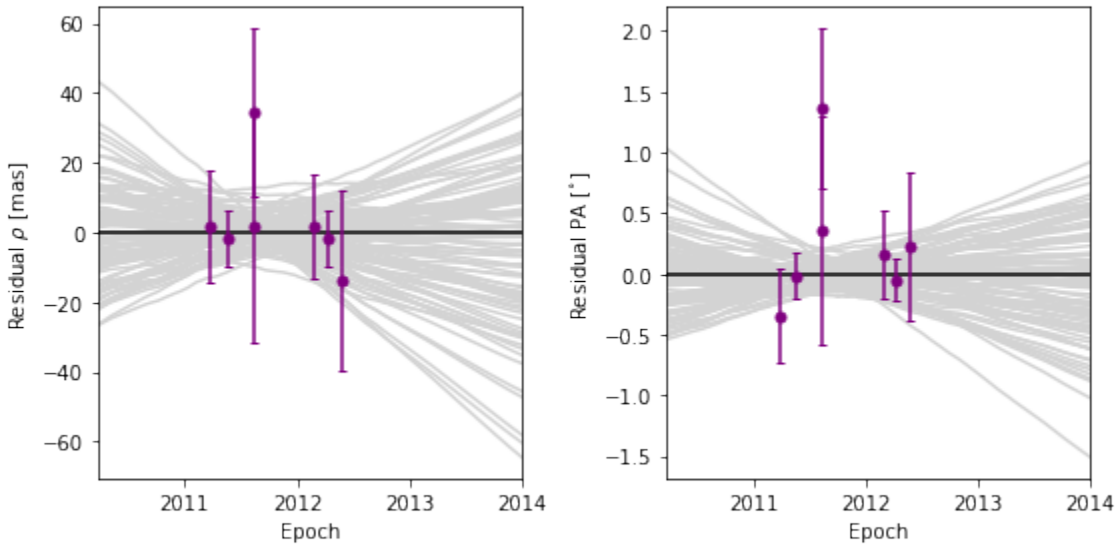
In this example, there is only one companion in orbit around the primary. When there are more than one, `plot_orbits` will plot the orbits of the first companion by default and we would use the `object_to_plot` argument to choose a different object (where 1 is the first companion).

4.3 Residual plots

It is often useful to look at the residuals to the orbit fits, to more carefully understand how well the model and data agree. This can be used to spot systematics in the data, or fitting issues in getting the model to fit the data. There are additional plotting features in the `orbitize.plot` module, including `plot_residuals()`. Currently, `plot_residuals()` can be used to look at the residuals to the relative astrometry data in separation and PA space. Submit a new issue if you want to see other residual plots.

In the example before, we modify the `start_mjd` and `sep_pa_end_year` keywords to zoom in on the segment with data.

[21]:
```
import orbitize.plot

orbitize.plot.plot_residuals(myResults, start_mjd=epochs[0] - 365, sep_pa_end_year=2014)
```



As you can see, the plot shows several models randomly drawn from the posterior. So what are the residuals with respect to? The residuals are with respect to the median sep-vs-time and PA-vs-time trajectories from the random draws. This is true for both the data points and for the models. The residuals of the model with respect to the median model is another way to visualize the spread in the model predictions. If you only plotted one model (by setting the keyword `num_orbits_to_plot=1`), the data residuals would by with respect to that model, and the model residuals would be 0 throughout.

4.4 Working with `matplotlib` Figure objects

The idea of the `Results` plotting functions is to create some basic plots and to return the `matplotlib` Figure object so that we can do whatever else we may want to customize the figure. We should consult the `matplotlib` API for all the details. Here, we will outline a few common tasks.

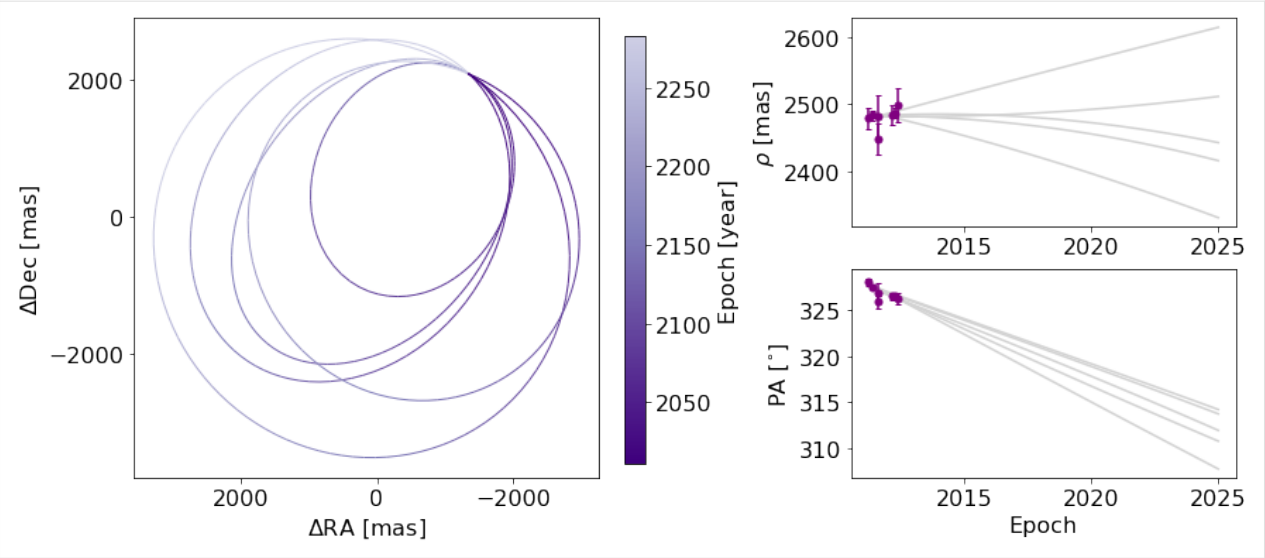
Let's increase the font sizes for all of the text (maybe for a poster or oral presentation) using `matplotlib.pyplot`. This (and other edits to the `rcParams` defaults) should be done before creating any figure objects.

[22]:
```
plt.rcParams.update({"font.size": 16})
```

Now, we will start with creating a figure with only 5 orbits plotted, for simplicity, with the name `orb_fig`. This `Figure` object has two axes, one for the orbit plot and one for the colorbar. We can use the `.axes` property to get a list of axes. Here, we've named the two axes `ax_orb` and `ax_cbar`. With these three objects (`orb_fig` and `ax_orb`, and `ax_cbar`) we can modify all aspects of our figure.

[23]:
```
orb_fig = myResults.plot_orbits(start_mjd=epochs[0], num_orbits_to_plot=5)
ax_orb, ax_sep, ax_pa, ax_cbar = orb_fig.axes
```

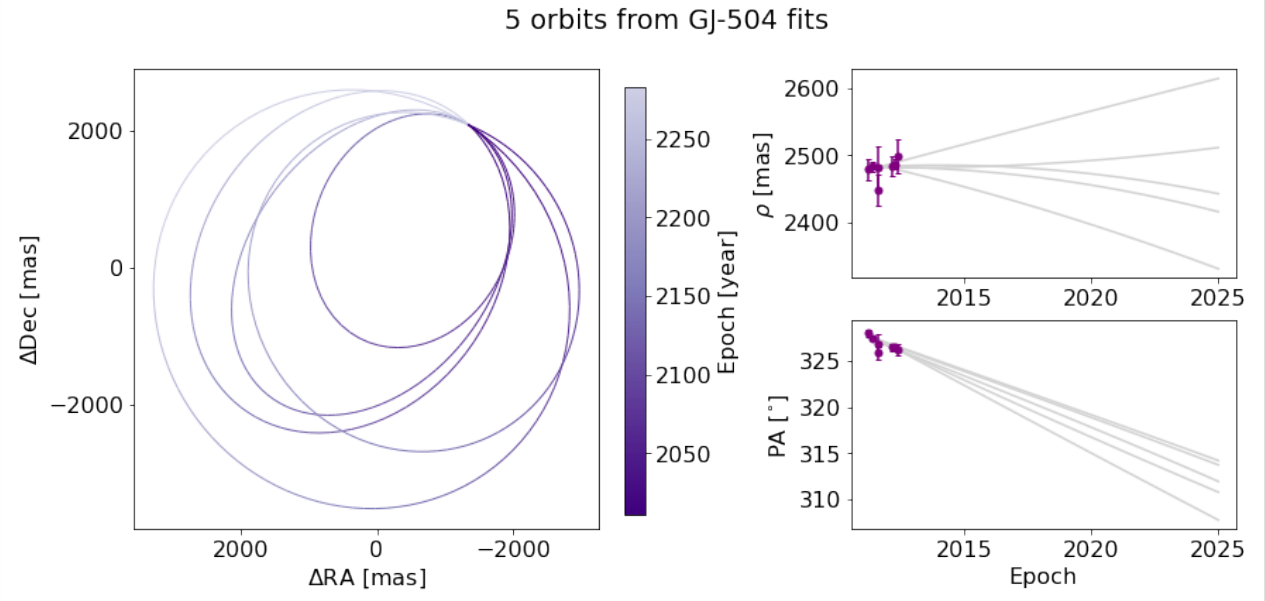
```
<Figure size 1008x432 with 0 Axes>
```



First, let's try to add a figure title. We have two options. We can use the Figure's `suptitle` method to add a title that spans the entire figure (including the colorbar).

[24]:
```
orb_fig.suptitle("5 orbits from GJ-504 fits")  # Adds a title spanning the figure
orb_fig
```

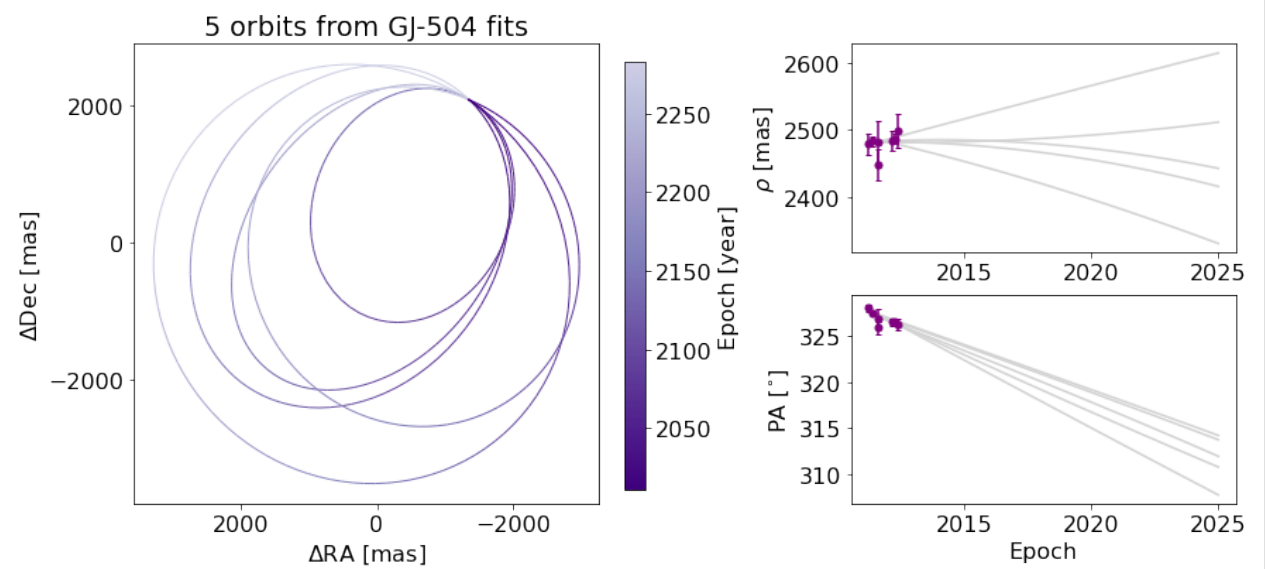
[24]:



Alternatively, we can just add the title over the Ra/Dec axes instead.

[25]:
```
orb_fig.suptitle("")  # Clears previous title
ax_orb.set_title("5 orbits from GJ-504 fits")  # sets title over Axes object only
orb_fig
```

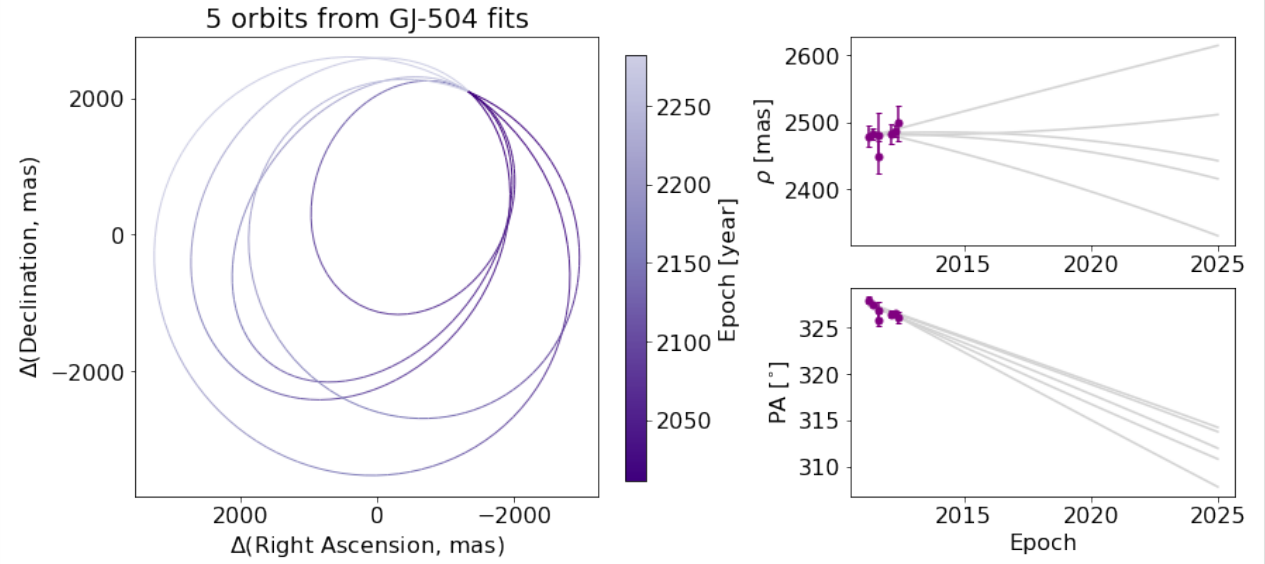
[25]:



We can also change the label of the axes, now using `matplotlib.Axes` methods.

[26]:
```
ax_orb.set_xlabel("$\Delta$(Right Ascension, mas)")
ax_orb.set_ylabel("$\Delta$(Declination, mas)")
orb_fig
```

[26]:



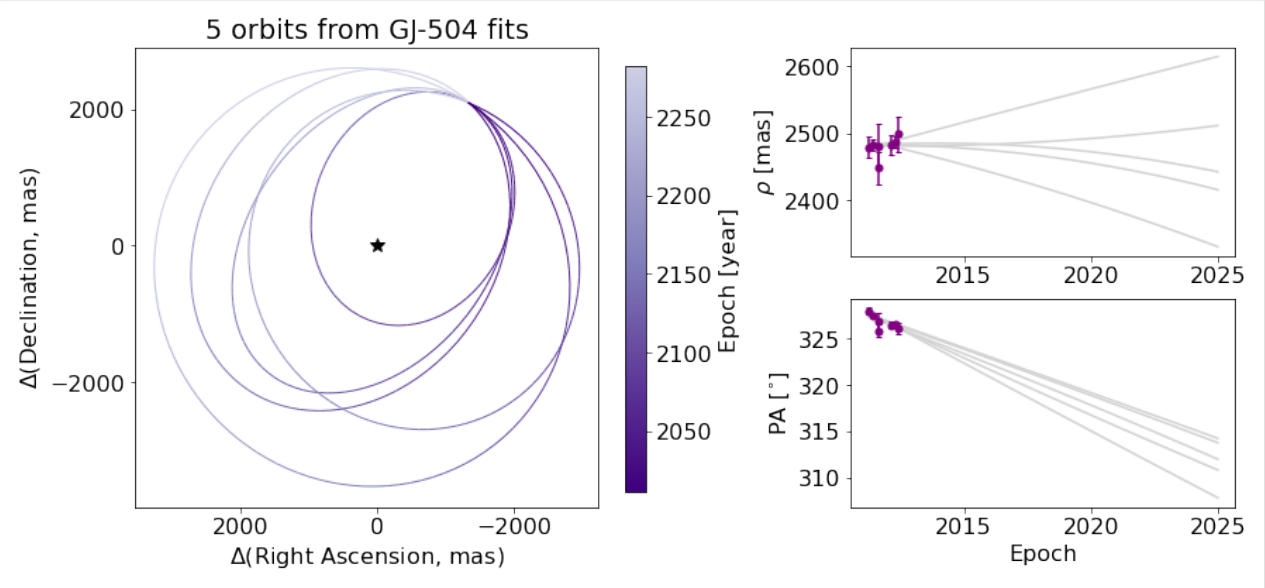
If we want to modify the colorbar axis, we need to access the `ax_cbar` object

ax_orb.set_xlabel('ΔRA [mas]') ax_orb.set_ylabel('ΔDec [mas]') # go back to what we had before ax_cbar.set_title('Year') # Put a title on the colorbar orb_fig

We may want to add to the plot itself. Here's an exmaple of putting an star-shaped point at the location of our primary star.

[27]:
```
ax_orb.plot(0, 0, marker="*", color="black", markersize=10)
orb_fig
```

[27]:



And finally, we can save the figure objects.

[28]:
```
orb_fig.savefig("plotting_tutorial_plot_orbit_example.png")
```

2.2.7 MCMC vs OFTI Comparison

by Sarah Blunt, 2018

Welcome to the OFTI/MCMC comparison tutorial! This tutorial is meant to help you understand the differences between OFTI and MCMC algorithms so you know which one to pick for your data.

Before we start, I'll give you the short answer: **for orbit fractions less than 5%, OFTI is generally faster to converge than MCMC**. This is not a hard-and-fast statistical rule, but I've found it to be a useful guideline.

This tutorial is essentially an abstract of Blunt et al (2017). To dig deeper, I encourage you to read the paper (Sections 2.2-2.3 in particular).

Goals of This Tutorial: - Understand qualitatively why OFTI converges faster than MCMC for certain datasets. - Learn to make educated choices of backend algorithms for your own datasets.

Prerequisites: - This tutorial assumes knowledge of the `orbitize` API. Please go through at least the OFTI and MCMC introduction tutorials before this one. - This tutorial also assumes a qualitative understanding of OFTI and MCMC algorithms. I suggest you check out at least Section 2.1 of Blunt et al (2017) and this blog post before attempting to decode this tutorial.

Jargon: - I will often use **orbit fraction**, or the fraction of the orbit spanned by the astrometric observations, as a figure of merit. In general, OFTI will converge faster than MCMC for small orbit fractions. - **Convergence** is defined differently for OFTI and for MCMC (see the OFTI paper for details). An OFTI run needs to accept a statistically large number of orbits for convergence, since each accepted orbit is independent of all others. For MCMC, convergence is a bit more complicated. At a high level, an MCMC run has converged when all walkers have explored the entire parameter space. There are several metrics for estimating MCMC convergence (e.g. GR statistic, min Tz statistic), but we'll just estimate convergence qualitatively in this tutorial.

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

(continues on next page)

(continued from previous page)

```
import astropy.table
import time

np.random.seed(5)

from orbitize.kepler import calc_orbit
from orbitize import system, sampler
from orbitize.read_input import read_file
```

Generate Synthetic Data

Let's start by defining a function to generate synthetic data. This will allow us to easily test convergence speeds for different orbit fractions. I'll include the number of observations and the noise magnitude as keywords; I encourage you to test out different values throughout the tutorial!

[2]:
```
mtot = 1.2  # total system mass [M_sol]
plx = 60.0  # parallax [mas]


def generate_synthetic_data(sma=30.0, num_obs=4, unc=10.0):
    """Generate an orbitize-table of synethic data

    Args:
        sma (float): semimajor axis (au)
        num_obs (int): number of observations to generate
        unc (float): uncertainty on all simulated RA & Dec measurements (mas)

    Returns:
        2-tuple:
            - `astropy.table.Table`: data table of generated synthetic data
            - float: the orbit fraction of the generated data
    """

    # assumed ground truth for non-input orbital parameters
    ecc = 0.5  # eccentricity
    inc = np.pi / 4  # inclination [rad]
    argp = 0.0
    lan = 0.0
    tau = 0.8

    # calculate RA/Dec at three observation epochs
    observation_epochs = np.linspace(
        51550.0, 52650.0, num_obs
    )  # `num_obs` epochs between ~2000 and ~2003 [MJD]
    num_obs = len(observation_epochs)
    ra, dec, _ = calc_orbit(
        observation_epochs, sma, ecc, inc, argp, lan, tau, plx, mtot
    )

    # add Gaussian noise to simulate measurement
    ra += np.random.normal(scale=unc, size=num_obs)
```

(continues on next page)

(continued from previous page)

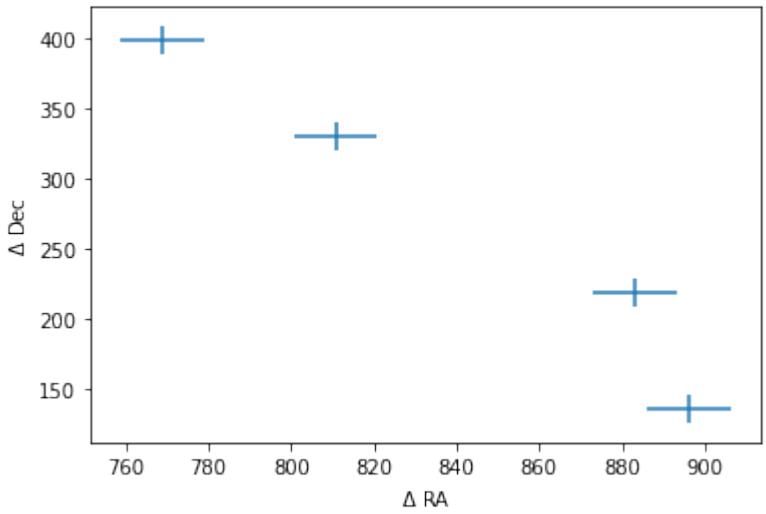
```
    dec += np.random.normal(scale=unc, size=num_obs)

    # define observational uncertainties
    ra_err = dec_err = np.ones(num_obs) * unc

    # make a plot of the data
    plt.figure()
    plt.errorbar(ra, dec, xerr=ra_err, yerr=dec_err, linestyle="")
    plt.xlabel("$\\Delta$ RA")
    plt.ylabel("$\\Delta$ Dec")

    # calculate the orbital fraction
    period = np.sqrt((sma**3) / mtot)
    orbit_coverage = (
        max(observation_epochs) - min(observation_epochs)
    ) / 365.25  # [yr]
    orbit_fraction = 100 * orbit_coverage / period

    data_table = astropy.table.Table(
        [observation_epochs, [1] * num_obs, ra, ra_err, dec, dec_err],
        names=("epoch", "object", "raoff", "raoff_err", "decoff", "decoff_err"),
    )
    # read into orbitize format
    data_table = read_file(data_table)

    return data_table, orbit_fraction
```

Short Orbit Fraction

Let's use the function above to generate some synthetic data with a short orbit fraction, and fit it with OFTI:

[3]:
```
# generate data with default kwargs
short_data_table, short_orbit_fraction = generate_synthetic_data()
print("The orbit fraction is {}%".format(np.round(short_orbit_fraction), 1))

# initialize orbitize `System` object
short_system = system.System(1, short_data_table, mtot, plx)

num2accept = 500  # run sampler until this many orbits are accepted
```

```
The orbit fraction is 2.0%
```



[4]:
```
start_time = time.time()

# set up OFTI `Sampler` object
short_OFTI_sampler = sampler.OFTI(short_system)

# perform OFTI fit
short_OFTI_orbits = short_OFTI_sampler.run_sampler(num2accept)

print(
    "OFTI took {} seconds to accept {} orbits.".format(
        time.time() - start_time, num2accept
    )
)
```

```
Converting ra/dec data points in data_table to sep/pa. Original data are stored in input_
↪table.
497/500 orbits found
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Input In [4], in <cell line: 7>()
      4 short_OFTI_sampler = sampler.OFTI(short_system)
      6 # perform OFTI fit
----> 7 short_OFTI_orbits = short_OFTI_sampler.run_sampler(num2accept)
      9 print(
     10     "OFTI took {} seconds to accept {} orbits.".format(
     11         time.time() - start_time, num2accept
     12     )
     13 )

File ~/Documents/GitHub/orbitize/orbitize/sampler.py:593, in OFTI.run_sampler(self,␣
↪total_orbits, num_samples, num_cores, OFTI_warning)
    588             OFTI_warning = None
    589     print(
    590         str(orbits_saved.value) + "/" + str(total_orbits) + " orbits found",
    591         end="\r",
```

(continues on next page)

(continued from previous page)

```
    592     )
--> 593     time.sleep(0.1)
    595 print(
    596     str(total_orbits) + "/" + str(total_orbits) + " orbits found", end="\r"
    597 )
    599 # join the processes

KeyboardInterrupt:
```

[ ]:
```
start_time = time.time()

# set up MCMC `Sampler` object
num_walkers = 20
short_MCMC_sampler = sampler.MCMC(short_system, num_temps=5, num_walkers=num_walkers)

# perform MCMC fit
num2accept_mcmc = 10 * num2accept
_ = short_MCMC_sampler.run_sampler(num2accept_mcmc, burn_steps=100)
short_MCMC_orbits = short_MCMC_sampler.results.post

print(
    "MCMC took {} steps in {} seconds.".format(
        num2accept_mcmc, time.time() - start_time
    )
)
```

[ ]:
```
plt.hist(
    short_OFTI_orbits[:, short_system.param_idx["ecc1"]],
    bins=40,
    density=True,
    alpha=0.5,
    label="OFTI",
)
plt.hist(
    short_MCMC_orbits[:, short_system.param_idx["ecc1"]],
    bins=40,
    density=True,
    alpha=0.5,
    label="MCMC",
)

plt.xlabel("Eccentricity")
plt.ylabel("Prob.")
plt.legend()
```

These distributions are different because the MCMC chains have not converged, resulting in a "lumpy" MCMC distribution. I set up the calculation so that MCMC would return 10x as many orbits as OFTI, but even so, the OFTI distribution is a much better representation of the underlying PDF.

If we run the MCMC algorithm for a greater number of steps (and/or increase the number of walkers and/or temperatures), the MCMC and OFTI distributions will become indistinguishable. **OFTI is NOT more correct than MCMC, but for this dataset, OFTI converges on the correct posterior faster than MCMC**.

Longer Orbit Fraction

Let's now repeat this exercise with a longer orbit fraction. For this dataset, OFTI will have to run for several seconds just to accept one orbit, so we won't compare the resulting posteriors.

[ ]:
```
# generate data
long_data_table, long_orbit_fraction = generate_synthetic_data(sma=10, num_obs=5)
print("The orbit fraction is {}%".format(np.round(long_orbit_fraction), 1))

# initialize orbitize `System` object
long_system = system.System(1, long_data_table, mtot, plx)
num2accept = 500  # run sampler until this many orbits are accepted
```

[ ]:
```
start_time = time.time()

# set up OFTI `Sampler` object
long_OFTI_sampler = sampler.OFTI(long_system)

# perform OFTI fit
long_OFTI_orbits = long_OFTI_sampler.run_sampler(1)

print("OFTI took {} seconds to accept 1 orbit.".format(time.time() - start_time))
```

[ ]:
```
start_time = time.time()

# set up MCMC `Sampler` object
num_walkers = 20
long_MCMC_sampler = sampler.MCMC(long_system, num_temps=10, num_walkers=num_walkers)

# perform MCMC fit
_ = long_MCMC_sampler.run_sampler(num2accept, burn_steps=100)
long_MCMC_orbits = long_MCMC_sampler.results.post

print("MCMC took {} steps in {} seconds.".format(num2accept, time.time() - start_time))
```

[ ]:
```
plt.hist(long_MCMC_orbits[:, short_system.param_idx["ecc1"]], bins=15, density=True)
plt.xlabel("Eccentricity")
plt.ylabel("Prob.")
```

It will take more steps for this MCMC to fully converge (see the MCMC tutorial for more detailed guidelines), but you can imagine that MCMC will converge much faster than OFTI for this dataset.

Closing Thoughts

If you play around with the `num_obs`, `sma`, and `unc` keywords in the `generate_synthetic_data` function and repeat this exercise, you will notice that the OFTI acceptance rate and MCMC convergence rate depend on many variables, not just orbit fraction. **In truth, the Gaussianity of the posterior space determines how quickly an MCMC run will converge, and its similarity to the prior space determines how quickly an OFTI run will converge. In other words, the more constrained your posteriors are (relative to your priors), the quicker MCMC will converge, and the slower OFTI will run.**

Orbit fraction is usually a great tracer of this "amount of constraint," but it's good to understand why!

Summary: - OFTI and MCMC produce the same posteriors, but often take differing amounts of time to converge on the correct solution. - OFTI is superior when your posteriors are similar to your priors, and MCMC is superior when your posteriors are highly constrained Gaussians.

2.2.8 Modifying MCMC Initial Positions

by Henry Ngo (2019) & Sarah Blunt (2021) & Mireya Arora (2021)

When you set up the MCMC Sampler, the initial position of your walkers are randomly determined. Specifically, they are uniformly distributed in your Prior phase space. This tutorial will show you how to change this default behaviour so that the walkers can begin at locations you specify. For instance, if you have an initial guess for the best fitting orbit and want to use MCMC to explore posterior space around this peak, you may want to start your walkers at positions centered around this peak and distributed according to an N-dimensional Gaussian distribution.

Note: This tutorial is meant to be read after reading the MCMC Introduction tutorial. If you are wondering what walkers are, you should start there!

The `Driver` class is the main way you might interact with `orbitize!` as it automatically reads your input, creates all the `orbitize!` objects needed to do your calculation, and defaults to some commonly used parameters or settings. However, sometimes you want to work directly with the underlying API to do more advanced tasks such as changing the MCMC walkers' initial positions, or modifying the priors.

This tutorial walks you through how to do that.

Goals of this tutorial: - Learn to modify the MCMC `Sampler` object - Learn about the structure of the `orbitize` code base

Import modules

[1]:
```
import numpy as np
from scipy.optimize import minimize as mn
import orbitize
from orbitize import driver
import multiprocessing as mp
```

1) Create Driver object

First, let's begin as usual and create our `Driver` object, as in the MCMC Introduction tutorial.

[2]:
```
filename = "{}/GJ504.csv".format(orbitize.DATADIR)

# system parameters
num_secondary_bodies = 1
total_mass = 1.75  # [Msol]
plx = 51.44  # [mas]
mass_err = 0.05  # [Msol]
plx_err = 0.12  # [mas]

# MCMC parameters
num_temps = 5
num_walkers = 30
num_threads = mp.cpu_count()  # or a different number if you prefer
```

(continues on next page)

(continued from previous page)

```
my_driver = driver.Driver(
    filename,
    "MCMC",
    num_secondary_bodies,
    total_mass,
    plx,
    mass_err=mass_err,
    plx_err=plx_err,
    mcmc_kwargs={
        "num_temps": num_temps,
        "num_walkers": num_walkers,
        "num_threads": num_threads,
    },
)
```

2) Access the `Sampler` object to view the walker positions

As mentioned in the introduction, the `Driver` class creates the objects needed for the orbit fit. At the time of this writing, it creates a `Sampler` object which you can access with the `.sampler` attribute and a `System` object which you can access with the `.system` attribute.

The `Sampler` object contains all of the information used by the orbit sampling algorithm (OFTI or MCMC) to fit the orbit and determine the posteriors. The `System` object contains information about the astrophysical system itself (stellar and companion parameters, the input data, etc.).

To see all of the attributes of the driver object, you can use `dir()`.

[3]:
```
print(dir(my_driver))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '_
↪_ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__
↪le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__
↪repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
↪'sampler', 'system']
```

This returns many other functions too, but you see `sampler` and `system` at the bottom. Don't forget that in Jupyter notebooks, you can use `my_driver?` to get the docstring for its class (i.e. the `Driver` class) and `my_driver??` to get the full source code of that class. You can also get this information in the API documentation.

Now, let's list the attributes of the `my_driver.sampler` attribute.

[4]:
```
print(dir(my_driver.sampler))
```

```
['__abstractmethods__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__
↪eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '_
↪_init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
↪ '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__slots__', '__str__', '__
↪subclasshook__', '__weakref__', '_abc_impl', '_fill_in_fixed_params', '_logl', '_
↪update_chains_from_sampler', 'check_prior_support', 'chi2_type', 'chop_chains', 'curr_
↪pos', 'custom_lnlike', 'examine_chains', 'fixed_params', 'has_corr', 'lnlike', 'num_
↪params', 'num_temps', 'num_threads', 'num_walkers', 'priors', 'results', 'run_sampler',
↪ 'sampled_param_idx', 'system', 'use_pt', 'validate_xyz_positions']
```

Again, you can use the ? and ?? features as well as the API documentation to find out more. Here we see an attribute `curr_pos` which contains the current position of all the walkers for the MCMC sampler. These positions were generated upon initialization of the `Sampler` object, which happened as part of the initialization of the `Driver` object.

Examine `my_driver.sampler.curr_pos`

`curr_pos` is an array and has shape (`n_temps`, `n_walkers`, `n_params`) for the parallel-tempered MCMC sampler and shape (`n_walkers`, `n_params`) for the affine-invariant ensemble sampler.

[5]:
```
my_driver.sampler.curr_pos.shape  # Here we are using the parallel-tempered MCMC sampler
```

[5]:
```
(5, 30, 8)
```

Basically, this is the same shape as the output of the Sampler. Let's look at the start position of the first five walkers at the lowest temperature, to get a better sense of what the strucutre is like.

[6]:
```
print(my_driver.sampler.curr_pos[0, 0:5, :])
```

```
[[9.27153771e+03 7.72052247e-02 2.48173602e+00 1.45797651e+00
  4.43581764e+00 5.33957331e-01 5.11155491e+01 1.80782845e+00]
 [4.20682067e-01 9.89156369e-01 4.61093206e-01 1.25947642e+00
  2.21880036e+00 8.30484942e-01 5.16175061e+01 1.74926489e+00]
 [2.93009249e+01 4.42876011e-01 1.85406431e+00 5.21790350e-02
  4.29764787e+00 7.13679469e-02 5.14691320e+01 1.76691096e+00]
 [1.46043332e+03 8.26045166e-01 1.17672512e-01 1.66771136e+00
  4.94514469e+00 5.58027305e-01 5.15399393e+01 1.83912467e+00]
 [5.26417460e+01 7.93835235e-01 2.01212573e+00 6.43273326e-01
  5.69960003e+00 9.72070078e-02 5.16224931e+01 1.77433834e+00]]
```

3) Replace `curr_pos` with your own initial positions for walkers

When the sampler is run with the `sampler.run_sampler()` method, it will start the walkers at the `curr_pos` values, run the MCMC forward for the given number of steps, and then update `curr_pos` to reflect where the walkers ended up. The next time `run_sampler()` is called, it does the same thing again.

Here, you have just created the sampler but have not run it yet. So, if you update `curr_pos` with our own custom start locations, when you run the sampler, it will begin at your custom start locations instead.

3.1) Generate your own initial positions

There are many ways to create your own walker start distribution and what you want to do will depend on your science question and prior knowledge.

If you have already generated and validated your own initial walker positions, you can skip down to the "Update sampler position". Some users use the output of OFTI or a previous MCMC run as the initial position.

If you need to generate your own positions, read on. Here, let's assume you know a possible best fit value and your uncertainty in that fit. Perhaps you got this through a least squares minimization. So, let's create a distribution of walkers that are centered on the best fit value and distributed normallly with the 1-sigma in each dimension equal to the uncertainty on that best fit value.

First, let's define the best fit value and the spread. As a reminder, the order of the parameters in the array is (for a single planet-star system): semimajor axis, eccentricity, inclination, argument of periastron, position angle of nodes, epoch of periastron passage, parallax and total mass. You can check the indices with this dict in the `system` object.

[7]:
```
print(my_driver.system.param_idx)
```

```
{'sma1': 0, 'ecc1': 1, 'inc1': 2, 'aop1': 3, 'pan1': 4, 'tau1': 5, 'plx': 6, 'mtot': 7}
```

[8]:
```
# Set centre and spread of the walker distribution
# Values from Table 1 in Blunt et al. 2017, AJ, 153, 229
sma_cen = 44.48
sma_sig = 15.0
ecc_cen = 0.0151
ecc_sig = 0.175
inc_cen = 2.30  # (131.7 deg)
inc_sig = 0.279  # (16.0 deg)
aop_cen = 1.60  # (91.7 deg)
aop_sig = 1.05  # (60.0 deg)
pan_cen = 2.33  # (133.7 deg)
pan_sig = 0.872  # (50.0 deg)
tau_cen = 0.77  # (2228.11 yr)
tau_sig = 0.65  # (121.0 yr)

# Note : parallax and stellar mass already defined above (plx, plx_err, total_mass, mass_
↪err)
walker_centres = np.array(
    [sma_cen, ecc_cen, inc_cen, aop_cen, pan_cen, tau_cen, plx, total_mass]
)
walker_1sigmas = np.array(
    [sma_sig, ecc_sig, inc_sig, aop_sig, pan_sig, tau_sig, plx_err, mass_err]
)
```

You can use `numpy.random.standard_normal` to generate normally distributed random numbers in the same shape as your walker initial positions (`my_driver.sampler.curr_pos.shape`). Then, multiply by `walker_1sigmas` to get the spread to match your desired distribution and add `walker_centres` to get the distribution centered on your desired values.

[9]:
```
curr_pos_shape = my_driver.sampler.curr_pos.shape  # Get shape of walker positions

# Draw from multi-variate normal distribution to generate new walker positions
new_pos = np.random.standard_normal(curr_pos_shape) * walker_1sigmas + walker_centres
```

3.2) Using an optimizer to obtain a best fit value

Other optimizing software can also be used to generate intial positions. Depending on the quality of data collected and whether a suitable guess array of parameters can be made, different optimizing software can provide better best fit values for for MCMC walkers. Below you will find a few options that cater to different scenarios.

3.2a) Using `scipy.optimize.minimize`

Assuming the data obtained allows for a suitable guess to be made for each parameter, a scipy.optimize.minimize software can be used to generate a best fit value. You may want to skip this step and input your guess values directly into MCMC's initial walker positions, however scipy can help refine the guess parameters.

First, we define a new log liklihood function function `neg_logl` based on the guess values we have. Note, since we have predefined a good guess, from the aforementioned Table, as `walker_centres` we will continue to use it as a guess array for examples below.

[10]:
```
# The following code performs a minimization whereas the log liklihood function is based␣
↪on maximization so we redefine the
# likelihood function is redefined to return -x to make this a minization scenario

m = my_driver.sampler


def neg_logl(paramarray):
    x = m._logl(
        paramarray, include_logp=True
    )  # set include_logp to true to include guess array in likelihood function

    return -x


guessarray = walker_centres
results = mn(neg_logl, guessarray, method="Powell")
print(results.x)  # results.x is the best fit value
```

```
/home/docs/checkouts/readthedocs.org/user_builds/orbitize/envs/latest/lib/python3.10/
↪site-packages/scipy/optimize/_optimize.py:2473: RuntimeWarning: invalid value␣
↪encountered in scalar multiply
  tmp2 = (x - v) * (fx - fw)
/home/docs/checkouts/readthedocs.org/user_builds/orbitize/envs/latest/lib/python3.10/
↪site-packages/orbitize/priors.py:564: RuntimeWarning: invalid value encountered in log
  lnprob = np.log(np.sin(element_array) / normalization)
```

```
[4.90426906e+01 9.99444988e-06 2.48501010e+00 1.60085022e+00
 2.33034111e+00 7.69934177e-01 5.14404237e+01 1.74922411e+00]
```

In our trials, Powell has given the best results, but you may replace it with a different minimizing method depending on your need.

3.3) Scattering walkers

To set up MCMC so that it explores the nearby probablity space thoroughly and finds the global minimum, you can scatter the initial positions of the walkers around the best fit value. This can be done by adding random numbers to `results.x`

This section overrides walker_1sigmas and creates a spread of `new_pos` in a different manner than above. The following is a template based on the aforementioned Table. The scatter is created using a variety of methods, we recommend reviewing the code to ensure it is compatible to your data.

[11]:
```
new_pos = np.random.standard_normal(curr_pos_shape) * 0.03 + results.x
```

3.4) Update sampler position

After generating and validating your new walker positions, through whatever methods you choose, it's now time to update the sampler object to have its `curr_pos` be your new positions.

[12]:
```
my_driver.sampler.curr_pos = np.copy(new_pos)
```

3.5) Validate your new positions

Drawing from a normal distribution can cause your walkers to start outside of your prior space. See the Modifying Priors tutorial for information on how to interact with the prior objects, which would allow you to find the limits on each parameter set by the priors etc.

Here, let's do something more simple and just check that all values are physically valid. After this we can begin to correct them.

The following function can be used to identify walkers that have been initialized outside of the appropriate prior probability space. It will raise a `ValueError` if walkers are initialized outside of the priors. You should update your positions until this method runs without raising an error.

[13]:
```
try:
    my_driver.sampler.check_prior_support()
except Exception as e:
    print(e)
```

```
Attempting to start with walkers outside of prior support: check parameter(s) 1
```

We should continue investigating which parameters are being initialized outside of the prior space until this function returns empty lists.

And you're done! You can continue at "Running the MCMC Sampler" in the MCMC Introduction Tutorial

2.2.9 Radial Velocity Tutorial for MCMC

By Roberto Tejada (2019)

This tutorial will assume the user is familiar with the `Driver` class and is acquainted with MCMC terminology. For more information about MCMC, see the MCMC Introduction Tutorial.

We explain how to jointly fit radial velocity data and relative astrometry using the MCMC technique. First we need a set of data containing radial velocity measurements. We check the data using `read_input` and observe the `quant_type` column for radial velocity data. For more information on `orbitize.read_input.read_file()`, see the Formatting Input Tutorial. You can fit for separate jitter and gamma terms for each RV instrument in your dataset by adding an "instrument" column to your data csv.

NOTE: Astrometry+RV fitting currently only works with MCMC and not OFTI.

Read and Format Data

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from orbitize import read_input, plot, priors, driver, DATADIR
import multiprocessing as mp

data_table = read_input.read_file("{}/HD4747.csv".format(DATADIR)) # print all columns
data_table.pprint_all()
```

```
  epoch   object  quant1  quant1_err quant2 quant2_err quant12_corr quant_type instrument
--------- ------ -------- ---------- ------ ---------- ------------ ---------- ----------
  56942.3      1    606.5        7.0 180.04       0.62          nan      seppa       defsp
  57031.2      1    606.6        6.4 180.52       0.58          nan      seppa       defsp
  57289.4      1    604.0        7.0  184.9        0.9          nan      seppa       defsp
50366.475      0 -0.54103    0.00123    nan        nan          nan         rv       defrv
50418.329      0 -0.40053    0.00206    nan        nan          nan         rv       defrv
50462.252      0 -0.24094     0.0011    nan        nan          nan         rv       defrv
50689.604      0  0.37292    0.00117    nan        nan          nan         rv       defrv
50784.271      0  0.46223    0.00133    nan        nan          nan         rv       defrv
50806.227      0  0.48519    0.00103    nan        nan          nan         rv       defrv
50837.217      0  0.49395    0.00117    nan        nan          nan         rv       defrv
50838.201      0  0.49751    0.00112    nan        nan          nan         rv       defrv
50839.211      0  0.50187    0.00112    nan        nan          nan         rv       defrv
 51009.62      0  0.53355    0.00135    nan        nan          nan         rv       defrv
51011.548      0  0.53164    0.00128    nan        nan          nan         rv       defrv
51013.602      0  0.53629     0.0016    nan        nan          nan         rv       defrv
51050.491      0  0.52154    0.00468    nan        nan          nan         rv       defrv
51170.248      0  0.50757     0.0014    nan        nan          nan         rv       defrv
51367.585      0  0.47678    0.00131    nan        nan          nan         rv       defrv
51409.527      0  0.46147    0.00523    nan        nan          nan         rv       defrv
51543.244      0  0.44311    0.00152    nan        nan          nan         rv       defrv
51550.229      0  0.43286    0.00147    nan        nan          nan         rv       defrv
51755.551      0  0.39329    0.00213    nan        nan          nan         rv       defrv
51899.284      0  0.36457    0.00163    nan        nan          nan         rv       defrv
52097.626      0  0.32986    0.00157    nan        nan          nan         rv       defrv
52488.542      0  0.26687     0.0015    nan        nan          nan         rv       defrv
52572.312      0  0.25035    0.00195    nan        nan          nan         rv       defrv
52987.228      0  0.19466    0.00238    nan        nan          nan         rv       defrv
52988.186      0  0.18469    0.00194    nan        nan          nan         rv       defrv
53238.456      0  0.16892    0.00134    nan        nan          nan         rv       defrv
53303.403      0  0.16769    0.00112    nan        nan          nan         rv       defrv
53339.265      0  0.16069    0.00119    nan        nan          nan         rv       defrv
53724.274      0  0.11302    0.00103    nan        nan          nan         rv       defrv
53724.276      0  0.11605    0.00112    nan        nan          nan         rv       defrv
54717.455      0  0.00984    0.00123    nan        nan          nan         rv       defrv
54718.508      0  0.01242    0.00115    nan        nan          nan         rv       defrv
 54719.51      0  0.01572    0.00123    nan        nan          nan         rv       defrv
 54720.47      0  0.01534    0.00113    nan        nan          nan         rv       defrv
54722.401      0  0.01479    0.00127    nan        nan          nan         rv       defrv
 54723.47      0  0.01422    0.00122    nan        nan          nan         rv       defrv
54724.474      0  0.01169     0.0012    nan        nan          nan         rv       defrv
54725.383      0  0.01383    0.00113    nan        nan          nan         rv       defrv
```

(continues on next page)

(continued from previous page)

```
54726.505      0   0.0195    0.00123    nan        nan        nan        rv      defrv
54727.452      0   0.0175    0.00113    nan        nan        nan        rv      defrv
 55014.62      0 -0.00636    0.00141    nan        nan        nan        rv      defrv
55015.624      0 -0.00409    0.00138    nan        nan        nan        rv      defrv
55016.624      0 -0.00566    0.00121    nan        nan        nan        rv      defrv
55048.525      0 -0.01975    0.00124    nan        nan        nan        rv      defrv
55076.584      0 -0.01614    0.00128    nan        nan        nan        rv      defrv
55077.594      0 -0.01303    0.00126    nan        nan        nan        rv      defrv
55134.464      0 -0.01689    0.00136    nan        nan        nan        rv      defrv
55198.254      0 -0.02885     0.0012    nan        nan        nan        rv      defrv
55425.584      0 -0.04359    0.00125    nan        nan        nan        rv      defrv
55522.379      0  -0.0512     0.0013    nan        nan        nan        rv      defrv
55806.547      0 -0.07697     0.0013    nan        nan        nan        rv      defrv
 56148.55      0 -0.10429    0.00128    nan        nan        nan        rv      defrv
56319.201      0  -0.1102    0.00128    nan        nan        nan        rv      defrv
56327.208      0 -0.11332    0.00149    nan        nan        nan        rv      defrv
56507.645      0 -0.12324    0.00133    nan        nan        nan        rv      defrv
56912.534      0 -0.17085    0.00113    nan        nan        nan        rv      defrv
```

The `quant_type` column displays the type of data each row contains: astrometry (radec or seppa), or radial velocity (rv). For astrometry, `quant1` column contains right ascension or separation, and the `quant2` column contains declination or position angle. For rv data, `quant1` contains radial velocity data in $\mathrm{km/s}$, while `quant2` is filled with `nan` to preserve the data structure. The table contains each respective error column.

We can now initialize the `Driver` class. MCMC samplers take time to converge to absolute maxima in parameter space, and the more parameters we introduce, the longer we expect it to take.

Create Driver Object

For joint orbital fits with RV data, we need to fit the stellar and companion masses (`m0` and `m1` respectively as separate free parameters). This differs from the astrometry-only case where fitting the total mass `mtot` suffices. We set the system keyword `fit_secondary_mass` to `True` when initializing the Driver object.

[2]:
```
filename = "{}/HD4747.csv".format(DATADIR)

# system parameters
num_secondary_bodies = 1
stellar_mass = 0.84 # [Msol]
plx = 53.18 # [mas]
mass_err = 0.04 # [Msol]
plx_err = 0.12 # [mas]

# MCMC parameters
num_temps = 5
num_walkers = 30
num_threads = 2 # or a different number if you prefer, eg mp.cpu_count()

my_driver = driver.Driver(
    filename, 'MCMC', num_secondary_bodies, stellar_mass, plx, mass_err=mass_err, plx_
↪err=plx_err,
    system_kwargs = {'fit_secondary_mass':True, 'tau_ref_epoch':0},
    mcmc_kwargs={'num_temps': num_temps, 'num_walkers': num_walkers, 'num_threads': num_
```

(continues on next page)

(continued from previous page)

```
↪threads}
)
```

Since MCMC is an object in `orbitize!`, we can assign a variable to the sampler and work with this variable:

[3]:
```
m = my_driver.sampler
```

RV Priors

The priors for the two RV parameters, the radial velocity offset (gamma), and jitter (sigma), have default uniform prior and log uniform prior respectively. The gamma uniform prior is set between $(-5, 5)$ km/s, and the jitter log uniform prior is set for $(10^{-4}, 0.05)$ km/s. The prior for `m1` is a log uniform prior and is set for $(10^{-3}, 2.0) M_{\odot}$. The current version of orbitize addressed in this tutorial returns the stellar radial velocity only.

NOTE: We may change the priors as instructed in the Modifying Priors tutorial:

[4]:
```
# getting the system object:
sys = my_driver.system

lab = sys.param_idx

print(sys.labels)
print(sys.sys_priors)

print(vars(sys.sys_priors[lab['m1']]))
```

```
['sma1', 'ecc1', 'inc1', 'aop1', 'pan1', 'tau1', 'plx', 'gamma_defrv', 'sigma_defrv', 'm1
↪', 'm0']
[Log Uniform, Uniform, Sine, Uniform, Uniform, Uniform, Gaussian, Uniform, Log Uniform,␣
↪Log Uniform, Gaussian]
{'minval': 1e-06, 'maxval': 2, 'logmin': -13.815510557964274, 'logmax': 0.
↪6931471805599453}
```

[5]:
```
# change the m1 prior:
sys.sys_priors[lab['m1']] = priors.LogUniformPrior(1e-4, 0.5)

print(sys.labels)
print(sys.sys_priors)
print(vars(sys.sys_priors[lab['m1']]))
```

```
['sma1', 'ecc1', 'inc1', 'aop1', 'pan1', 'tau1', 'plx', 'gamma_defrv', 'sigma_defrv', 'm1
↪', 'm0']
[Log Uniform, Uniform, Sine, Uniform, Uniform, Uniform, Gaussian, Uniform, Log Uniform,␣
↪Log Uniform, Gaussian]
{'minval': 0.0001, 'maxval': 0.5, 'logmin': -9.210340371976182, 'logmax': -0.
↪6931471805599453}
```

Running the MCMC Sampler

As noted in the MCMC Introduction Tutorial, we must choose the sampler step for MCMC and can save every nth sample to avoid using too much disk space using `thin`.

[6]:
```
total_orbits = 1000 # number of steps x number of walkers (at lowest temperature)
burn_steps = 10 # steps to burn in per walker
thin = 2 # only save every 2 steps
```

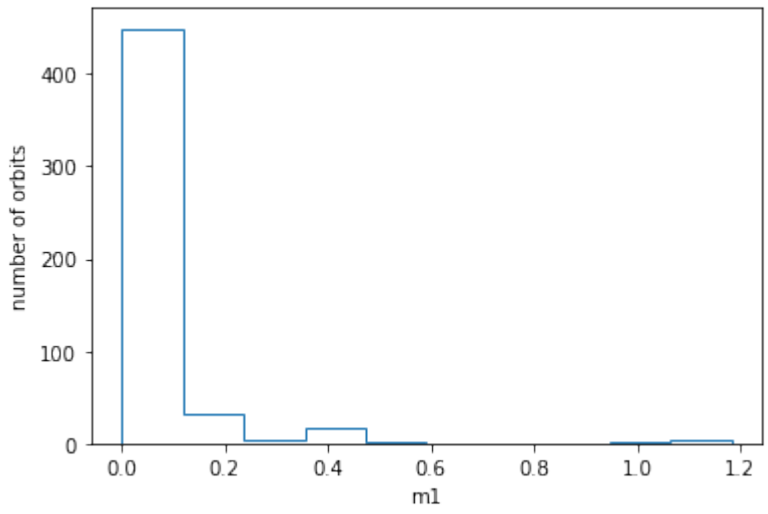
[7]:
```
m.run_sampler(total_orbits, burn_steps=burn_steps, thin=thin)
```

```
Starting Burn in
/home/sblunt/Projects/orbitize/orbitize/priors.py:354: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = -np.log((element_array*normalizer))
/home/sblunt/Projects/orbitize/orbitize/priors.py:354: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = -np.log((element_array*normalizer))
/home/sblunt/Projects/orbitize/orbitize/priors.py:463: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = np.log(np.sin(element_array)/normalization)
/home/sblunt/Projects/orbitize/orbitize/priors.py:463: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = np.log(np.sin(element_array)/normalization)
10/10 steps of burn-in complete
Burn in complete. Sampling posterior now.

Run complete
```

[7]:
```
<ptemcee.sampler.Sampler at 0x7f17c0d19b50>
```

Now we can plot the distribution of MCMC parameter of interest:

[8]:
```
accepted_m1 = m.results.post[:, lab['m1']]
plt.hist(accepted_m1,histtype='step')
plt.xlabel('m1'); plt.ylabel('number of orbits')
plt.show()
```



Saving Results over Extended MCMC Run

Sometimes our MCMC run will need to run for an extended period of time to let the walkers converge. To observe the convergence, we often need to see the walkers' progress along parameter space. We can save the sampler results periodically and keep running the sampler until convergence. To run for a greater number of steps and periodically save the results, we can create a for-loop and run for as many iterations as we'd like.

[9]:
```
filename = "{}/HD4747.csv".format(DATADIR)

total_orbits = 1000 # number of steps x number of walkers (at lowest temperature)
burn_steps = 10 # steps to burn in per walker
thin = 2 # only save every 10th step

# system parameters
num_secondary_bodies = 1
stellar_mass = 0.84 # [Msol]
plx = 53.18 # [mas]
mass_err = 0.04 # [Msol]
plx_err = 0.12 # [mas]

# MCMC parameters
num_temps = 5
num_walkers = 30
num_threads = 2 # or a different number if you prefer, e.g. mp.cpu_count()


my_driver = driver.Driver(
    filename, 'MCMC', num_secondary_bodies, stellar_mass, plx, mass_err=mass_err, plx_
↪err=plx_err,
    system_kwargs = {'fit_secondary_mass':True, 'tau_ref_epoch':0},
    mcmc_kwargs={'num_temps': num_temps, 'num_walkers': num_walkers, 'num_threads': num_
↪threads}
)

m = my_driver.sampler
```

We're now ready for the loop! The `results` object contains a `save_results` function which lets us save the results for our directory, and we will use the `load_results` object from `results` to access the data later. We also define the `n_iter` below to mark how many MCMC runs to save our within results.

NOTE: To avoid long convergence periods, we may initialize the walkers in a sphere around the global minima of the parameter space as outlined in our Modifying MCMC Initial Positions Tutorial.

[10]:
```
# file name to save as:
hdf5_filename = 'my_rv_posterior_%1d.hdf5'
```

[11]:
```
n_iter = 2 # number of iterations
for i in range(n_iter):
    # running the sampler:
    orbits = m.run_sampler(total_orbits, burn_steps=burn_steps, thin=thin)
    myResults = m.results
    hdf5_filename = 'my_rv_posterior_%1d.hdf5' % i
    myResults.save_results(hdf5_filename) # saves results object as an hdf5 file
```

```
Starting Burn in
```

```
/home/sblunt/Projects/orbitize/orbitize/priors.py:354: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = -np.log((element_array*normalizer))
/home/sblunt/Projects/orbitize/orbitize/priors.py:463: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = np.log(np.sin(element_array)/normalization)
/home/sblunt/Projects/orbitize/orbitize/priors.py:354: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = -np.log((element_array*normalizer))
/home/sblunt/Projects/orbitize/orbitize/priors.py:463: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = np.log(np.sin(element_array)/normalization)
```

```
10/10 steps of burn-in complete
Burn in complete. Sampling posterior now.

Run complete
Starting Burn in
```

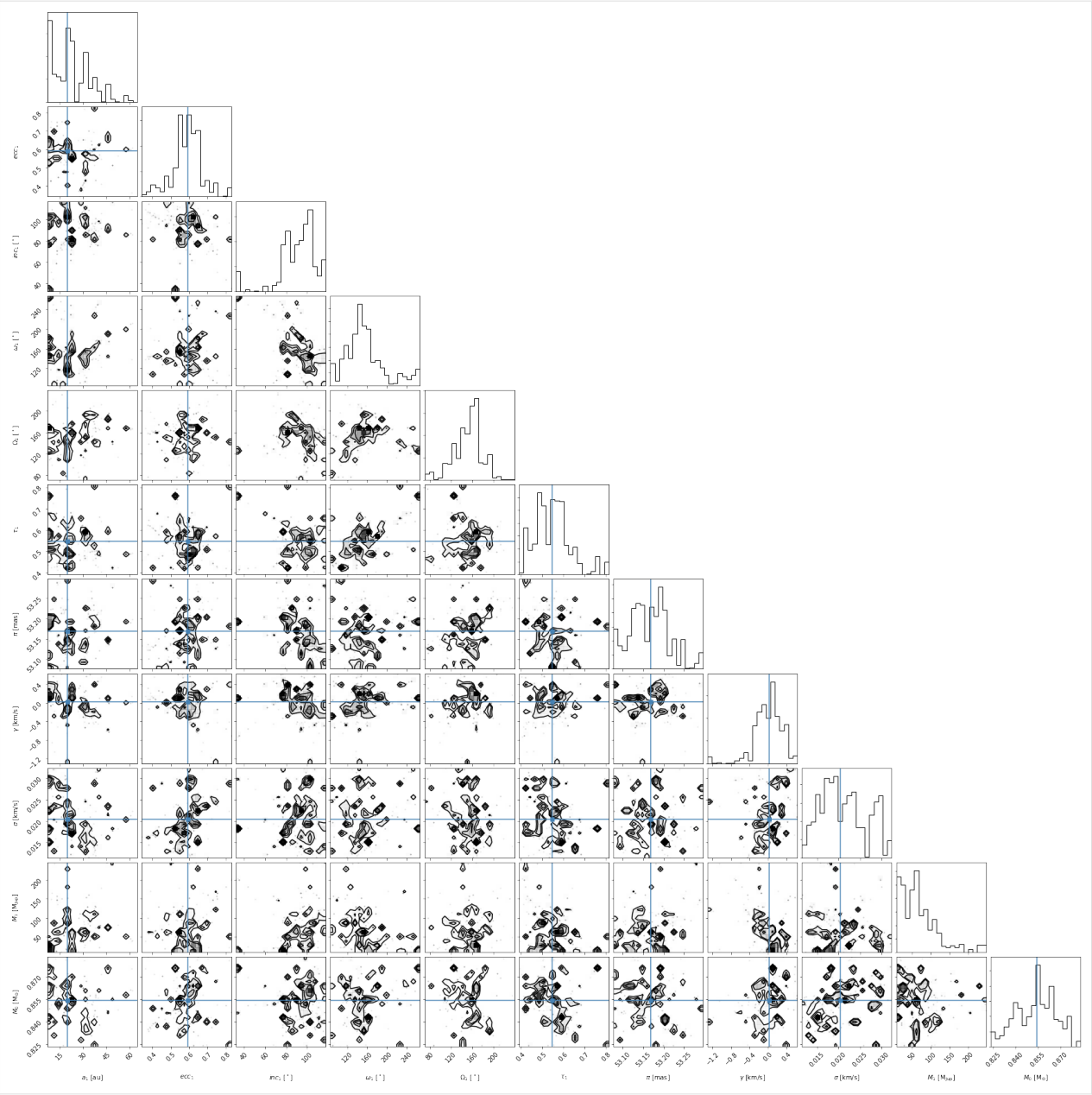
```
/home/sblunt/Projects/orbitize/orbitize/priors.py:354: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = -np.log((element_array*normalizer))
/home/sblunt/Projects/orbitize/orbitize/priors.py:354: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = -np.log((element_array*normalizer))
/home/sblunt/Projects/orbitize/orbitize/priors.py:463: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = np.log(np.sin(element_array)/normalization)
/home/sblunt/Projects/orbitize/orbitize/priors.py:463: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = np.log(np.sin(element_array)/normalization)
```

```
10/10 steps of burn-in complete
Burn in complete. Sampling posterior now.

Run complete
```

Plotting and Accesing Saved Results

We can plot the corner plot saved in the results object by following the steps in the Advanced Plotting Tutorial:

[12]:
```
median_values = np.median(myResults.post,axis=0) # Compute median of each parameter
range_values = np.ones_like(median_values)*0.95 # Plot only 95% range for each parameter
corner_figure_median_95 = myResults.plot_corner(
    range=range_values,
    truths=median_values
)
```



As illustrated in the plot above, MCMC needs more time to run. We only performed two iterations in the loop to demonstrate its useage, but with increased `n_iter`, the trendplots saved in the loop and the corner plot will show how the walkers converge to absolute extrema in parameter space.

To access the saved data, we can read it into a results object as shown in the MCMC Introduction Tutorial:

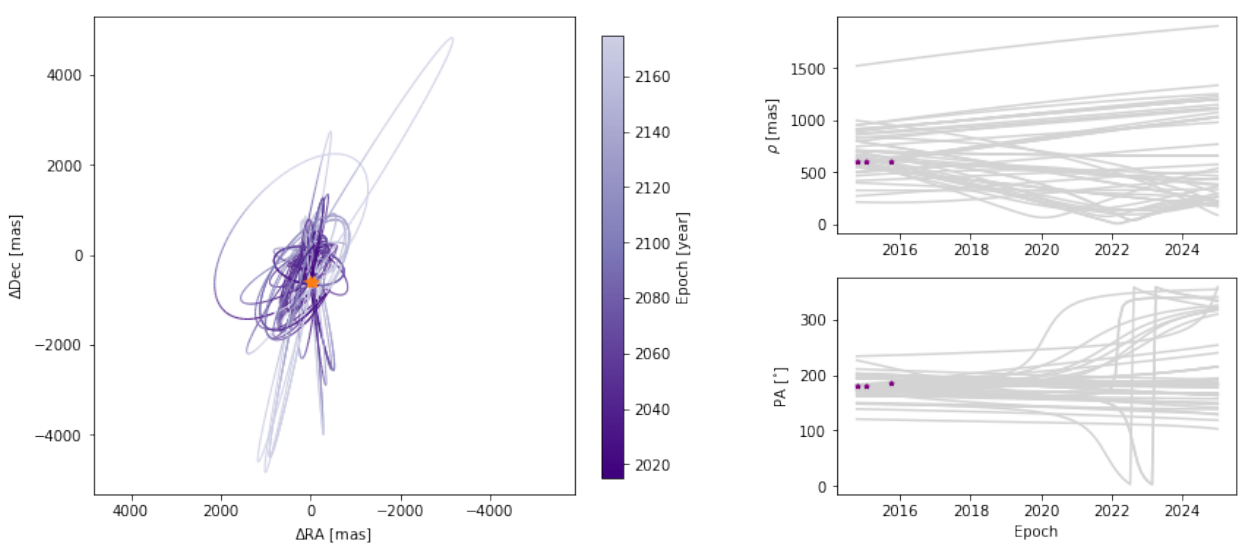
[13]:
```
from orbitize import results

loaded_results = results.Results() # Create blank results object for loading
loaded_results.load_results('my_rv_posterior_%1d.hdf5' % (n_iter-1))
```

To demonstrate use of the loaded results file above, we can use the saved results to plot our orbital plots:

[14]:
```
epochs = my_driver.system.data_table['epoch']
orbit_plot_fig = loaded_results.plot_orbits(
    object_to_plot = 1, # Plot orbits for the first (and only, in this case) companion
    num_orbits_to_plot= 50, # Will plot 50 randomly selected orbits of this companion
    start_mjd=epochs[0], # Minimum MJD for colorbar (here we choose first data epoch)
)
```

```
WARNING: ErfaWarning: ERFA function "d2dtf" yielded 1 of "dubious year (Note 5)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "dtf2d" yielded 1 of "dubious year (Note 6)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "utctai" yielded 1 of "dubious year (Note 3)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "taiutc" yielded 1 of "dubious year (Note 4)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "dtf2d" yielded 10 of "dubious year (Note 6)"␣
↪[astropy._erfa.core]
```

```
<Figure size 1008x432 with 0 Axes>
```

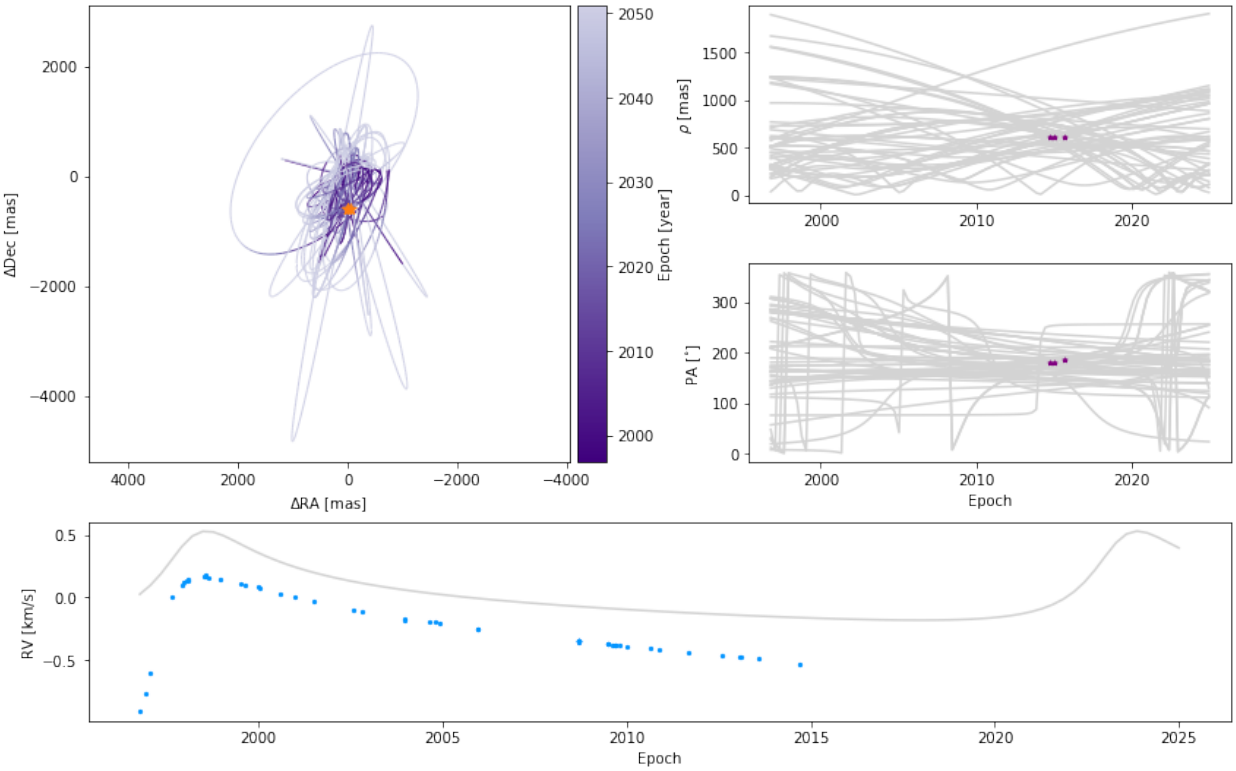


We can pass the `rv_time_series = True` argument in `plot_orbits` to display the RV time series plot as a third panel of `plot_orbits`:

[15]:
```
epochs = my_driver.system.data_table['epoch']

orbit_plot_fig = plot.plot_orbits(
    loaded_results,
    object_to_plot = 1, # Plot orbits for the first (and only, in this case) companion
    num_orbits_to_plot= 50, # Will plot 50 randomly selected orbits of this companion
    start_mjd=np.min(epochs), # Minimum MJD for colorbar (here we choose first data␣
↪epoch)
    show_colorbar = True,
    rv_time_series = True
)
orbit_plot_fig.savefig('HD4747_rvtimeseries_panelplot.png', dpi=250)
```

```
WARNING: ErfaWarning: ERFA function "d2dtf" yielded 1 of "dubious year (Note 5)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "dtf2d" yielded 1 of "dubious year (Note 6)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "utctai" yielded 1 of "dubious year (Note 3)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "taiutc" yielded 1 of "dubious year (Note 4)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "dtf2d" yielded 4 of "dubious year (Note 6)"␣
↪[astropy._erfa.core]
```

```
<Figure size 1008x432 with 0 Axes>
```



2.2.10 Multi-planet Fits

by Jason Wang (2020)

In `orbitize!`, we can fit for multiple planets in the system in the same way as fitting a single planet. Note that currently, `orbitize!` handles planet-planet gravitational interations with an nbody integrator solver. However, here, we only take into account star-planet interactions. For example, the distance between planet b and the star will change with the existance of planet c because planet c has some finite mass and perturbs the star from the system barycenter. Even without planet-planet scattering, one can approximately fit for dynamical masses this way. By default in `orbitize!`, we assume the planets are test particles (mass = 0), so there is no star-planet interactions. Later in this tutorial in the "Multiplanet Dynamical Mass" section, we will describe how to turn on this feature.

Multi-planet capabilities are generally handled under the hood, requiring not many modifications to the procedure for fitting a single planet. In this example, we will fit a few measurements of the HR 8799 b and c planets.

[1]:
```
import os
import orbitize
import orbitize.driver
```

In this tutorial, we will do an example with OFTI just because it is fast. However, in most cases, you will likely want to use MCMC as OFTI slows down significantly with multiple planets (even if each planet only has two astrometric data points as shown in the example). MCMC also requires longer run time typically, but it generally scales better than OFTI.

We follow the same steps as in the OFTI tutorial but set number of secondary bodies to 2 and read in a data file that contains astrometry for two planets in the system (in the example, a shortened version of the HR 8799 b and c astrometry). For MCMC, do the same thing as the single planet MCMC tutorial and make the same challenges as we have here with OFTI. In summary, all that needs to be done is to include both planets' measurements in the input data file and adjust the number of secondary bodies in the system.

[2]:
```
input_file = os.path.join(orbitize.DATADIR, "test_val_multi.csv")
my_driver = orbitize.driver.Driver(input_file, 'OFTI',
                                   2, # number of secondary bodies in system
                                   1.52, # total mass [M_sun]
                                   24.76, # total parallax of system [mas]
                                   mass_err=0.15,
                                   plx_err=0.64)
```

```
Converting ra/dec data points in data_table to sep/pa. Original data are stored in input_
↪table.
```

Next we run the sampler as usual:

[3]:
```
s = my_driver.sampler
orbits = s.run_sampler(1000)
```

With two planets, we have 2 sets of 6 orbtial parameters as well as the 2 system parameters (parallax and total mass). Our posterior, stored in `orbits`, is a (1000 x 14) array instead of the (1000 x 8) array had we fit a single planet.

As it gets confusing to track with index corresponds to which orbital parameter, we recommend you use the system.param_idx to index the parameters you are interested in. As a reminder, the abbreviations are: semi-major axis (sma), eccentricity (ecc), inclination (inc), argument of periastron (aop), position angle of nodes (pan), and epoch of periastron passage in fraction of the orbital period (tau). The 1 and 2 correspond to the two secondary bodies (in this case, HR 8799 b and HR 8799 c respectively)

[4]:
```
print(orbits[0])
print(orbits.shape)
print(s.system.param_idx)
```

```
[56.24156344  0.27898557  0.30103121  1.9027183   2.28889823  0.46425206
 66.25721525  0.58545346  1.68028001  4.79289978  2.23970056  0.33901155
 24.8454356   1.43000678]
(1000, 14)
{'sma1': 0, 'ecc1': 1, 'inc1': 2, 'aop1': 3, 'pan1': 4, 'tau1': 5, 'sma2': 6, 'ecc2': 7,
↪'inc2': 8, 'aop2': 9, 'pan2': 10, 'tau2': 11, 'plx': 12, 'mtot': 13}
```

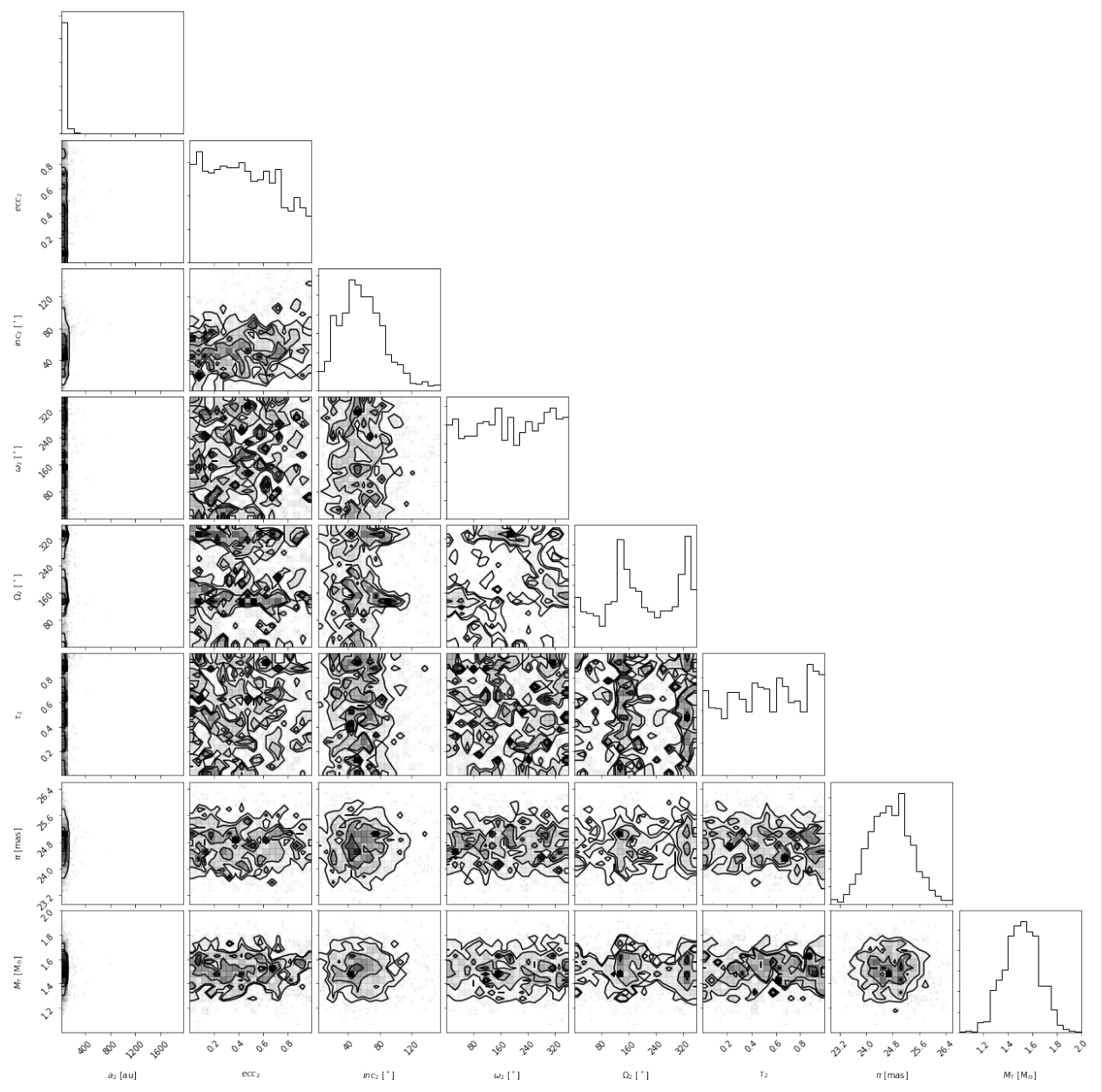
Plotting

We will go over briefly some considerations for visualizing multiplanet fits using the `orbitize!` API. For a more detailed guide on data visualization capabilities within orbitize, see the Orbitize plotting tutorial.

Corner Plot

Corner plots are slow when trying to plot too many parameters (the number of subplots scales as n^2 where n is the number of dimensions). It also is difficult to read a plot with too many subplots. For this reason, we recommend looking at particular parameter covariances, or breaking it up to have one corner plot for each planet. Again, use the notation in `system.param_idx` notation to grab the parameters you want.

[5]:
```
my_results = s.results
corner_figure = my_results.plot_corner(param_list=['sma2', 'ecc2', 'inc2', 'aop2', 'pan2
↪','tau2', 'plx', 'mtot'])
```



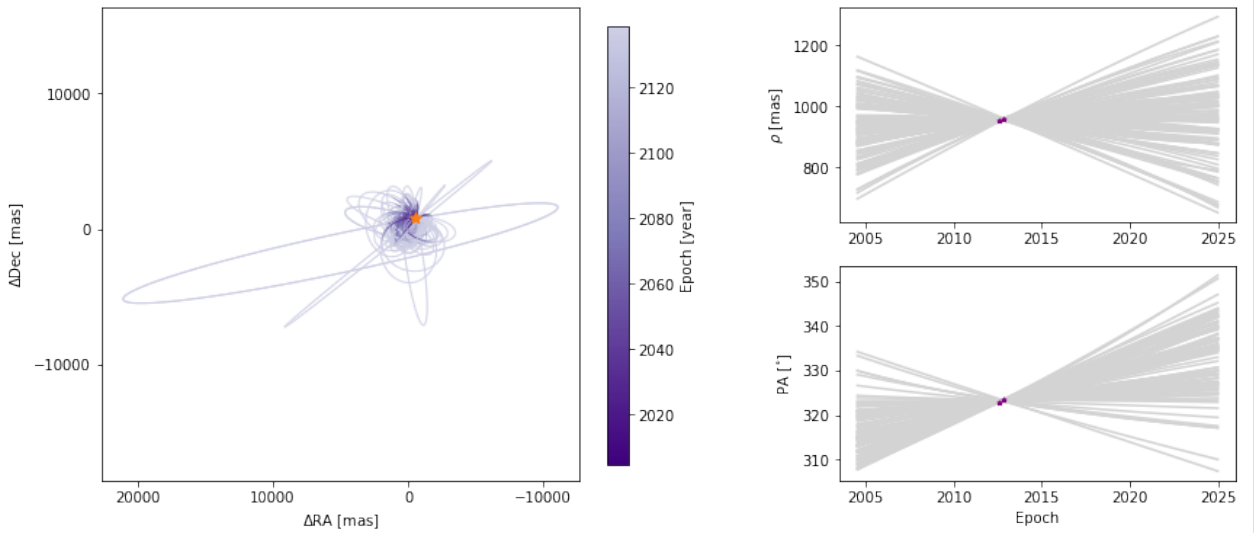
Orbit Plot

Currently, the orbit plotting tool in results class only plots the orbit of one body at a time. You can select which body you wish to plot.

[6]:
```
epochs = my_driver.system.data_table['epoch']

orbit_figure = my_results.plot_orbits(
    start_mjd=epochs[0], # Minimum MJD for colorbar (here we choose first data epoch),
    object_to_plot=2 # plot planet c
)
```

```
WARNING: ErfaWarning: ERFA function "d2dtf" yielded 1 of "dubious year (Note 5)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "dtf2d" yielded 1 of "dubious year (Note 6)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "utctai" yielded 1 of "dubious year (Note 3)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "taiutc" yielded 1 of "dubious year (Note 4)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "dtf2d" yielded 5 of "dubious year (Note 6)"␣
↪[astropy._erfa.core]
```

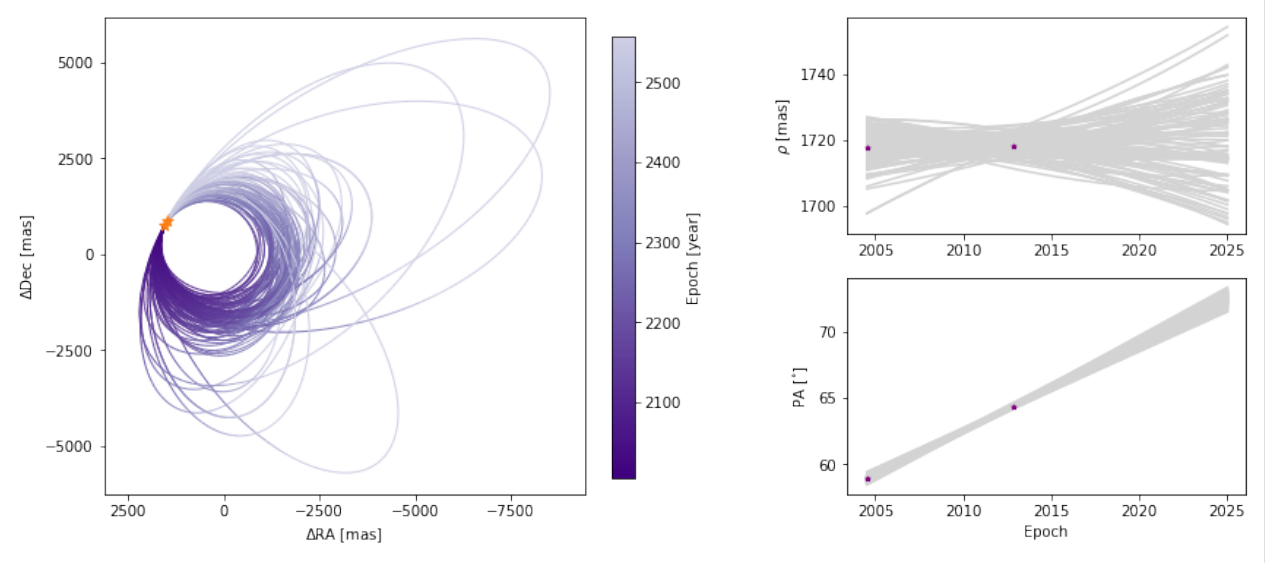
```
<Figure size 1008x432 with 0 Axes>
```



[7]:
```
orbit_figure = my_results.plot_orbits(
    start_mjd=epochs[0], # Minimum MJD for colorbar (here we choose first data epoch),
    object_to_plot=1 # plot planet b
)
```

```
WARNING: ErfaWarning: ERFA function "d2dtf" yielded 1 of "dubious year (Note 5)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "dtf2d" yielded 1 of "dubious year (Note 6)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "utctai" yielded 1 of "dubious year (Note 3)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "taiutc" yielded 1 of "dubious year (Note 4)"␣
↪[astropy._erfa.core]
WARNING: ErfaWarning: ERFA function "dtf2d" yielded 5 of "dubious year (Note 6)"␣
↪[astropy._erfa.core]
```

```
<Figure size 1008x432 with 0 Axes>
```



Multiplanet Dynamical Mass

In the case we want to fit dynamical masses, the procedure again remains unchanged as for a single planet system (as described in the RV MCMC Tutorial). The only thing to be aware of the extra parameters for the indiviudal masses of the components. Again, use the `system.param_idx` dictionary to keep track of the indices instead of trying to keep track of the ordering in one's head. Here, we will not demo a fit, but just show the increasing of parameters by 2 when fitting for the masses of the two secondary bodies.

Here, the new parameters are `m1` and `m2`, the masses of planets b and c. `m0` remains the mass of the star.

[8]:
```
input_file = os.path.join(orbitize.DATADIR, "test_val_multi.csv")
my_driver = orbitize.driver.Driver(input_file, 'MCMC',
                                   2, # number of secondary bodies in system
                                   1.52, # stellar mass [M_sun]
                                   24.76, # total parallax of system [mas]
                                   mass_err=0.15,
                                   plx_err=0.64,
                                   system_kwargs={'fit_secondary_mass' : True })
print(my_driver.system.param_idx)
print(len(my_driver.system.sys_priors))
```

```
{'sma1': 0, 'ecc1': 1, 'inc1': 2, 'aop1': 3, 'pan1': 4, 'tau1': 5, 'sma2': 6, 'ecc2': 7,
↪'inc2': 8, 'aop2': 9, 'pan2': 10, 'tau2': 11, 'plx': 12, 'm1': 13, 'm2': 14, 'm0': 15}
16
```

2.2.11 Using non-orbitize! Posteriors as Priors

By Jorge Llop-Sayson (2021)

This tutorial shows how to use posterior distribution from any source as `orbitize!` priors using a Kernel Density Estimator (KDE).

The user will need their posterior chains, consisiting of any number of correlated parameters, which will be used to get a KDE fit of the chains to be used as priors to `orbitize!`.

Once the priors are initialized the user can select their favorite fit method.

Read Data

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from orbitize import system, priors, basis, read_input, DATADIR
import pandas as pd

# Read RadVel posterior chain
pdf_fromRadVel = pd.read_csv(
    "{}/sample_radvel_chains.csv.bz2".format(DATADIR), compression="bz2", index_col=0
)

per1 = pdf_fromRadVel.per1  # Period
k1 = pdf_fromRadVel.k1  # Doppler semi-amplitude
secosw1 = pdf_fromRadVel.secosw1
sesinw1 = pdf_fromRadVel.sesinw1
tc1 = pdf_fromRadVel.tc1  # time of conj.

len_pdf = len(pdf_fromRadVel)
```

```
/Users/bluez3303/miniconda3/envs/python3.10/lib/python3.10/site-packages/scipy/__init__.
↪py:146: UserWarning: A NumPy version >=1.17.3 and <1.25.0 is required for this version␣
↪of SciPy (detected version 1.25.0
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```

The `quant_type` column displays the type of data each row contains: astrometry (radec or seppa), or radial velocity (rv). For astrometry, `quant1` column contains right ascension or separation, and the `quant2` column contains declination or position angle. For rv data, `quant1` contains radial velocity data in $\mathrm{km/s}$, while `quant2` is filled with `nan` to preserve the data structure. The table contains each respective error column.

We can now initialize the `Driver` class. MCMC samplers take time to converge to absolute maxima in parameter space, and the more parameters we introduce, the longer we expect it to take.

Format Data

In this example we use data from `RadVel`, so we want to change format to use it in `orbitize`.

[2]:
```
# From P to sma:
system_mass = 1  # [Msol]
system_mass_err = 0.01  # [Msol]
per_yr = per1 / 365.25
pdf_msys = np.random.normal(system_mass, system_mass_err, size=len_pdf)
sma_prior = (per_yr**2 * pdf_msys) ** (1 / 3)

# ecc from RadVel parametrization
ecc_prior = sesinw1**2 + secosw1**2

# little omega, i.e. argument of the periastron, from RadVel parametrization
w_prior = (
    np.arctan2(sesinw1, secosw1) + np.pi
)  # +pi bc of RadVel vs Orbitize convention
w_prior[w_prior >= np.pi] = w_prior[w_prior >= np.pi] - 2 * np.pi


def Tc_to_Tp(tc, per, ecc, omega):  # stolen from radvel
    f = np.pi / 2 - omega
    ee = 2 * np.arctan(np.tan(f / 2) * np.sqrt((1 - ecc) / (1 + ecc)))
    tp = tc - per / (2 * np.pi) * (ee - ecc * np.sin(ee))
    return tp


# tau, i.e. fraction of elapsed time of node passage, from RadVel parametrization
tp1 = Tc_to_Tp(tc1, per1, ecc_prior, w_prior)
tau_prior = basis.tp_to_tau(tp1, 55000, per1)

# m1 prior
K_0 = 28.4329
m1sini_prior = (
    k1
    / K_0
    * np.sqrt(1.0 - ecc_prior**2.0)
    * pdf_msys ** (2.0 / 3.0)
    * per_yr ** (1 / 3.0)
) * 1e-3
sini_prior = priors.SinPrior()
i_prior_samples = sini_prior.draw_samples(len_pdf)
m1_prior = m1sini_prior / np.sin(i_prior_samples)
```

We now have the orbital parameters that are accessible with RV: SMA, eccentricity, argument of periastron, and tau. Plus, given that we obtained `Msini` from RV, we draw random values for the inclination to get correlated values for `m1` and `inc`. We thus end in this case with a correlated 6-parameter set.

Initialize Priors

We initilize the System object to initialize the priors

[3]:
```
# Initialize System object which stores data & sets priors
data_table = read_input.read_file("{}/test_val.csv".format(DATADIR))  # read data
num_secondary_bodies = 1

system_orbitize = system.System(
    num_secondary_bodies,
    data_table,
    system_mass,
    1e1,
    mass_err=system_mass_err,
    plx_err=0.1,
    tau_ref_epoch=55000,
    fit_secondary_mass=True,
)

param_idx = system_orbitize.param_idx

print(param_idx)
```

```
{'sma1': 0, 'ecc1': 1, 'inc1': 2, 'aop1': 3, 'pan1': 4, 'tau1': 5, 'plx': 6, 'gamma_defrv
↪': 7, 'sigma_defrv': 8, 'm1': 9, 'm0': 10}
```

Let's initilize the KDE prior object with the default bandwidth.

[4]:
```
from scipy.stats import gaussian_kde

# The values go into a matrix
total_params = 6
values = np.empty((total_params, len_pdf))
values[0, :] = sma_prior
values[1, :] = ecc_prior
values[2, :] = i_prior_samples
values[3, :] = w_prior
values[4, :] = tau_prior
values[5, :] = m1_prior

kde = gaussian_kde(
    values, bw_method=None
)  # None indicates that the KDE bandwidth is set to default
kde_prior_obj = priors.KDEPrior(
    kde, total_params
)  # ,bounds=bounds_priors,log_scale_arr=[False,False,False,False,False,False])

system_orbitize.sys_priors[
    param_idx["sma1"]
] = kde_prior_obj  # priors.GaussianPrior(np.mean(sma_prior), np.std(sma_prior))#priors.
↪KDEPrior(gaussian_kde(values[0,:], bw_method=None),1)#kde_prior_obj#
system_orbitize.sys_priors[
    param_idx["ecc1"]
] = kde_prior_obj  # priors.GaussianPrior(np.mean(ecc_prior), np.std(ecc_prior))#kde_
```

(continues on next page)

(continued from previous page)

```
↪prior_obj#priors.GaussianPrior(np.mean(pdf_fromRadVel['e1']), np.std(pdf_fromRadVel['e1
↪']))#priors.KDEPrior(gaussian_kde(values[1,:], bw_method=None),1)#kde_prior_obj
system_orbitize.sys_priors[param_idx["inc1"]] = kde_prior_obj
system_orbitize.sys_priors[
    param_idx["aop1"]
] = kde_prior_obj  # priors.GaussianPrior(np.mean(w_prior), 0.1)#kde_prior_obj#kde_prior_
↪obj
system_orbitize.sys_priors[
    param_idx["tau1"]
] = kde_prior_obj  # priors.GaussianPrior(np.mean(tau_prior), 0.01)#np.std(tau_prior))
↪#kde_prior_obj#kde_prior_obj#priors.KDEPrior(gaussian_kde(values[4,:], bw_method=None),
↪1)#kde_prior_obj
system_orbitize.sys_priors[-2] = kde_prior_obj
```

We can plot the KDE fit against the actual distribution to see if the selected bandwidth is adecuate for the data.

[5]:
```
fig, axs = plt.subplots(2, 3, figsize=(11, 6))
# sma
axs[0, 0].hist(
    kde.resample(len_pdf)[0, :],
    100,
    density=True,
    color="b",
    label="Posterior Draw",
    stacked=True,
)
axs[0, 0].hist(
    sma_prior,
    100,
    density=True,
    color="r",
    histtype="step",
    label="Prior Draw",
    stacked=True,
)  # color='r')#
axs[0, 0].set_xlabel("SMA [AU]")
axs[0, 0].set_yticklabels([])
# ecc
axs[0, 1].hist(
    kde.resample(len_pdf)[1, :],
    100,
    density=True,
    color="b",
    label="Posterior Draw",
    stacked=True,
)
axs[0, 1].hist(
    ecc_prior,
    100,
    density=True,
    color="r",
    histtype="step",
```

(continues on next page)

(continued from previous page)

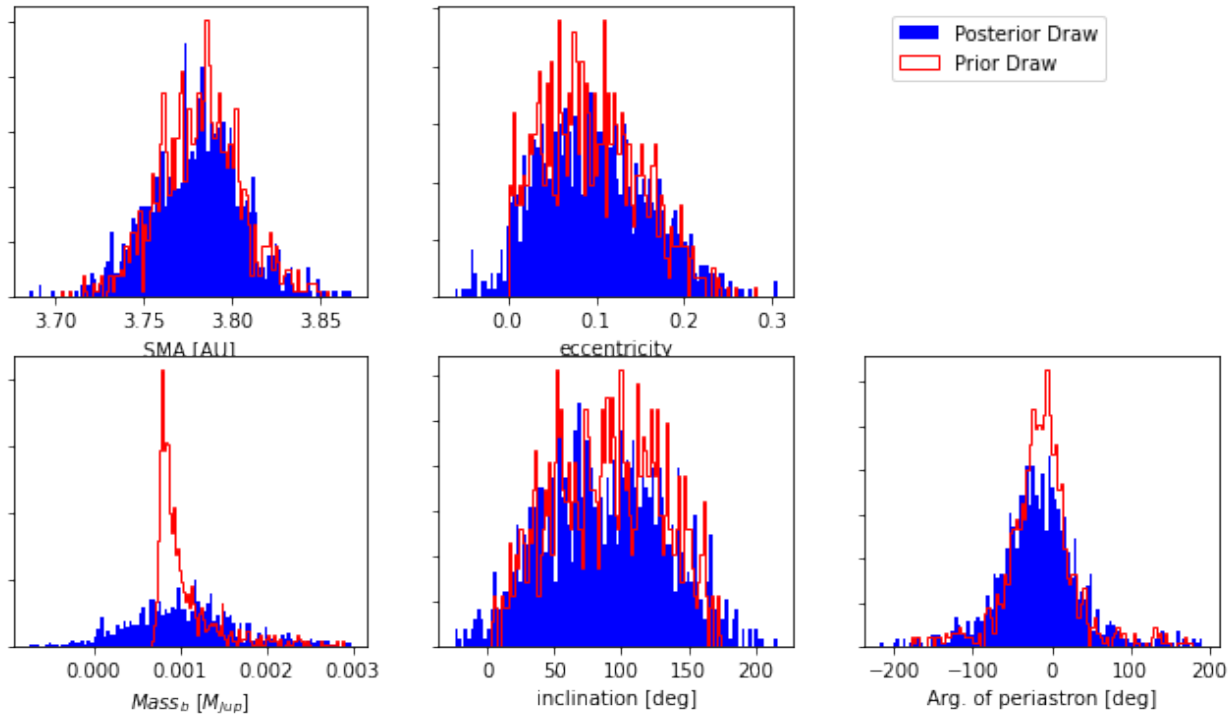
```
    label="Prior Draw",
    stacked=True,
)  # color='r')#
axs[0, 1].set_xlabel("eccentricity")
axs[0, 1].set_yticklabels([])
# mass
masskde_arr = kde.resample(len_pdf)[-1, :]
masskde_arr = masskde_arr[masskde_arr < 0.003]
m1_prior_draw = m1_prior[m1_prior < 0.003]
axs[1, 0].hist(
    masskde_arr, 100, density=True, color="b", label="Posterior Draw", stacked=True
)
axs[1, 0].hist(
    (m1_prior_draw),
    100,
    density=True,
    color="r",
    histtype="step",
    label="Prior Draw",
    stacked=True,
)  # color='r')#
axs[1, 0].set_xlabel("$Mass_b$ [$M_{Jup}$]")
axs[1, 0].set_yticklabels([])
# inc
axs[1, 1].hist(
    np.rad2deg(kde.resample(len_pdf)[2, :]),
    100,
    density=True,
    color="b",
    label="Posterior Draw",
    stacked=True,
)
axs[1, 1].hist(
    np.rad2deg(i_prior_samples),
    100,
    density=True,
    color="r",
    histtype="step",
    label="Prior Draw",
    stacked=True,
)  # color='r')#
axs[1, 1].set_xlabel("inclination [deg]")
axs[1, 1].set_yticklabels([])
# w
axs[1, 2].hist(
    np.rad2deg(kde.resample(len_pdf)[3, :]),
    100,
    density=True,
    color="b",
    label="Posterior Draw",
    stacked=True,
)
```

(continues on next page)

(continued from previous page)

```
axs[1, 2].hist(
    np.rad2deg(w_prior),
    100,
    density=True,
    color="r",
    histtype="step",
    label="Prior Draw",
    stacked=True,
)  # color='r')#
axs[1, 2].set_xlabel("Arg. of periastron [deg]")
axs[1, 2].set_yticklabels([])

axs[0, 1].legend(bbox_to_anchor=(1.25, 1), loc="upper left", ncol=1)
axs[0, 2].axis("off")
```

[5]: (0.0, 1.0, 0.0, 1.0)



As seen from the plot produced above, `m1` is not well fit by the KDE; the bandwidth selected (we selected the default) is too broad and the lower bound of the mass is not well reproduced.

Select the KDE bandwidth

A temptation to fix the problem presented above, the oversmoothing of the data, would be to pick a very narrow bandwith, one that reproduces perfectly the data. However, the data from our posterior chains is finite, and contains throughout the distribution peaks and valleys that would introduce artifacts in the KDE fit contaminating the prior probabilities.

[6]:
```
numtry_prior = 20
sma_arr = np.mean(sma_prior) * np.linspace(0.98, 1.02, numtry_prior)

# Initialize KDE with default bandwidth
kde1 = gaussian_kde(
    values, bw_method=None
)  # None indicates that the KDE bandwidth is set to default
# Initialize KDE with narrow bandwidth
bw2 = 0.15
kde2 = gaussian_kde(values, bw_method=bw2)

lnprior_arr1 = np.zeros((numtry_prior))
lnprior_arr2 = np.zeros((numtry_prior))
for idx_prior, sma in enumerate(sma_arr):
    lnprior_arr1[idx_prior] = kde1.logpdf(
        [
            sma,
            np.mean(ecc_prior),
            np.pi / 2,
            np.mean(w_prior),
            np.mean(tau_prior),
            np.mean(m1_prior),
        ]
    )
    lnprior_arr2[idx_prior] = kde2.logpdf(
        [
            sma,
            np.mean(ecc_prior),
            np.pi / 2,
            np.mean(w_prior),
            np.mean(tau_prior),
            np.mean(m1_prior),
        ]
    )

plt.figure(100)
plt.plot(sma_arr, lnprior_arr1, label="KDE BW = Default")
plt.plot(sma_arr, lnprior_arr2, label="KDE BW = {} (narrow)".format(bw2))
plt.xlabel("SMA [AU]")
plt.ylabel("log-prior probability")
plt.legend(loc="upper right")  # , fontsize='x-large')
```

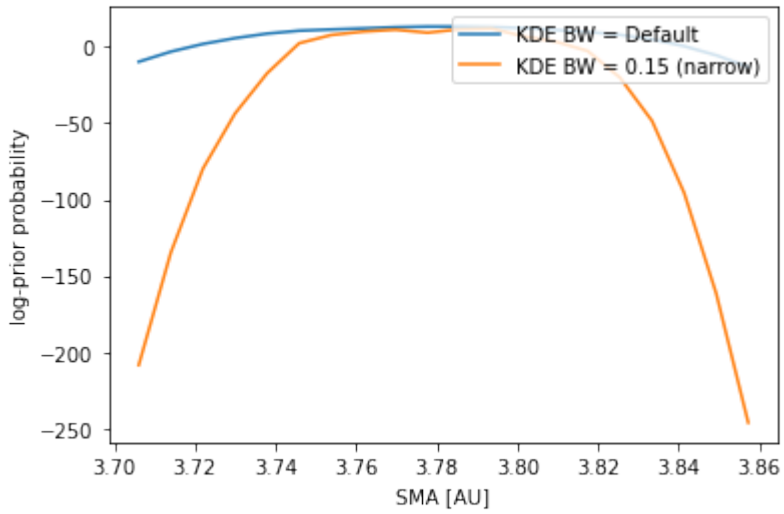
```
/var/folders/y8/lw5f1dcj04g4txq4y2znyc200000gn/T/ipykernel_15758/100748084.py:15:␣
↪DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated,␣
↪and will error in future. Ensure you extract a single element from your array before␣
↪performing this operation. (Deprecated NumPy 1.25.)
  lnprior_arr1[idx_prior] = kde1.logpdf(
```

(continues on next page)

(continued from previous page)

```
/var/folders/y8/lw5f1dcj04g4txq4y2znyc200000gn/T/ipykernel_15758/100748084.py:25:␣
↪DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated,␣
↪and will error in future. Ensure you extract a single element from your array before␣
↪performing this operation. (Deprecated NumPy 1.25.)
  lnprior_arr2[idx_prior] = kde2.logpdf(
```

[6]:
```
<matplotlib.legend.Legend at 0x7f9fe2a17d30>
```



The plot above ilustrates that we cannot go arbitrarily narrow for the KDE bandwidth.

A way of choosing the KDE bandwidth is: 1. Pick an acceptable change in the median and 68th interval limits of the KDE fit w.r.t the actual posterior distribution. This will set a maximum acceptable bandwidth. 2. Pick an acceptable variation of the log-prior probability when evaluating the priors for a SMA around the prior median SMA. This will set a minimum acceptable bandwidth.

For this we will loop over a set of bandwidths computing the median and 68th interval limits, and for each bandwidth we will compute the variation of log-prior with a set of SMAs around the prior median SMA: we will fit a Gaussian and the standard deviation of the residuals to the fit will be our cost function.

[7]:
```
from scipy.optimize import curve_fit


def gaussian(x, amp, cen, wid, bias):
    return amp * np.exp(-((x - cen) ** 2) / wid) + bias


numtry_bw = 10
kde_bw_arr = np.linspace(0.05, 0.1, numtry_bw)
diff_mean_arr = np.zeros((numtry_bw))
diff_p_arr = np.zeros((numtry_bw))
diff_m_arr = np.zeros((numtry_bw))
res_fit_prior2gauss = np.zeros((numtry_bw))

numtry_prior = 20
sma_arr = np.mean(sma_prior) * np.linspace(0.98, 1.02, numtry_prior)

for idx_bw, kde_bw in enumerate(kde_bw_arr):
```

(continues on next page)

(continued from previous page)

```
    kde = gaussian_kde(values, bw_method=kde_bw)

    # Check pdf
    lnprior_arr = np.zeros((numtry_prior))
    for idx_prior, sma in enumerate(sma_arr):
        lnprior_arr[idx_prior] = kde.logpdf(
            [
                sma,
                np.mean(ecc_prior),
                np.pi / 2,
                np.mean(w_prior),
                np.mean(tau_prior),
                np.mean(m1_prior),
            ]
        )

    # Quarentiles comparison
    masskde_arr = kde.resample(len_pdf)[5, :]
    masskde_arr = masskde_arr[masskde_arr < 0.004]
    masskde_quantiles = np.quantile(
        masskde_arr * 1000, [(1 - 0.68), 0.5, 0.5 + (1 - 0.68)]
    )
    massprior_quantiles = np.quantile(
        m1_prior_draw * 1000, [(1 - 0.68), 0.5, 0.5 + (1 - 0.68)]
    )
    diff_mean_arr[idx_bw] = (
        masskde_quantiles[1] - massprior_quantiles[1]
    ) / massprior_quantiles[1]
    diff_p_arr[idx_bw] = (
        (masskde_quantiles[1] - masskde_quantiles[0])
        - (massprior_quantiles[1] - massprior_quantiles[0])
    ) / massprior_quantiles[1]
    diff_m_arr[idx_bw] = np.abs(
        (
            (masskde_quantiles[2] - masskde_quantiles[1])
            - (massprior_quantiles[2] - massprior_quantiles[1])
        )
        / massprior_quantiles[1]
    )

    # fit to Gaussian
    n = len(sma_arr)
    mean = np.mean(sma_prior)
    sigma = 0.04  # note this correction
    best_vals, covar = curve_fit(
        gaussian,
        sma_arr,
        lnprior_arr,
        p0=[
            np.abs(np.max(lnprior_arr) - np.min(lnprior_arr)),
            np.mean(sma_prior),
            0.04,
```

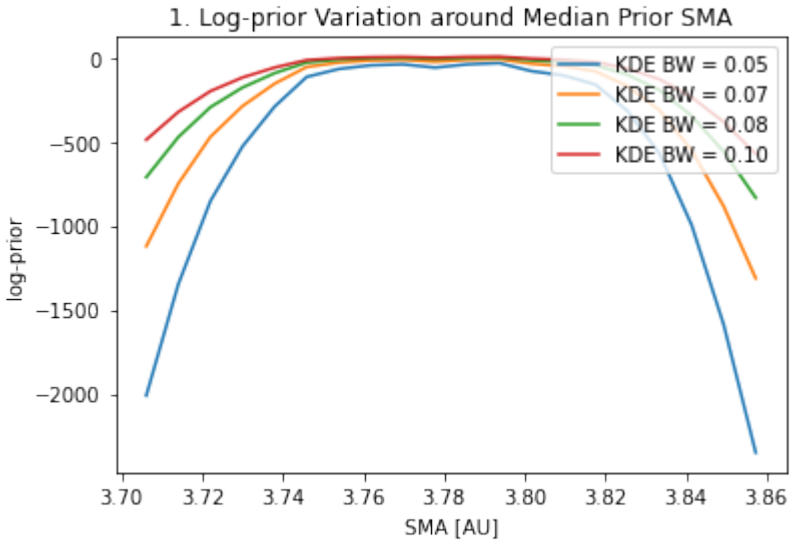
(continues on next page)

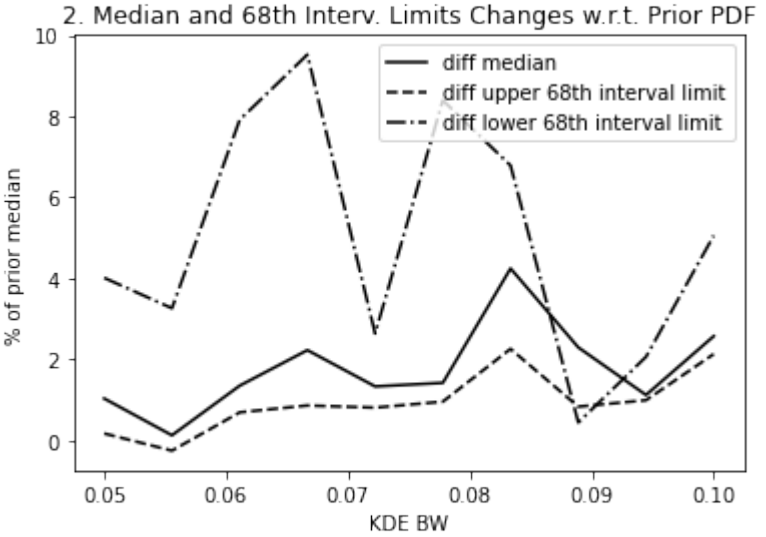
(continued from previous page)

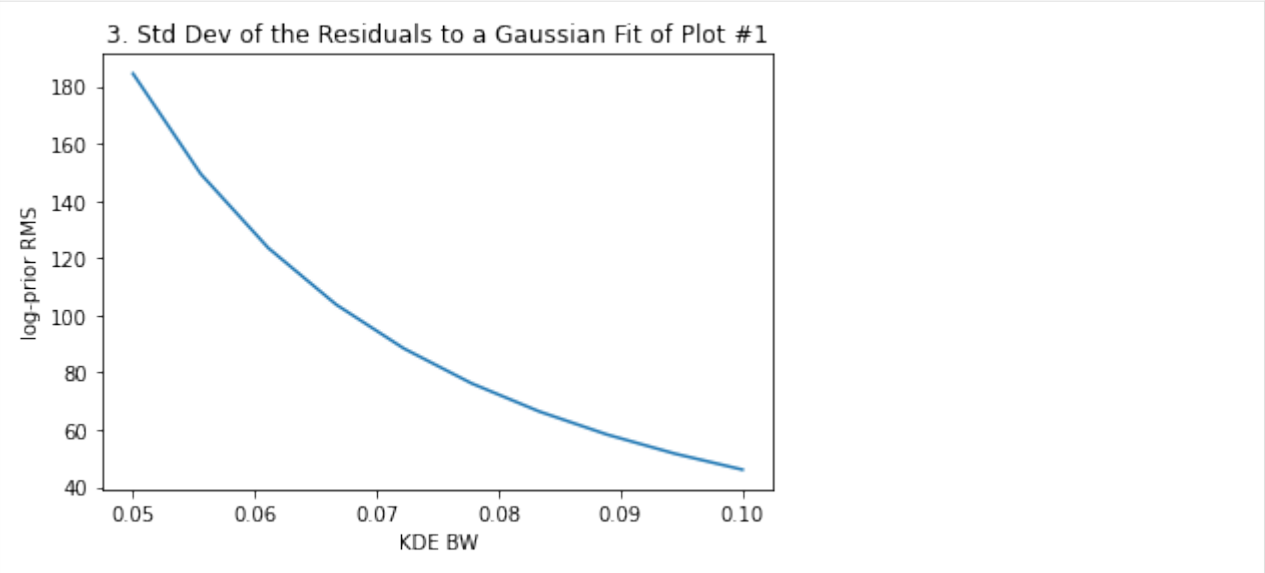
```
            np.mean(lnprior_arr) * 4,
        ],
    )  # [np.mean(lnprior_arr),mean,sigma])
    gauss_fit = gaussian(sma_arr, *best_vals)
    if idx_bw % 3 == 0:
        plt.figure(101)
        plt.plot(sma_arr, lnprior_arr, label="KDE BW = {:.2f}".format(kde_bw))

    res_fit_prior2gauss[idx_bw] = np.std(gauss_fit - lnprior_arr)
plt.figure(101)
plt.legend(loc="upper right")
plt.xlabel("SMA [AU]")
plt.ylabel("log-prior")
plt.title("1. Log-prior Variation around Median Prior SMA")

plt.figure(301)
plt.plot(kde_bw_arr, diff_mean_arr * 100, color="k", label="diff median")
plt.plot(
    kde_bw_arr,
    diff_p_arr * 100,
    color="k",
    linestyle="--",
    label="diff upper 68th interval limit",
)
plt.plot(
    kde_bw_arr,
    diff_m_arr * 100,
    color="k",
    linestyle="-.",
    label="diff lower 68th interval limit",
)
plt.xlabel("KDE BW")
plt.ylabel("% of prior median")
plt.legend(loc="upper right")  # , fontsize='x-large')
plt.title("2. Median and 68th Interv. Limits Changes w.r.t. Prior PDF")

plt.figure(302)
plt.plot(kde_bw_arr, res_fit_prior2gauss, label="")
plt.xlabel("KDE BW")
plt.ylabel("log-prior RMS")
plt.title("3. Std Dev of the Residuals to a Gaussian Fit of Plot #1")
```

```
/var/folders/y8/lw5f1dcj04g4txq4y2znyc200000gn/T/ipykernel_15758/2442406081.py:24:␣
↪DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated,␣
↪and will error in future. Ensure you extract a single element from your array before␣
↪performing this operation. (Deprecated NumPy 1.25.)
  lnprior_arr[idx_prior] = kde.logpdf(
/Users/bluez3303/miniconda3/envs/python3.10/lib/python3.10/site-packages/scipy/optimize/_
↪minpack_py.py:833: OptimizeWarning: Covariance of the parameters could not be estimated
  warnings.warn('Covariance of the parameters could not be estimated',
```

[7]:
```
Text(0.5, 1.0, '3. Std Dev of the Residuals to a Gaussian Fit of Plot #1')
```


1. Log-prior Variation around Median Prior SMA
KDE BW = 0.05
KDE BW = 0.07
KDE BW = 0.08
KDE BW = 0.10
log-prior
SMA [AU]


2. Median and 68th Interv. Limits Changes w.r.t. Prior PDF
diff median
diff upper 68th interval limit
diff lower 68th interval limit
% of prior median
KDE BW



The above plots #2 and #3 give us the information to select the most adecuate KDE bandwidth. For instance, if we decide that a ~3-4% difference in the median and 68th interval limits is acceptable, that sets a maximum bandwidth of ~0.9

To make a choice for the log-prior probability variation in the SMA range we picked, we can see what variation does the log-likelihood present for the same SMA range. For the same SMA range, we compute the log-likelihood of our model and see the peak-to-valley to assess what variation we want to allow for a narrow bandwidth.

Once you've chosen the bandwidth that best fits your posteriors you can use them as priors for a MCMC fit. Link to *MCMC tutotial*.

2.2.12 Fitting in different orbital bases

In this tutorial, we show how one can perform orbit-fits in different coordinate bases amongst the ones supported by `orbitize`. Currently fitting in different bases is only supported in MCMC, so we will use MCMC to perform an orbit-fit in an orbital basis distinct from the default one. For a general introduction to MCMC, be sure to check out the MCMC Introduction tutorial first!

The "standard" and "XYZ" bases

The default way to define an orbit in `orbitize` is through what we call the 'standard basis', which consists of eight parameters: semi-major axis (sma), eccentricity (ecc), inclination (inc), argument of periastron (aop), position angle of the nodes (pan), epoch of periastron expressed as a fraction of the period past a reference epoch (tau), parallax (plx) and total system mass (mtot). Each orbital element has an associated default prior; to see how to explore and modify these priors check out the Modifying priors tutorial.

An alternative way to define an orbit is through its position and velocity components in XYZ space for a given epoch; we will call this the 'XYZ basis'. The orbit is thus defined with the array (x, y, z, $\dot{x}$, $\dot{y}$,$\dot{z}$, plx, mtot), with position coordinates measured in AU and velocity components in km s^{-1}. In this basis, the sky-plane coordinates (x, y) are the separations of the planet relative to the primary, with the positive x and y directions coinciding with the positive RA and Dec directions, respectively. The z direction is the line-of-sight coordinate, such that movement in the positive z direction causes a redshift. The default priors are uniform all uniform.

Setting up Sampler in the XYZ basis

The easiest way to run an orbit-fit in an alternative orbital basis in `orbitize` is through the `orbitize.driver.Driver` interface. The process is exactly like initializing a regular `Driver` object, but setting the `fitting_basis` keyword to 'XYZ':

[1]:
```
import numpy as np

import orbitize
from orbitize import driver
import multiprocessing as mp

filename = "{}xyz_test_data.csv".format(orbitize.DATADIR) # a file with input in radec␣
↪since rn it only works for that

# system parameters
num_secondary_bodies = 1
system_mass = 1.75 # [Msol]
plx = 51.44 # [mas]
mass_err = 0.05 # [Msol]
plx_err = 0.12 # [mas]

# MCMC parameters
num_temps = 5
num_walkers = 20
num_threads = mp.cpu_count() # or a different number if you prefer


my_driver = driver.Driver(
    filename, 'MCMC', num_secondary_bodies, system_mass, plx, mass_err=mass_err, plx_
↪err=plx_err,
    mcmc_kwargs={'num_temps': num_temps, 'num_walkers': num_walkers, 'num_threads': num_
↪threads},
    system_kwargs={'fitting_basis': 'XYZ'}
)

s = my_driver.sampler
```

```
Converting ra/dec data points in data_table to sep/pa. Original data are stored in input_
↪table.
```

(Properly) initializing walkers in the XYZ basis

In the standard basis at this point we would be ready to use the `s.run_sampler` method to start the sampling, but with the XYZ basis we have to make sure that all our walkers are initialized in a valid region of parameter space. This is because randomly generated values of $(x, y, z, \dot{x}, \dot{y}, \dot{z})$ can result in unbound, invalid orbits with, for example, negative eccentricities (which is not cool). This can be easily corrected with the `s.validate_xyz_positions` method:

[2]:
```
s.validate_xyz_positions()
```

```
All walker positions validated.
```

```
/home/sblunt/Projects/orbitize/orbitize/basis.py:944: RuntimeWarning: invalid value␣
↪encountered in arccos
```

(continues on next page)

(continued from previous page)

```
  eanom = np.arccos(cos_eanom)
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/units/
↪quantity.py:481: RuntimeWarning: invalid value encountered in sqrt
  result = super().__array_ufunc__(function, method, *arrays, **kwargs)
```

After this is done, the sampler can be run and the results saved normally:

[3]:
```
total_orbits = 600 # number of steps x number of walkers (at lowest temperature)
burn_steps = 10 # steps to burn in per walker
thin = 2 # only save every 2nd step

s.run_sampler(total_orbits, burn_steps=burn_steps, thin=thin)
s.results.save_results('my_posterior.hdf5')
```

```
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
```

```
Starting Burn in
```

```
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
```

(continues on next page)

(continued from previous page)

```
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
```

```
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
```

```
10/10 steps of burn-in complete
Burn in complete. Sampling posterior now.
```

```
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
```

```
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
```

```
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
```

```
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
```

```
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
```

(continues on next page)

(continued from previous page)

```
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
/data/user/sblunt/miniconda3/envs/python3.7/lib/python3.7/site-packages/astropy/table/
↪column.py:1020: RuntimeWarning: invalid value encountered in greater
  result = getattr(super(), op)(other)
/home/sblunt/Projects/orbitize/orbitize/kepler.py:112: RuntimeWarning: invalid value␣
↪encountered in sqrt
  tanom = 2.*np.arctan(np.sqrt((1.0 + ecc)/(1.0 - ecc))*np.tan(0.5*eanom))
```

```
30/30 steps completed
Run complete
```

Loading and converting results

You can load the results as you normally would. The orbit posteriors are saved in the `results.post` attribute, and the basis you used for the fit in the `results.fitting_basis` attribute:

[4]:
```
myResults = orbitize.results.Results() # create empty Results object
myResults.load_results('my_posterior.hdf5')
print('The used basis for the fit was ', myResults.fitting_basis)
print('The posteriors are ', myResults.post)
```

```
Converting ra/dec data points in data_table to sep/pa. Original data are stored in input_
↪table.
The used basis for the fit was  XYZ
The posteriors are  [[-1.55895340e+01 -3.20352269e+01  9.38119986e+00 ... -7.46195838e-03
   5.15299823e+01  1.72400897e+00]
 [-1.56081092e+01 -3.20562773e+01  1.49810951e+00 ...  8.53387015e-02
   5.13883909e+01  1.73579787e+00]
 [-1.55612462e+01 -3.20801251e+01 -2.96308303e-01 ...  7.18261775e-01
   5.14206555e+01  1.76608956e+00]
 ...
 [-1.55671475e+01 -3.20089823e+01  3.62094122e+01 ...  3.72598271e-01
   5.15487665e+01  1.74432532e+00]
 [-1.55794884e+01 -3.20303979e+01  2.75912018e+01 ... -2.40407388e-01
   5.14895599e+01  1.78876637e+00]
 [-1.55837449e+01 -3.20532712e+01  1.93382300e+01 ... -1.24185021e-01
   5.14598433e+01  1.80483891e+00]]
```

Let's convert back to the good old standard basis:

[5]:
```
xyz_posterior = myResults.post

standard_posterior = myResults.system.basis.to_standard_basis(xyz_posterior)

print('My posterior in standard basis is ', standard_posterior)
```

```
My posterior in standard basis is  [[  7.89543529   6.92670011   4.84646336 ...   0.
↪36153132  -6.60018059
   1.52254522]
 [  1.00000429   1.00000659   0.96238341 ...   1.00053929   1.00014326
   0.9999977 ]
 [  1.43499876   1.22274162   0.79516331 ...   1.46937244   2.46248534
   1.23409917]
 ...
 [-15.56714749 -32.00898233  36.20941224 ...   0.37259827  51.54876654
   1.74432532]
 [-15.57948843 -32.03039793  27.59120179 ...  -0.24040739  51.48955988
   1.78876637]
 [-15.58374489 -32.0532712   19.33822997 ...  -0.12418502  51.45984325
   1.80483891]]
```

And we're done! Enjoy the XYZ basis.

2.2.13 Working with the Hipparcos Intermediate Astrometric Data (IAD)

Sarah Blunt (2021)

The Hipparcos IAD are notoriously annoying to work with, and several methods have been developed for incorporating them into orbit-fits. In this tutorial, we'll take you through the method outlined in Nielsen et al. (2020). In the near future, we'll add in other algorithms (in particular, that of Brandt et al.).

The purpose of this tutorial is not to convince you that one method is better than the other, but rather to teach you to use the method of Nielsen et al. I recommend reading through Section 3.1 of Nielsen et al. (2020) before diving into this tutorial.

If you want to skip right to "what's the syntax for doing an orbit-fit with the IAD," I suggest checking out the beta Pictoris end-to-end test I coded up here.

This tutorial will take you through: - Obtaining the IAD for your object(s). - Refitting the IAD (i.e. re-doing the fit the Hipparcos team did), which allows you to evaluate whether they are suitable for orbit-fitting. - Incorporating the IAD into your orbit-fit.

Part 1: Obtaining the IAD

`orbitize!` assumes the IAD are an updated version of the van Leeuwen (2007) re-reduction. You can learn about the data here, and download them here. Note that this download will take ~350 MB of space; if you prefer to to keep the data necessary for a few fits, you can download all the data and remove unnecessary files (or just send me an email). We've provided the IAD files for a few representative systems in this repository.

NOTE: if you prefer to use the DVD files, `orbitize!` can handle those as well. However, be warned that these files do not mark rejected scans.

The good news is that the hard part is actually getting the data. Once you have the file, `orbitize!` can read it in its raw form.

Part 2: Refitting the IAD

Once you have the IAD, the next step is to convince yourself that they're suitable for orbit-fitting (i.e. free of transcription errors, etc). Here's a handy function to do this (this should take a few mins to run, but if it's taking much longer, you can decrease the number of steps). This code reproduces the test at the end of Section 3.1 of Nielsen et al. (2020), so check that out for more information.

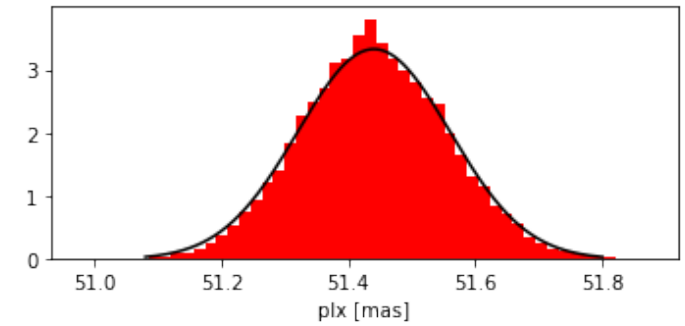
[2]:
```
import orbitize
import os
from datetime import datetime

from orbitize.hipparcos import nielsen_iad_refitting_test

# The Hipparcos ID of your target. Available on Simbad.
hip_num = "027321"

# Name/path for the plot this function will make
saveplot = "bPic_IADrefit.png"

# Location of the Hipparcos IAD file.
IAD_file = "{}H{}.d".format(orbitize.DATADIR, hip_num)

# These `emcee` settings are sufficient for the 5-parameter fits we're about to run,
```

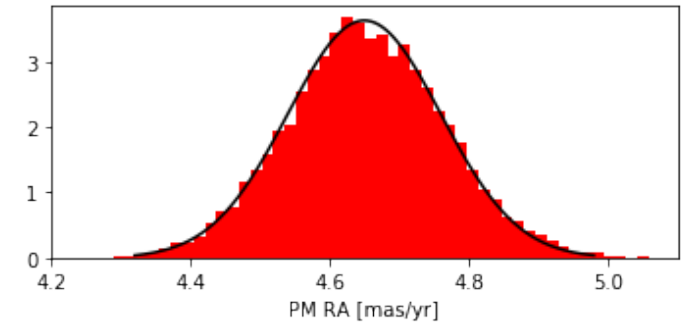
(continues on next page)

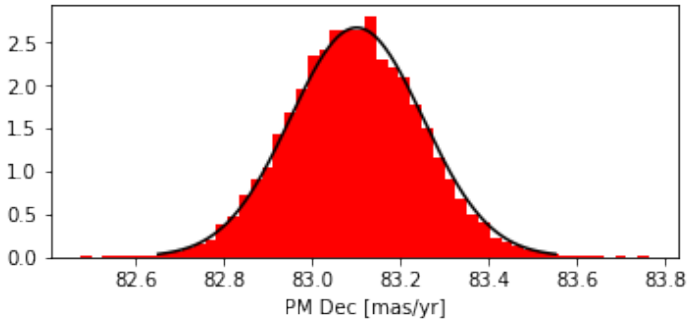
(continued from previous page)

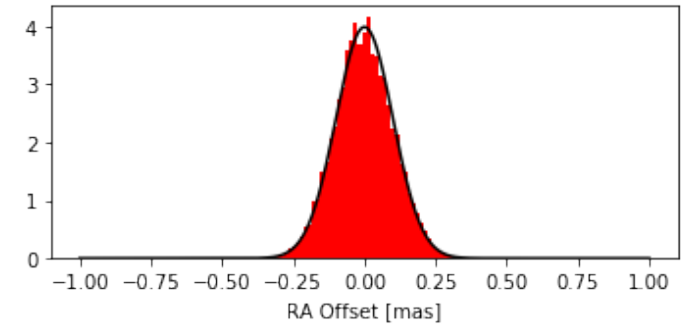
```
#   although I'd probably run it for 5,000-10,000 steps if I wanted to publish it.
burn_steps = 100
mcmc_steps = 100
start = datetime.now()

# run the fit
nielsen_iad_refitting_test(
    IAD_file,
    hip_num=hip_num,
    saveplot=saveplot,
    burn_steps=burn_steps,
    mcmc_steps=mcmc_steps,
)

end = datetime.now()
duration_mins = (end - start).total_seconds() / 60

print("Done! This fit took {:.1f} mins on my machine.".format(duration_mins))

# If you don't want to save the plot, you can run this line to remove it
_ = os.system("rm {}".format(saveplot))
```

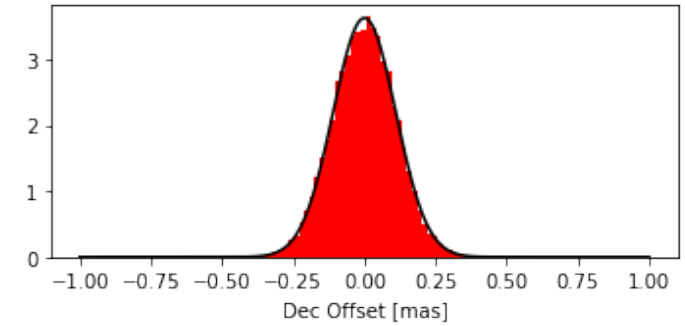
```
Go get a coffee. This will take a few mins! :)
```


0
1
2
3
51.0
51.2
51.4
51.6
51.8
plx [mas]


0
1
2
3
4.2
4.4
4.6
4.8
5.0
PM RA [mas/yr]


0.0
0.5
1.0
1.5
2.0
2.5
82.6
82.8
83.0
83.2
83.4
83.6
83.8
PM Dec [mas/yr]


0
1
2
3
4
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
RA Offset [mas]


0
1
2
3
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
Dec Offset [mas]

Part 3: Using the IAD in your Orbit-fit

Congrats, you've now reproduced a Hipparcos fit! The last thing to do is actually run your orbit-fit. Here's an example, also using the Gaia astrometric point from eDR3. This code snippet essentially repeats the beta Pictoris end-to-end test I coded up here.

[2]:
```
import os.path
import orbitize
from orbitize import read_input, hipparcos, gaia, system, priors, sampler

"""
As with most `orbitize!` fits, we'll start by reading in our data file. The
Hipparcos data file is kept separate; your main data file only needs to contain
the relative astrometry and RVs you're using in your fit.
"""

input_file = os.path.join(orbitize.DATADIR, "betaPic.csv")
data_table = read_input.read_file(input_file)

"""
Next, we'll instantiate a `HipparcosLogProb` object.
"""
num_secondary_bodies = 1  # number of planets/companions orbiting your primary
hipparcos_number = "027321"  # (can look up your object's Hipparcos ID on Simbad)
hipparcos_filename = os.path.join(
    orbitize.DATADIR, "H027321.d"
)  # location of your IAD data file

betaPic_Hip = hipparcos.HipparcosLogProb(
    hipparcos_filename, hipparcos_number, num_secondary_bodies
)

"""
Next, instantiate a `GaiaLogProb` object.
"""
betapic_edr3_number = 4792774797545800832
betaPic_Gaia = gaia.GaiaLogProb(betapic_edr3_number, betaPic_Hip, dr="edr3")

"""
Next, we'll instantiate a `System` object, a container for all the information
relevant to the system you're fitting.
"""

m0 = 1.75  # median mass of primary [M_sol]
plx = 51.5  # [mas]
fit_secondary_mass = True  # Tell orbitize! we want to get dynamical masses
#  (not possible with only relative astrometry).
mass_err = 0.01  # we'll overwrite these in a sec
plx_err = 0.01

betaPic_system = system.System(
    num_secondary_bodies,
    data_table,
```

(continues on next page)

(continued from previous page)

```
    m0,
    plx,
    hipparcos_IAD=betaPic_Hip,
    gaia=betaPic_Gaia,
    fit_secondary_mass=fit_secondary_mass,
    mass_err=mass_err,
    plx_err=plx_err,
)

"""
If you'd like to change any priors from the defaults (given in Blunt et al. 2020),
do it like this:
"""

# set uniform parallax prior
plx_index = betaPic_system.param_idx["plx"]
betaPic_system.sys_priors[plx_index] = priors.UniformPrior(plx - 1.0, plx + 1.0)

# set uniform primary mass prior
m0_index = betaPic_system.param_idx["m0"]
betaPic_system.sys_priors[m0_index] = priors.UniformPrior(1.5, 2.0)
```

```
INFO: Query finished. [astroquery.utils.tap.core]
```

Finally, set up and run your MCMC!

[3]:
```
"""
These are the MCMC parameters I'd use if I were publishing this fit.
This would take a while to run (takes about a day on my machine).
"""
# num_threads = 50
# num_temps = 20
# num_walkers = 1000
# num_steps = 10000000 # n_walkers x n_steps_per_walker
# burn_steps = 10000
# thin = 100

"""
Here are some parameters you can use for the tutorial. These chains will not
be converged.
"""

num_threads = 1
num_temps = 1
num_walkers = 100
num_steps = 10  # n_walkers x n_steps_per_walker
burn_steps = 10
thin = 1

betaPic_sampler = sampler.MCMC(
    betaPic_system,
    num_threads=num_threads,
    num_temps=num_temps,
```

(continues on next page)

(continued from previous page)

```
    num_walkers=num_walkers,
)
betaPic_sampler.run_sampler(num_steps, burn_steps=burn_steps, thin=thin)
```

```
Starting Burn in
```

```
/home/sblunt/Projects/orbitize/orbitize/priors.py:354: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = -np.log((element_array*normalizer))
/home/sblunt/Projects/orbitize/orbitize/priors.py:463: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = np.log(np.sin(element_array)/normalization)
```

```
10/10 steps of burn-in complete
Burn in complete. Sampling posterior now.

Run complete
```

[3]:
```
<emcee.ensemble.EnsembleSampler at 0x7f2f2e3a2650>
```

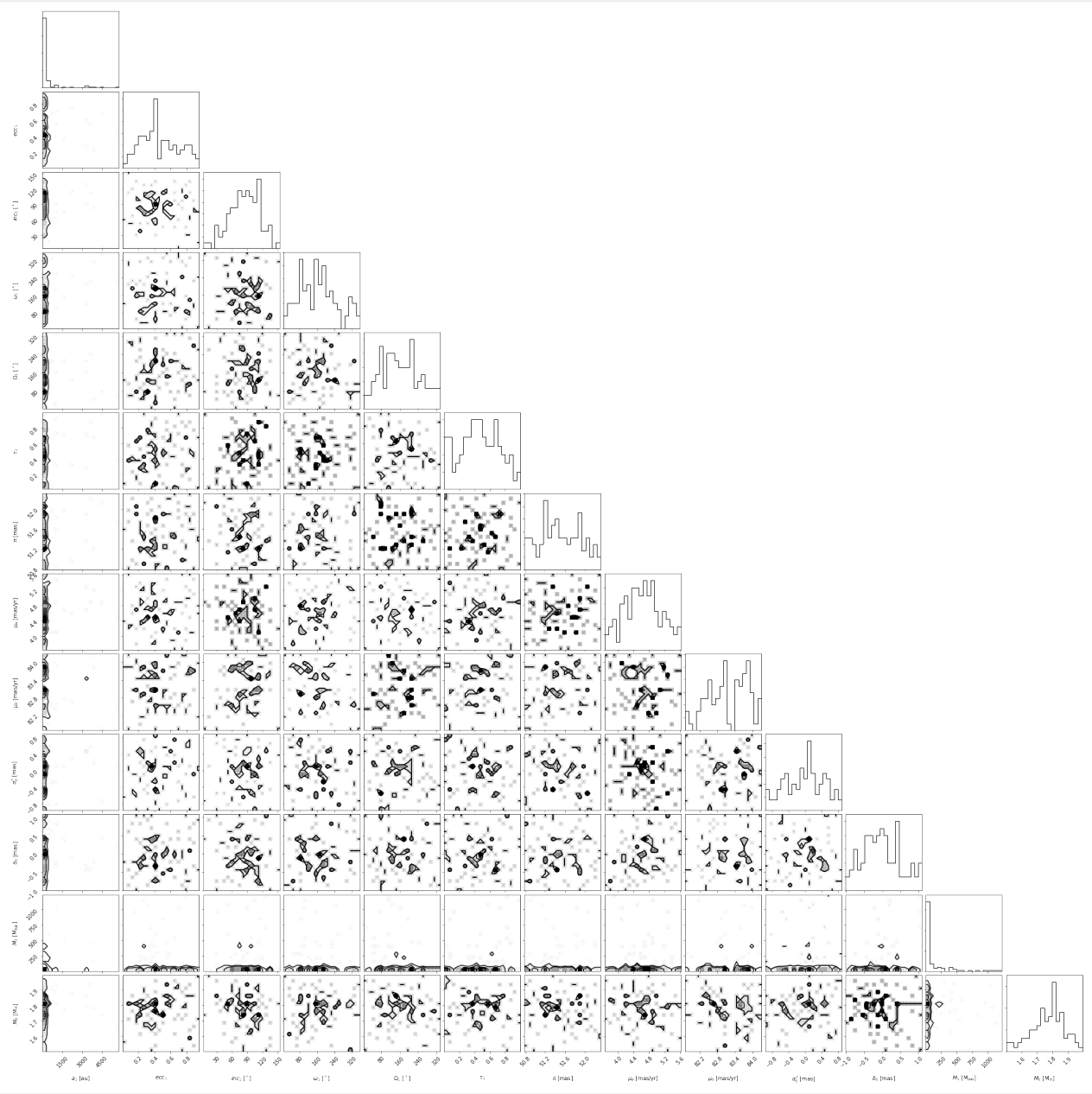
[4]:
```
# make corner plot
fig = betaPic_sampler.results.plot_corner()
```

```
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
```

(continues on next page)

(continued from previous page)

```
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
```



2.2.14 Fitting the Hipparcos-Gaia Catalog of Acclerations (HGCA)

Jason Wang (2023)

We will demonstrate how to fit the stellar absolute astrometry from the Hipparcos-Gaia Catalog of Accelerations (HGCA; Brandt 2021) to help constrain the mass and orbital parameters of the companion. The implementation of fitting the HGCA here is based on Brandt et al. 2019, but utilizes the DR3 version of the HGCA.

Difference to fitting IAD directly

This method is an alternative to fitting the Hipparcos IAD and Gaia astrometry. Instead of fitting the individual epochs of Hipparcos data, which also includes needing to fit for the position, proper motion, and parallax of the star, we are only fitting for two differential proper motions: the proper motion difference between Hipparcos and the change in position between Hipparcos and Gaia; the proper motion difference between Gaia and the change in position between Hipparcos and Gaia. This simplifies the fit, but also ignores any detectable curvature in the stellar astrometry seen by Hipparcos. For planet companion cases, the acceleration should be well within the noise of the Hipparcos measurements.

The benefit of using this technique is that the errors should be more robust. The HGCA catalog inflates the error bars from the Hipparcos and Gaia measurements on a global scale to match the true observed scatter in the data. Additionally, there are bad epochs in the Hipparcos IAD that may not be removed.

We encourage users to try both to see how similar or different the results are, as the two methods utilize the same underlying data.

Obtain the necessary data

While `orbitize!` will automatically download the HGCA catalog, you need to obtain two other data files to reconstruct the Hipparcos and Gaia observations for your star. These files tell us when and in what orientation did Hipparcos and Gaia take data of the star for the forward modeling that happens in `orbitize!`.

1. The Hipparcos IAD file of the star. Follow the section in the Hipparocs IAD tutorial on how to obtain this file for your star of interest.
2. The anticipated Gaia epochs and scan directions in a CSV file that is obtained from the Gaia Observation Forecast Tool (GOST). For GOST, after entering the target name and resolving its coordinates, use 2014-07-26T00:00:00 as the start date. For the end date, use 2016-05-23T00:00:00 for DR2 and 2017-05-28T00:00:00 for EDR3. You probably will be fitting the EDR3. The output CSV file is all you need.

Setup the `HGCALogProb` Object

The user just needs to setup the `orbitize.gaia.HGCALogProb` object, which will handle the rest of the HGCA modeling under the hood. To setup the `HGCALogProb` object, we also need to create an `orbitize.hipparcos.HipparcosLogProb` instance, but it will not actually be used in the fitting. It is simply used to handle reading in the Hipparcos IAD to maximize code reuse.

In the example below, we setup the HGCA fitting for beta Pictoris (HIP 27321).

[1]:
```
import os
from orbitize import DATADIR, hipparcos, gaia

# the necessary input data for beta Pic is part of the orbitize! example data!
iad_filepath = os.path.join(DATADIR, "HIP027321.d")
gost_filepath = os.path.join(DATADIR, "gaia_edr3_betpic_epochs.csv")

# Create the HGCA and helper Hipparcos object
# we're just going to assume one planet for simplicity
hipparcos_lnprob = hipparcos.HipparcosLogProb(iad_filepath, 27321, 1)
hgca_lnprob = gaia.HGCALogProb(27321, hipparcos_lnprob, gost_filepath)
```

```
Using HGCA catalog stored in /home/sblunt/Projects/orbitize/orbitize/example_data/HGCA_
↪vEDR3.fits
```

Setup orbit fit

After you do this step, the rest of the orbit fit setup is basically the same as a standard `orbitize!` orbit fit. The only differences is that you need to pass the `HGCALogProb` object into the system class (but not the `HipparcosLogProb` because the Hipparcos data is already being handled in HGCA). We also recommend you explicitly set up the system to fit for the dynamical mass of the companions in the system (this should happen automatically with the inclusion of HGCA data, but we recommend being explicit so it is clear what you are fitting for).

We will also note that that the default prior on the companion mass is a log-uniform prior between 1e-6 and 2 solar masses. If you want a more restricted mass, now is also the time to adjust that. We present one example here, but refer to the Modifying Priors tutorial for further details.

[2]:
```
from orbitize import read_input, system, priors
import astropy.units as u

# read in relative astrometry
astrometry_filepath = os.path.join(DATADIR, "betaPic.csv")
data_table = read_input.read_file(astrometry_filepath)

# set up the system, passing in hgca_lnprob and setting it fit dynamical mass
stellar_mass = 1.75
stellar_mass_err = 0.05
plx = 51.44
plx_err = 0.12

this_system = system.System(
    1,
    data_table,
    stellar_mass,
    plx,
    mass_err=stellar_mass_err,
    plx_err=plx_err,
    fit_secondary_mass=True,
    gaia=hgca_lnprob,
)

# adjust the prior on mass to be log uniform between 1 and 50 Jupiter masses
mjup = u.Mjup.to(u.Msun)
this_system.sys_priors[this_system.param_idx["m1"]] = priors.LogUniformPrior(
    mjup, 50 * mjup
)
```

Run orbitize!

From here onwards, everything is the same as an regular `orbitize` fit, so we refer to reader to the other tutorials. We recommend using the MCMC sampler, since the orbits are generally too constrained for OFTI. To pick the right number of temperatures, walkers, steps for MCMC, check out papers that performed similar fits (e.g., Section 3.1 of GRAVITY Collaboration et al. (2020) and Section 3.1 of Hinkley et al. (2023)).

[3]:
```
from orbitize import sampler

# MCMC parameters
# for demonstration purposes only. You will need to increase these likely
```

(continues on next page)

(continued from previous page)

```
n_temps = 2
n_walkers = 50
n_threads = 1
burn_steps = 1
total_orbits = 100 * n_walkers

# create the sampler, run it, and save posteriors
this_sampler = sampler.MCMC(this_system, n_temps, n_walkers, n_threads)

this_sampler.run_sampler(total_orbits, burn_steps=burn_steps, thin=10)

this_sampler.results.save_results("demo_hgca.hdf5")
```

```
Starting Burn in
```

```
/home/sblunt/Projects/orbitize/orbitize/priors.py:354: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = -np.log((element_array*normalizer))
/home/sblunt/Projects/orbitize/orbitize/priors.py:463: RuntimeWarning: invalid value␣
↪encountered in log
  lnprob = np.log(np.sin(element_array)/normalization)
```

```
Burn in complete. Sampling posterior now.
100/100 steps completed
Run complete
```

Plot proper motions from the orbit fit over the HGCA observations

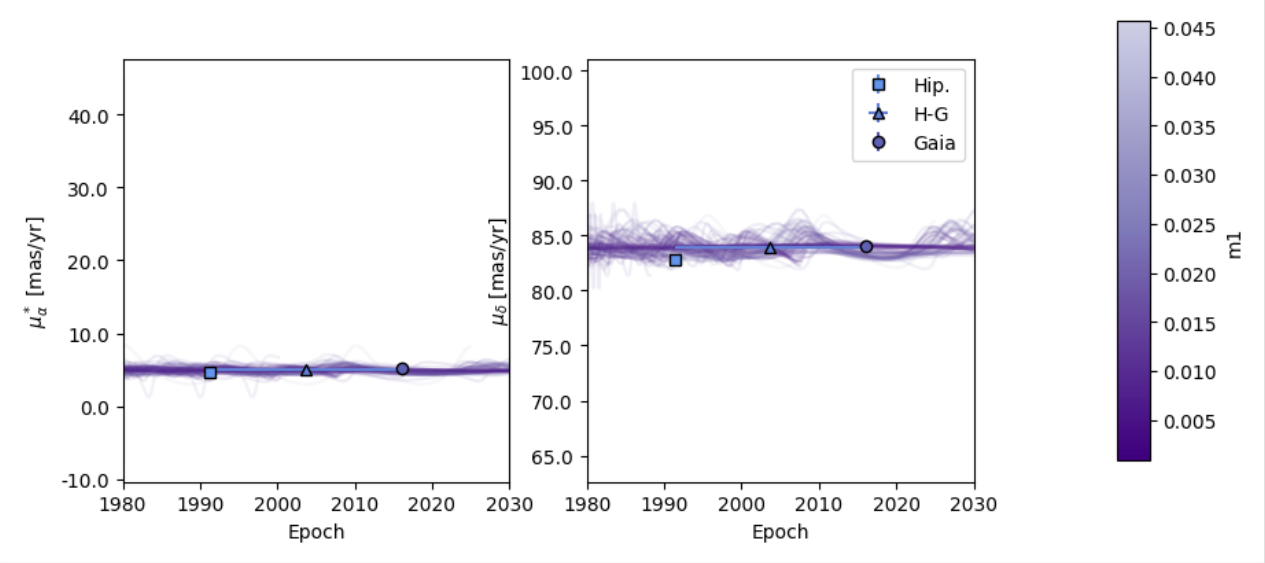
William Balmer (2023)

Now, you can plot the result of your fit using the `orbitize.plot.plot_propermotion` function directly, or it's wrapper within the `sampler.results` object. The function is similar to `orbitize.plot.plot_orbits` that you may already be familiar with. These plots show the H-G proper motion (with a large x-axis error bar) in addition to the two "real" proper motion measurements from Hipparchos and Gaia.

Note that the HGCALogprob fits for the time-averaged proper motions, and not the instantaneous proper motions that are shown in this visualization. This is a subtle point, but important, because the data points show over the orbits here are not exactly what is driving the MCMC solver towards the most likely orbits.

[4]:
```
figure = this_sampler.results.plot_propermotion(
    cbar_param="m1", alpha=0.1, periods_to_plot=3
)
```

```
Important Note of Caution: the orbitize! implementation of the HGCA
 fits for the time-averaged proper motions, and not the instantaneous proper
 motions that are being plotted here. This plot is provided only for the
 purpose of an approximate check on the fit.
```



2.2.15 Fitting Arbitrary Absolute Astrometry

by Sarah Blunt (2023)

This tutorial walks you through using orbitize! to perform a fit on arbitary absolute astrometry. By "arbitrary," I mean astrometry not taken by Gaia or Hipparcos (which orbitize! has dedicated modules for; see the HGCA and Hipparcos IAD tutorials). Let's imagine we have astrometry for a single star derived from wide-field images taken over several years, and we want to combine these data with measurements from Hipparcos. We are going to perform a fit to jointly constrain astrometric parameters (parallax and proper motion) and orbital parameters of a secondary companion.

This tutorial will take you through: - formatting absolute astrometry measurements for input into orbitize! - setting up an orbit fit incorporating these measurements

This tutorial assumes the following prerequities: - Using the Hipparcos IAD

Input Data Format

Following Nielsen et al 2020 (see the Hipparcos IAD tutorial), orbitize! defines astrometric data points as offset from the *reported Hipparcos position* at the *reported Hipparcos epoch*. Let's start by defining an `orbitize.hipparcos.Hipparcos` object, which holds onto information from the Hipparcos mission observations of our object of interest. I'm going to use beta Pictoris as an example since you already have that IAD file in your orbitize! distribution. See the IAD tutorial for info on how to download the data for your object.

[1]:
```
from orbitize import hipparcos, DATADIR

hip_num = "027321"  # beta Pic

# Location of the Hipparcos IAD file.
IAD_file = "{}H{}.d".format(DATADIR, hip_num)

# The HipparcosLogProb object needs to know how many companions are in your fit
# in order to compute likelihood. There are 2 known planets around beta Pic, but let's
# keep it simple for the tutorial
num_secondary_bodies = 1
```

(continues on next page)

(continued from previous page)

```
betaPicHipObject = hipparcos.HipparcosLogProb(IAD_file, hip_num, num_secondary_bodies)
```

Generally, when you're deriving (or using published) absolute astrometry, it will be in the form 82 02 14.35787 (J2000). However, `orbitize!` expects astrometry to be input *relative* to the Hipparcos position. Our friends at `astropy` have made these calculations very easy to do! Here's an example:

[2]:
```
from astropy.coordinates import SkyCoord
from astropy import units as u
import numpy as np

# let's imagine our data look like this:
datapoints = ["05 47 17.123456 -51 03 59.123456", "05 47 17.234567 -51 03 59.234567"]
data_epochs = ["2020.1234", "2020.2345"]
num_datapoints = len(datapoints)

hipparcos_coordinate = SkyCoord(
    betaPicHipObject.alpha0, betaPicHipObject.delta0, unit=(u.deg, u.deg)
)

raoffs = np.zeros(num_datapoints)
deoffs = np.zeros(num_datapoints)
for i in range(num_datapoints):
    my_data_coordinate = SkyCoord(datapoints[i], unit=(u.hourangle, u.deg))

    # take difference between reported Hipparcos position and convert to mas
    raoff, deoff = hipparcos_coordinate.spherical_offsets_to(my_data_coordinate)

    # n.b. orbitize! expects raw ra offsets, NOT multiplied by cos(delta0). Don't
    # multiply by cos(delta0) here.
    raoffs[i] = raoff.to(u.mas).value
    deoffs[i] = deoff.to(u.mas).value

print(raoffs, deoffs)
```

```
[ 377.81305615 1425.17609269] [1044.81957171  933.70290544]
```

Sweet! These absolute astrometry points are now suitable for an orbitize! input file. You can add them to an existing file with other types of data (relative astrometry and RVs) and/or fit them on their own. Here's what the data file for our two points would look like:

[3]:
```
from pandas import DataFrame
from astropy.time import Time

df_orbitize = DataFrame(Time(data_epochs, format="decimalyear").mjd, columns=["epoch"])

# this line tells orbitize! "these measurements are astrometry of the primary"
df_orbitize["object"] = 0

df_orbitize["raoff"] = raoffs
df_orbitize["deoff"] = deoffs

df_orbitize["deoff_err"] = 123.4  # error on the declination measurement, in mas
```

(continues on next page)

(continued from previous page)

```
df_orbitize["raoff_err"] = 123.4  # error on the RA measurement, in mas

df_orbitize.to_csv("data_for_orbit_fit.csv", index=False)
df_orbitize
```

[3]:

	epoch	object	raoff	deoff	deoff_err	raoff_err
0	58894.1644	0	377.813056	1044.819572	123.4	123.4
1	58934.8270	0	1425.176093	933.702905	123.4	123.4

Setting up & Running Your Fit

The hard part is over– we have formatted our input data! `orbitize!` will now function the same as any other fit. Behind the scenes, `orbitize!` will automatically recognize that you have inputted absolute astrometry, and set up a fit that includes position, parallax, and proper motion terms as free parameters. Observe:

[4]:
```
from orbitize import read_input, system, priors, sampler
import os

data_table = read_input.read_file("data_for_orbit_fit.csv")

fit_secondary_mass = True  # tell orbitize! we want to get dynamical masses
m0 = 1
plx = 1

# this sets up a joint fit of Hipparcos time series data and the absolute astrometry
# from the data table we just created.
betaPicSystem = system.System(
    num_secondary_bodies,
    data_table,
    m0,
    plx,
    hipparcos_IAD=betaPicHipObject,
    fit_secondary_mass=fit_secondary_mass,
)

# change any priors you want to:
plx_idx = betaPicSystem.param_idx["plx"]
betaPicSystem.sys_priors[plx_idx] = priors.UniformPrior(10, 15)

# run the fit!
tutorialSampler = sampler.MCMC(betaPicSystem)
# tutorialSampler.run_sampler(you_choose, burn_steps=you_choose)

# clean up
os.system("rm data_for_orbit_fit.csv")
```

[4]:
```
0
```

2.3 Frequently Asked Questions

Here are some questions we get often. Please suggest more by raising an issue in the Github Issue Tracker.

What does this orbital parameter mean?

We think the best way to understand the orbital parameters is to see how they affect the orbit visually. Play around with this interactive orbital elements notebook (you'll need to run on your machine).

What is and how is it related to epoch of periastron?

We use to define the epoch of periastron as we do not know when periastron will be for many of our directly imaged planets. A detailed description of how is related to other quantities such as t_p is available:

2.3.1 τ and Time of Periastron

Here, we will discuss what exactly is τ, the parameter `orbitize!` uses to parametrize the epoch of periastron, and how it is related to other quantities of the epoch of periastron in the literature.

Time of Periastron and Motivation for τ

The time (or epoch) of periastron is an important quantity for describing an orbit. It defines when the two orbiting bodies are closest to one another (i.e., when a planet is closest to its star). In many papers in the literature, the epoch of periastron is described by t_p, which is literally a date at which periastron occurs. This is a very important date because we use this date to anchor our orbit in time.

The value of t_p is well constrained when we know we observed periastron, which is often the case for radial velociy or transiting exoplanets when the orbital periods are short and our data covers a full orbital period. In those cases, we know approximately when t_p should be in time, so it is easy to define prior bounds for it. However, in the case of direct imaging, many of our companions have orbital periods that are orders of magnitude larger than the current orbital coverage of the data where we do not really know if the next periastron is in years, decades, centuries, or even millennia. This is the motivation for τ.

Definition of τ

τ is a dimentionless quantity between 0 and 1 defined with respect to a reference epoch t_{ref}. For a planet that has a t_p and an orbital period (P), then we define τ as:

$$\tau = \frac{t_p - t_{ref}}{P}.$$

Because τ is always between 0 and 1, it is easy to figure out the bounds of τ whereas if the orbital period is highly uncertain, it may be difficult to put bounds on t_p that would encompass all allowable bound orbits.

Relation to t_p

As seen in the above equation, it is relatively straightforward to covert between orbital parameter sets that use τ and t_p. You just need to know the orbital period and reference epoch. In `orbitize!`, both the `System` class and the `Results` class has the attribute `tau_ref_epoch` which stores t_{ref}, so there should always be a convenient way to grab this number. By default, we use $t_{ref} = 58849$ MJD.

One thing to note that is a given orbit has only a single valid τ, but that an orbit can be defined by many t_p, since the orbit is periodic. Thus, $t_p + P$ is another valid time of periastron.

We also provide some helper functions to covert between t_p and τ

[1]:
```
import numpy as np
import orbitize.basis

# How to get orbital period in the orbitize! standard basis
sma = 9 # au, semi-major axis
mtot = 1.2 # Solar masses, total mass
period = np.sqrt(sma**3/mtot) # years, period

tau = 0.2
tau_ref_epoch = 58849

# convert tau to tp
tp = orbitize.basis.tau_to_tp(tau, tau_ref_epoch, period)

print(tp)

# convert tp back to tau
tau2 = orbitize.basis.tp_to_tau(tp, tau_ref_epoch, period)

print(tau2)

# convert tau to tp, but pick the first tp after MJD = 0
tp_new = orbitize.basis.tau_to_tp(tau, tau_ref_epoch, period, after_date=0)

print(tp_new)
```

```
60649.50097715886
0.2000000000000002
6634.471662393138
```

Relation to Mean Anomaly

The mean anomaly (M) of an orbit describes the current orbital phase of a planet. M goes from 0 to 2π, and M = 0 means the planet is at periastron. Unlike t_p and τ which describe the epoch of periastron, M describes the current position of the planet in its orbit.

To compute M of a planet at some time t, we have provided the following helper function:

[2]:
```
# Use the orbit defined in the previous example

t = 60000 # time in MJD when we want to know the M of the particle

M = orbitize.basis.tau_to_manom(t, sma, mtot, tau, tau_ref_epoch)

print(M)

# now compute M for periastron
M_peri = orbitize.basis.tau_to_manom(tp, sma, mtot, tau, tau_ref_epoch)

print(M_peri)
```

```
5.829874251150844
1.3951473992034527e-15
```

Why is the default prior on inclination a sine prior?

Our goal with the default prior is to have an isotropic distribution of the orbital plane. To obtain this, we use inclination and position angle of the ascending node (PAN) to define the orbital plane. They actually coorespond to the two angles in a spherical coordinate system: inclinaion is the polar angle and PAN is the azimuthal angle. Becuase of this choice of coordinates, there are less orbital configurations near the poles (when inclination is near 0 or 180), so we use a sine prior to downweigh those areas to give us an isotropic distribution. A more detailed discussion of this is here:

2.3.2 Defining the orbital plane with i and Ω

(Jason Wang, 2021)

Here we discuss how the orbit plane orientation is defined. We also encourage you to play around with this interactive orbital elements notebook to get a feel for the orbital elements. Also note that we refer to the values of the angles in degrees when discussing them but all angles are in radians in `orbitize!`.

Inclination (i) and Position angle of the Ascending Node (PAN; Ω) define the plane of the orbit in the sky. Inclination describes the tilt of the orbital plane relative to the plane in the sky, and PAN describes the rotation of the line of nodes, which is the intersection between the orbital plane and the plane of the sky.

An intuitive way to think about it is in terms of spherical coordinates. i is equivalent to θ (range from 0 to 180 deg), and Ω is equivalent to φ (range from 0 to 360 deg) in common spherical notation. The sphere in this case is tilted, with one of the poles pointed towards us.

Why do we use a sine prior on inclination?

You might also hear about using a uniform prior in $\cos(i)$, which is an equivalent statement. We use a sine prior on inclination because our ultimate goal is to have an isotropic prior on the orbital plane, and due to how we choose coordiantes, this becomes a sine prior on inclination. We can go through some math/visual explaination, but the easiest is perhaps seeing the distribution of orbital plane orientations on a sphere. We use the orbital plane normal (perpendicular vector from the orbit plane) to define the orientation of the plane in 3D space. The normal points to a single point on a unit sphere. Isotropic distributions will uniformly cover the unit sphere.

In the exercise below, if we randomly draw orbital plane orientations using either an uniform or sine prior for inclination, you'll see that the uniform prior causes more points to cluster near the poles, whereas the sine prior is more uniform. Note that generally the edges of the sphere look darker merely due to a viewing angle effect.

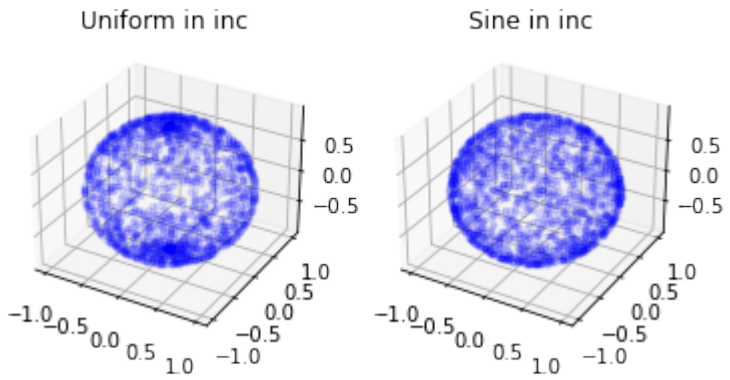
[4]:
```
import numpy as np
import matplotlib.pylab as plt
from orbitize.priors import UniformPrior, SinPrior
%matplotlib inline

def spherical_to_xyz(theta, phi, rho=1):
    """
    Transformation with theta and phi in radians
    """
    z = rho * np.cos(theta)
    x = rho * np.sin(theta) * np.cos(phi)
    y = rho * np.sin(theta) * np.sin(phi)
    return x,y,z

fig = plt.figure()

# try a uniform distribution in both
```

(continues on next page)

(continued from previous page)

```
ax1 = fig.add_subplot(121, projection='3d')

inc_uni_prior = UniformPrior(0, np.pi)
pan_uni_prior = UniformPrior(0, 2*np.pi)

incs_uni = inc_uni_prior.draw_samples(2000)
pan_uni = pan_uni_prior.draw_samples(2000)

x_uni, y_uni, z_uni = spherical_to_xyz(incs_uni, pan_uni)
ax1.plot(x_uni, y_uni, z_uni, 'b.', alpha=0.1)
ax1.set_title("Uniform in inc")

# try a sine distribution  for inclination
ax2 = fig.add_subplot(122, projection='3d')

inc_sine_prior = SinPrior()

incs_sine = inc_sine_prior.draw_samples(2000)

x_uni, y_uni, z_uni = spherical_to_xyz(incs_sine, pan_uni)
ax2.plot(x_uni, y_uni, z_uni, 'b.', alpha=0.1)
ax2.set_title("Sine in inc")
```

[4]:
```
Text(0.5, 0.92, 'Sine in inc')
```



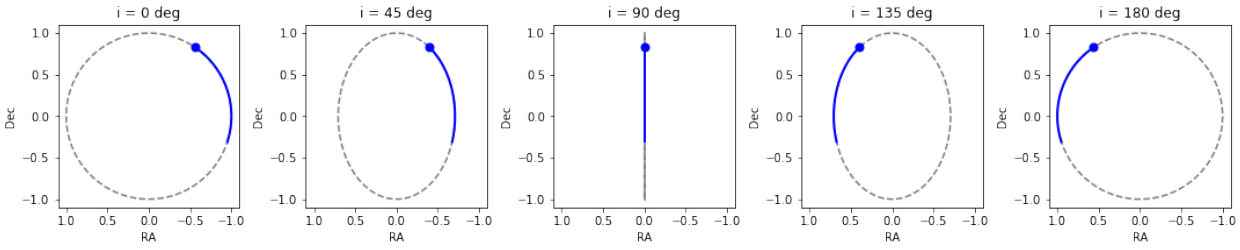
What do the values of i mean?

Inclination in `orbitize!` is defined to go from 0 to 180 degrees. Some other places use -90 to 90 instead, but -90 to 0 is the same as 90 to 180. There are actual a couple of quick things to learn about the orbit from the value of inclination. Here is a summary:

- $i = 0°$: orbit is face-on and body orbits counterclockwise in the sky (North-up, East-left)
- $0° < i < 90°$: orbit is inclined and body orbits counterclockwise in the sky (North-up, East-left)
- $i = 90°$: orbit is viewed edge-on
- $90° < i < 180°$: orbit is inclined and body orbits clockwise in the sky (North-up, East-left)
- $i = 180°$: orbit is face-on and body orbits clockwise in the sky (North-up, East-left)

Here are some examples below:

[5]:
```
from orbitize.kepler import calc_orbit

sma = 1
ecc, pan, aop, plx, mtot = 0, 0, 0, 1, 1
tau = 0.3

incs = np.radians([0, 45, 90, 135, 180])

all_eps = np.linspace(0, 365.25, 200)
arc_eps = np.linspace(0, 75, 75)

fig = plt.figure(figsize=(15,4))
for i, inc in enumerate(incs):
    ax = fig.add_subplot(1, 5, i+1)

    all_ras, all_decs, _ = calc_orbit(all_eps, sma, ecc, inc, aop, pan, tau, plx, mtot,␣
↪tau_ref_epoch=0)
    ax.plot(all_ras, all_decs, 'k--', alpha=0.5)

    arc_ras, arc_decs, _ = calc_orbit(arc_eps, sma, ecc, inc, aop, pan, tau, plx, mtot,␣
↪tau_ref_epoch=0)
    ax.plot(arc_ras, arc_decs, 'b-', linewidth=2)
    ax.plot(arc_ras[-1], arc_decs[-1], 'bo', markersize=7)

    ax.set_title("i = {0:d} deg".format(int(np.degrees(inc))))
    ax.set_aspect("equal")
    ax.set_xlim([1.1, -1.1])
    ax.set_ylim([-1.1, 1.1])
    ax.set_xlabel("RA")
    ax.set_ylabel("Dec")

fig.tight_layout()
```



What do the values of Ω mean?

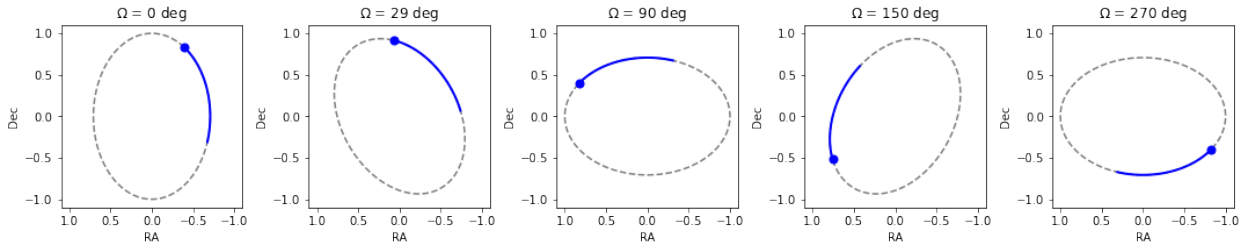
Ω or PAN defines the rotation of the orbit in the plane of the sky. Increasing PAN will rotate your orbit counterclockwise in the sky. Here are some examples:

[6]:
```
sma = 1
ecc, aop, plx, mtot = 0, 0, 1, 1
inc = np.pi/4 # 45 degrees
tau = 0.3
```

(continues on next page)

(continued from previous page)

```
pans = np.radians([0, 30, 90, 150, 270])

all_eps = np.linspace(0, 365.25, 200)
arc_eps = np.linspace(0, 75, 75)

fig = plt.figure(figsize=(15,4))
for i, pan in enumerate(pans):
    ax = fig.add_subplot(1, 5, i+1)

    all_ras, all_decs, _ = calc_orbit(all_eps, sma, ecc, inc, aop, pan, tau, plx, mtot,␣
↪tau_ref_epoch=0)
    ax.plot(all_ras, all_decs, 'k--', alpha=0.5)

    arc_ras, arc_decs, _ = calc_orbit(arc_eps, sma, ecc, inc, aop, pan, tau, plx, mtot,␣
↪tau_ref_epoch=0)
    ax.plot(arc_ras, arc_decs, 'b-', linewidth=2)
    ax.plot(arc_ras[-1], arc_decs[-1], 'bo', markersize=7)

    ax.set_title(r"$\Omega$ = {0:d} deg".format(int(np.degrees(pan))))
    ax.set_aspect("equal")
    ax.set_xlim([1.1, -1.1])
    ax.set_ylim([-1.1, 1.1])
    ax.set_xlabel("RA")
    ax.set_ylabel("Dec")

fig.tight_layout()
```



2.4 Contributing to the Code

`orbitize` is under active development, and we've still got a lot to do! To get involved, check out our contributor guidelines, look over our issues list, and/or reach out to Sarah. We'd love to have you on our team!

Members of our team have collectively drafted this community agreement stating both our values and ground rules. In joining our team, we ask that you read and (optionally) suggest changes to this document.

2.5 Detailed API Documentation

2.5.1 Basis

class orbitize.basis.**Basis**(*stellar_or_system_mass*, *mass_err*, *plx*, *plx_err*, *num_secondary_bodies*, *fit_secondary_mass*, *angle_upperlim=6.283185307179586*, *hipparcos_IAD=None*, *rv=False*, *rv_instruments=None*)

Abstract base class for different basis sets. All new basis objects should inherit from this class. This class is meant to control how priors are assigned to various basis sets and how conversions are made from the basis sets to the standard keplarian set.

Author: Tirth, 2021

set_default_mass_priors(*priors_arr*, *labels_arr*)

Adds the necessary priors for the stellar and/or companion masses.

Parameters

- **priors_arr** (*list of orbitize.priors.Prior objects*) – holds the prior objects for each parameter to be fitted (updated here)
- **labels_arr** (*list of strings*) – holds the name of all the parameters to be fitted (updated here)

set_hip_iad_priors(*priors_arr*, *labels_arr*)

Adds the necessary priors relevant to the hipparcos data to 'priors_arr' and updates 'labels_arr' with the priors' corresponding labels.

Parameters

- **priors_arr** (*list of orbitize.priors.Prior objects*) – holds the prior objects for each parameter to be fitted (updated here)
- **labels_arr** (*list of strings*) – holds the name of all the parameters to be fitted (updated here)

set_rv_priors(*priors_arr*, *labels_arr*)

Adds the necessary priors if radial velocity data is supplied to 'priors_arr' and updates 'labels_arr' with the priors' corresponding labels. This function assumes that 'rv' data has been supplied and a secondary mass is being fitted for.

Parameters

- **priors_arr** (*list of orbitize.priors.Prior objects*) – holds the prior objects for each parameter to be fitted (updated here)
- **labels_arr** (*list of strings*) – holds the name of all the parameters to be fitted (updated here)

verify_params()

Displays warnings about the 'fit_secondary_mass' and 'rv' parameters for all basis sets. Can be overriden by any basis class depending on the necessary parameters that need to be checked.

class orbitize.basis.**ObsPriors**(*stellar_or_system_mass*, *mass_err*, *plx*, *plx_err*, *num_secondary_bodies*, *fit_secondary_mass*, *angle_upperlim=6.283185307179586*, *hipparcos_IAD=None*, *rv=False*, *rv_instruments=None*, *tau_ref_epoch=58849*)

Basis used in Kosmo O'Neil+ 2019, and implemented here for use with the orbitize.priors.ObsPriors prior, following that paper. The basis is the same as the orbitize.basis.Period basis, except tp (time of periastron passage) is used in place of tau.

Parameters

- **stellar_or_system_mass** (*float*) – mass of the primary star (if fitting for dynamical masses of both components) or total system mass (if fitting using relative astrometry only) [M_sol]
- **mass_err** (*float*) – uncertainty on 'stellar_or_system_mass', in M_sol
- **plx** (*float*) – mean parallax of the system, in mas
- **plx_err** (*float*) – uncertainty on 'plx', in mas
- **num_secondary_bodies** (*int*) – number of secondary bodies in the system, should be at least 1
- **fit_secondary_mass** (*bool*) – if True, include the dynamical mass of orbitting body as fitted parameter, if False, 'stellar_or_system_mass' is taken to be total mass
- **angle_upperlim** (*float*) – either pi or 2pi, to restrict the prior range for 'pan' parameter (default: 2pi)
- **tau_ref_epoch** (*float*) – reference epoch for defining tau. Default 58849, the same as it is elsewhere in the code.

Limiations:

This basis cannot be used with RVs or absolute astrometry. It assumes inputs are vanilla Ra/Dec for relative astrometry.

`construct_priors()`

Generates the parameter label array and initializes the corresponding priors for each parameter to be sampled. For this basis, the parameters common to each companion are: per, ecc, inc, aop, pan, Tp. Parallax and mass priors both assumed to be fixed, and are added at the end.

Returns

list: list of strings (labels) that indicate the names of each parameter to sample

list: list of orbitize.priors.Prior objects that indicate the prior distribution of each label

Return type

tuple

`to_obspriors_basis(`*param_arr*, *after_date*`)`

Convert parameter array from Standard basis to ObsPriors basis. This function is used primarily for testing purposes.

Parameters

- **param_arr** (*np.array of float*) – RxM array of fitting parameters in the Standard basis, where R is the number of parameters being fit, and M is the number of orbits. If M=1 (for MCMC), this can be a 1D array.
- **after_date** (*float or np.array*) – tp will be the first periastron after this date. If None, use ref_epoch.

Returns

modifies 'param_arr' to contain ObsPriors elements.
Shape of 'param_arr' remains the same.

Return type
np.array of float

`to_standard_basis`(*param_arr*)

Convert parameter array from ObsPriors basis to Standard basis.

Parameters
param_arr (*np.array of float*) – RxM array of fitting parameters in the ObsPriors basis, where R is the number of parameters being fit, and M is the number of orbits. If M=1 (for MCMC), this can be a 1D array.

Returns

modifies 'param_arr' to contain Standard basis elements.
Shape of 'param_arr' remains the same.

Return type
np.array of float

`class orbitize.basis.Period`(*stellar_or_system_mass*, *mass_err*, *plx*, *plx_err*, *num_secondary_bodies*, *fit_secondary_mass*, *angle_upperlim=6.283185307179586*, *hipparcos_IAD=None*, *rv=False*, *rv_instruments=None*)

Modification of the standard basis, swapping our sma for period: (per, ecc, inc, aop, pan, tau).

Parameters

- **stellar_or_system_mass** (*float*) – mass of the primary star (if fitting for dynamical masses of both components) or total system mass (if fitting using relative astrometry only) [M_sol]
- **mass_err** (*float*) – uncertainty on 'stellar_or_system_mass', in M_sol
- **plx** (*float*) – mean parallax of the system, in mas
- **plx_err** (*float*) – uncertainty on 'plx', in mas
- **num_secondary_bodies** (*int*) – number of secondary bodies in the system, should be at least 1
- **fit_secondary_mass** (*bool*) – if True, include the dynamical mass of orbitting body as fitted parameter, if False, 'stellar_or_system_mass' is taken to be total mass
- **angle_upperlim** (*float*) – either pi or 2pi, to restrict the prior range for 'pan' parameter (default: 2pi)
- **hipparcos_IAD** (*orbitize.HipparcosLogProb object*) – if not 'None', then add relevant priors to this data (default: None)
- **rv** (*bool*) – if True, then there is radial velocity data and assign radial velocity priors, if False, then there is no radial velocity data and radial velocity priors are not assigned (default: False)
- **rv_instruments** (*np.array*) – array of unique rv instruments from the originally supplied data (default: None)

`construct_priors`()

Generates the parameter label array and initializes the corresponding priors for each parameter that's to be sampled. For the standard basis, the parameters common to each companion are: per, ecc, inc, aop, pan, tau. Parallax, hipparcos (optional), rv (optional), and mass priors are added at the end.

Returns

list: list of strings (labels) that indicate the names of each parameter to sample

list: list of orbitize.priors.Prior objects that indicate the prior distribution of each label

Return type
tuple

`to_period_basis`(*param_arr*)

Convert parameter array from standard basis to period basis by swapping out the semi-major axis parameter to period for each companion. This function is used primarily for testing purposes.

Parameters
param_arr (*np.array of float*) – RxM array of fitting parameters in the standard basis, where R is the number of parameters being fit, and M is the number of orbits. If M=1 (for MCMC), this can be a 1D array.

Returns

modifies 'param_arr' to contain the period for each companion
in each orbit rather than semi-major axis. Shape of 'param_arr' remains the same.

Return type
np.array of float

`to_standard_basis`(*param_arr*)

Convert parameter array from period basis to standard basis by swapping out the period parameter to semi-major axis for each companion.

Parameters
param_arr (*np.array of float*) – RxM array of fitting parameters in the period basis, where R is the number of parameters being fit, and M is the number of orbits. If M=1 (for MCMC), this can be a 1D array.

Returns

modifies 'param_arr' to contain the semi-major axis for each companion
in each orbit rather than period. Shape of 'param_arr' remains the same.

Return type
np.array of float

`class` orbitize.basis.**SemiAmp**(*stellar_or_system_mass*, *mass_err*, *plx*, *plx_err*, *num_secondary_bodies*, *fit_secondary_mass*, *angle_upperlim=6.283185307179586*, *hipparcos_IAD=None*, *rv=False*, *rv_instruments=None*)

Modification of the standard basis, swapping our sma for period and additionally sampling in the stellar radial velocity semi-amplitude: (per, ecc, inc, aop, pan, tau, K).

Note: Ideally, 'fit_secondary_mass' is true and rv data is supplied.

Parameters

- **stellar_or_system_mass** (*float*) – mass of the primary star (if fitting for dynamical masses of both components) or total system mass (if fitting using relative astrometry only) [M_sol]
- **mass_err** (*float*) – uncertainty on 'stellar_or_system_mass', in M_sol
- **plx** (*float*) – mean parallax of the system, in mas

- **plx_err** (*float*) – uncertainty on 'plx', in mas
- **num_secondary_bodies** (*int*) – number of secondary bodies in the system, should be at least 1
- **fit_secondary_mass** (*bool*) – if True, include the dynamical mass of orbitting body as fitted parameter, if False, 'stellar_or_system_mass' is taken to be total mass
- **angle_upperlim** (*float*) – either pi or 2pi, to restrict the prior range for 'pan' parameter (default: 2*pi)
- **hipparcos_IAD** (*orbitize.HipparcosLogProb object*) – if not 'None', then add relevant priors to this data (default: None)
- **rv** (*bool*) – if True, then there is radial velocity data and assign radial velocity priors, if False, then there is no radial velocity data and radial velocity priors are not assigned (default: False)
- **rv_instruments** (*np.array*) – array of unique rv instruments from the originally supplied data (default: None)

compute_companion_mass(*period*, *ecc*, *inc*, *semi_amp*, *m0*)

Computes a single companion's mass given period, eccentricity, inclination, stellar rv semi-amplitude, and stellar mass. Uses scipy.fsolve to compute the masses numerically.

Parameters

- **period** (*np.array of float*) – the period values for each orbit for a single companion (can be float)
- **ecc** (*np.array of float*) – the eccentricity values for each orbit for a single companion (can be float)
- **inc** (*np.array of float*) – the inclination values for each orbit for a single companion (can be float)
- **semi_amp** (*np.array of float*) – the stellar rv-semi amp values for each orbit (can be float)
- **m0** (*np.array of float*) – the stellar mass for each orbit (can be float)

Returns
the companion mass values for each orbit (can also just be a single float)

Return type
np.array of float

compute_companion_sma(*period*, *m0*, *m_n*)

Computes a single companion's semi-major axis using Kepler's Third Law for each orbit.

Parameters

- **period** (*np.array of float*) – the period values for each orbit for a single companion (can be float)
- **m0** (*np.array of float*) – the stellar mass for each orbit (can be float)
- **m_n** (*np.array of float*) – the companion mass for each orbit (can be float)

Returns
the semi-major axis values for each orbit

Return type
np.array of float

`construct_priors()`

Generates the parameter label array and initializes the corresponding priors for each parameter that's to be sampled. For the semi-amp basis, the parameters common to each companion are: per, ecc, inc, aop, pan, tau, K (stellar rv semi-amplitude). Parallax, hipparcos (optional), rv (optional), and mass priors are added at the end.

The mass parameter will always be m0.

Returns

list: list of strings (labels) that indicate the names of each parameter to sample

list: list of orbitize.priors.Prior objects that indicate the prior distribution of each label

Return type
tuple

`func(x, lhs, m0)`

Define function for scipy.fsolve to use when computing companion mass.

Parameters

- **x** (*float*) – the companion mass to be calculated (Msol)
- **lhs** (*float*) – the left hand side of the rv semi-amplitude equation (Msol^(1/3))
- **m0** (*float*) – the stellar mass (Msol)

Returns

the difference between the rhs and lhs of the rv semi-amplitude equation, 'x' is a
good companion mass when this difference is very close to zero

Return type
float

`to_semi_amp_basis(param_arr)`

Convert parameter array from standard basis to semi-amp basis by swapping out the semi-major axis parameter to period for each companion and computing the stellar rv semi-amplitudes for each companion.

Parameters
param_arr (*np.array of float*) – RxM array of fitting parameters in the period basis, where R is the number of parameters being fit, and M is the number of orbits. If M=1 (for MCMC), this can be a 1D array.

Returns

modifies 'param_arr' to contain the semi-major axis for each companion
in each orbit rather than period, appends stellar rv semi-amplitude parameters, and removes companion masses

Return type
np.array of float

`to_standard_basis(param_arr)`

Convert parameter array from semi-amp basis to standard basis by swapping out the period parameter to semi-major axis for each companion and computing the masses of each companion.

Parameters
param_arr (*np.array of float*) – RxM array of fitting parameters in the period basis, where R is the number of parameters being fit, and M is the number of orbits. If M=1 (for MCMC), this can be a 1D array.

Returns

modifies 'param_arr' to contain the semi-major axis for each companion
in each orbit rather than period, removes stellar rv semi-amplitude parameters for each companion, and appends the companion masses to 'param_arr'

Return type
np.array of float

`verify_params()`

Additionally warns that this basis will sample stellar mass rather than sample mass regardless of whether 'fit_secondary_mass' is True or not.

`class orbitize.basis.Standard(`*stellar_or_system_mass*, *mass_err*, *plx*, *plx_err*, *num_secondary_bodies*, *fit_secondary_mass*, *angle_upperlim=6.283185307179586*, *hipparcos_IAD=None*, *rv=False*, *rv_instruments=None*`)`

Standard basis set based upon the 6 standard Keplarian elements: (sma, ecc, inc, aop, pan, tau).

Parameters

- **stellar_or_system_mass** (*float*) – mass of the primary star (if fitting for dynamical masses of both components) or total system mass (if fitting using relative astrometry only) [M_sol]
- **mass_err** (*float*) – uncertainty on 'stellar_or_system_mass', in M_sol
- **plx** (*float*) – mean parallax of the system, in mas
- **plx_err** (*float*) – uncertainty on 'plx', in mas
- **num_secondary_bodies** (*int*) – number of secondary bodies in the system, should be at least 1
- **fit_secondary_mass** (*bool*) – if True, include the dynamical mass of orbitting body as fitted parameter, if False, 'stellar_or_system_mass' is taken to be total mass
- **angle_upperlim** (*float*) – either pi or 2pi, to restrict the prior range for 'pan' parameter (default: 2pi)
- **hipparcos_IAD** (*orbitize.HipparcosLogProb object*) – if not 'None', then add relevant priors to this data (default: None)
- **rv** (*bool*) – if True, then there is radial velocity data and assign radial velocity priors, if False, then there is no radial velocity data and radial velocity priors are not assigned (default: False)
- **rv_instruments** (*np.array*) – array of unique rv instruments from the originally supplied data (default: None)

`construct_priors()`

Generates the parameter label array and initializes the corresponding priors for each parameter that's to be sampled. For the standard basis, the parameters common to each companion are: sma, ecc, inc, aop, pan, tau. Parallax, hipparcos (optional), rv (optional), and mass priors are added at the end.

Returns

list: list of strings (labels) that indicate the names of each parameter to sample

list: list of orbitize.priors.Prior objects that indicate the prior distribution of each label

Return type
tuple

`to_standard_basis`(*param_arr*)

For standard basis, no conversion needs to be made.

Parameters
param_arr (*np.array of float*) – RxM array of fitting parameters in the standard basis, where R is the number of parameters being fit, and M is the number of orbits. If M=1 (for MCMC), this can be a 1d array.

Returns
param_arr without any modification

Return type
np.array of float

class orbitize.basis.**XYZ**(*stellar_or_system_mass*, *mass_err*, *plx*, *plx_err*, *num_secondary_bodies*, *fit_secondary_mass*, *data_table*, *best_epoch_idx*, *epochs*, *angle_upperlim=6.283185307179586*, *hipparcos_IAD=None*, *rv=False*, *rv_instruments=None*)

Defines an orbit using the companion's position and velocity components in XYZ space (x, y, z, xdot, ydot, zdot). The conversion algorithms used for this basis are defined in the following paper: http://www.dept.aoe.vt.edu/~lutze/AOE4134/9OrbitInSpace.pdf

Note: Does not have support with sep,pa data yet.

Note: Does not work for all multi-body data.

Parameters

- **stellar_or_system_mass** (*float*) – mass of the primary star (if fitting for dynamical masses of both components) or total system mass (if fitting using relative astrometry only) [M_sol]
- **mass_err** (*float*) – uncertainty on 'stellar_or_system_mass', in M_sol
- **plx** (*float*) – mean parallax of the system, in mas
- **plx_err** (*float*) – uncertainty on 'plx', in mas
- **num_secondary_bodies** (*int*) – number of secondary bodies in the system, should be at least 1
- **fit_secondary_mass** (*bool*) – if True, include the dynamical mass of orbitting body as fitted parameter, if False, 'stellar_or_system_mass' is taken to be total mass
- **input_table** (*astropy.table.Table*) – output from 'orbitize.read_input.read_file()'
- **best_epoch_idx** (*list*) – indices of the epochs corresponding to the smallest uncertainties
- **epochs** (*list*) – all of the astrometric epochs from 'input_table'
- **angle_upperlim** (*float*) – either pi or 2pi, to restrict the prior range for 'pan' parameter (default: 2*pi)
- **hipparcos_IAD** (*orbitize.HipparcosLogProb object*) – if not 'None', then add relevant priors to this data (default: None)
- **rv** (*bool*) – if True, then there is radial velocity data and assign radial velocity priors, if False, then there is no radial velocity data and radial velocity priors are not assigned (default: False)

- **rv_instruments** (*np.array*) – array of unique rv instruments from the originally supplied data (default: None)

Author: Rodrigo

construct_priors()

Generates the parameter label array and initializes the corresponding priors for each parameter that's to be sampled. For the xyz basis, the parameters common to each companion are: x, y, z, xdot, ydot, zdot. Parallax, hipparcos (optional), rv (optional), and mass priors are added at the end.

The xyz basis describes the position and velocity vectors with reference to the local coordinate system (the origin of the system is star).

Returns

list: list of strings (labels) that indicate the names of each parameter to sample

list: list of orbitize.priors.Prior objects that indicate the prior distribution of each label

Return type
tuple

standard_to_xyz(*epoch*, *elems*, *tau_ref_epoch=58849*, *tolerance=1e-09*, *max_iter=100*)

Converts array of orbital elements from the regular base of Keplerian orbits to positions and velocities in xyz. Uses code from orbitize.kepler

Parameters

- **epoch** (*float*) – Date in MJD of observation to calculate time of periastron passage (tau).
- **elems** (*np.array of floats*) – Orbital elements (sma, ecc, inc, aop, pan, tau, plx, mtot). If more than 1 set of parameters is passed, the first dimension must be the number of the orbital elements, and the second the number of orbital parameter sets.

Returns

Orbital elements in xyz (x-coordinate [au], y-coordinate [au], z-coordinate [au], velocity in x [km/s], velocity in y [km/s], velocity in z [km/s], parallax [mas], total mass of the two-body orbit

(M_* + M_planet) [Solar masses])

Return type
np.array

to_standard_basis(*param_arr*)

Makes a call to 'xyz_to_standard' to convert each companion's xyz parameters to the standard parameters an returns the updated array for conversion.

Parameters
param_arr (*np.array of float*) – RxM array of fitting parameters in the period basis, where R is the number of parameters being fit, and M is the number of orbits. If M=1 (for MCMC), this can be a 1D array.

Returns
Orbital elements in the standard basis for all companions.

Return type
np.array

to_xyz_basis(*param_arr*)

Makes a call to 'standard_to_xyz' to convert each companion's standard keplerian parameters to the xyz parameters an returns the updated array for conversion.

Parameters
param_arr (*np.array of float*) – RxM array of fitting parameters in the period basis, where R is the number of parameters being fit, and M is the number of orbits. If M=1 (for MCMC), this can be a 1D array.

Returns
Orbital elements in the xyz for all companions.

Return type
np.array

verify_params()

For now, additionally throws exceptions if data is supplied in sep/pa or if the best epoch for each body is one of the last two (which would inevtably mess up how the priors are imposed).

xyz_to_standard(*epoch*, *elems*, *tau_ref_epoch=58849*)

Converts array of orbital elements in terms of position and velocity in xyz to the standard basis.

Parameters

- **epoch** (*float*) – Date in MJD of observation to calculate time of periastron passage (tau).
- **elems** (*np.array of floats*) – Orbital elements in xyz basis (x-coordinate [au], y-coordinate [au], z-coordinate [au], velocity in x [km/s], velocity in y [km/s], velocity in z [km/s], parallax [mas], total mass of the two-body orbit (M_* + M_planet) [Solar masses]). If more than 1 set of parameters is passed, the first dimension must be the number of orbital parameter sets, and the second the orbital elements.

Returns

Orbital elements in the standard basis
(sma, ecc, inc, aop, pan, tau, plx, mtot)

Return type
np.array

orbitize.basis.**switch_tau_epoch**(*old_tau*, *old_epoch*, *new_epoch*, *period*)

Convert tau to another tau that uses a different referench epoch

Parameters

- **old_tau** (*float or np.array*) – old tau to convert
- **old_epoch** (*float or np.array*) – old reference epoch (days, typically MJD)
- **new_epoch** (*float or np.array*) – new reference epoch (days, same system as old_epoch)
- **period** (*float or np.array*) – orbital period (years)

Returns
new taus

Return type
float or np.array

orbitize.basis.**tau_to_manom**(*date*, *sma*, *mtot*, *tau*, *tau_ref_epoch*)

Gets the mean anomlay. Wrapper for kepler.tau_to_manom, kept here for backwards compatibility.

Parameters

- **date** (*float or np.array*) – MJD
- **sma** (*float*) – semi major axis (AU)
- **mtot** (*float*) – total mass (M_sun)
- **tau** (*float*) – epoch of periastron, in units of the orbital period
- **tau_ref_epoch** (*float*) – reference epoch for tau

Returns
mean anomaly on that date [0, 2pi)

Return type
float or np.array

orbitize.basis.**tau_to_tp**(*tau*, *ref_epoch*, *period*, *after_date=None*)

Convert tau (epoch of periastron in fractional orbital period after ref epoch) to t_p (date in days, usually MJD, but works with whatever system ref_epoch is given in)

Parameters

- **tau** (*float or np.array*) – value of tau to convert
- **ref_epoch** (*float or np.array*) – date (in days, typically MJD) that tau is defined relative to
- **period** (*float or np.array*) – period (in years) that tau is noralized with
- **after_date** (*float or np.array*) – tp will be the first periastron after this date. If None, use ref_epoch.

Returns
corresponding t_p of the taus

Return type
float or np.array

orbitize.basis.**tp_to_tau**(*tp*, *ref_epoch*, *period*)

Convert t_p to tau

Parameters

- **tp** (*float or np.array*) – value to t_p to convert (days, typically MJD)
- **ref_epoch** (*float or np.array*) – reference epoch (in days) that tau is defined from. Same system as tp (e.g., MJD)
- **period** (*float or np.array*) – period (in years) that tau is defined by

Returns
corresponding taus

Return type
float or np.array

2.5.2 Driver

class orbitize.driver.**Driver**(*input_data*, *sampler_str*, *num_secondary_bodies*, *stellar_or_system_mass*, *plx*, *mass_err=0*, *plx_err=0*, *lnlike='chi2_lnlike'*, *chi2_type='standard'*, *system_kwargs=None*, *mcmc_kwargs=None*)

Runs through `orbitize` methods in a standardized way.

Parameters

- **input_data** – Either a relative path to data file or astropy.table.Table object in the orbitize format. See `orbitize.read_input`
- **sampler_str** (*str*) – algorithm to use for orbit computation. "MCMC" for Markov Chain Monte Carlo, "OFTI" for Orbits for the Impatient
- **num_secondary_bodies** (*int*) – number of secondary bodies in the system. Should be at least 1.
- **stellar_or_system_mass** (*float*) – mass of the primary star (if fitting for dynamical masses of both components) or total system mass (if fitting using relative astrometry only) [M_sol]
- **plx** (*float*) – mean parallax of the system [mas]
- **mass_err** (*float, optional*) – uncertainty on `stellar_or_system_mass` [M_sol]
- **plx_err** (*float, optional*) – uncertainty on `plx` [mas]
- **lnlike** (*str, optional*) – name of function in `orbitize.lnlike` that will be used to compute likelihood. (default="chi2_lnlike")
- **chi2_type** (*str, optional*) – either "standard", or "log"
- **system_kwargs** (*dict, optional*) – `restrict_angle_ranges`, `tau_ref_epoch`, `fit_secondary_mass`, `hipparcos_IAD`, `gaia`, `use_rebound`, `fitting_basis` for `orbitize.system.System`.
- **mcmc_kwargs** (*dict, optional*) – `num_temps`, `num_walkers`, and `num_threads` kwargs for `orbitize.sampler.MCMC`

Written: Sarah Blunt, 2018

2.5.3 Gaia API Module

class orbitize.gaia.**GaiaLogProb**(*gaia_num*, *hiplogprob*, *dr='dr2'*, *query=True*, *gaia_data=None*)

Class to compute the log probability of an orbit with respect to a single astrometric position point from Gaia. Uses astroquery to look up Gaia astrometric data, and computes log-likelihood. To be used in conjunction with orbitize.hipparcos.HipLogProb; see documentation for that object for more detail.

Follows Nielsen+ 2020 (studying the orbit of beta Pic b). Note that this class currently only fits for the position of the star in the Gaia epoch, not the star's proper motion.

Note: In orbitize, it is possible to perform a fit to just the Hipparcos IAD, but not to just the Gaia astrometric data.

Parameters

- **gaia_num** (*int*) – the Gaia source ID of the object you're fitting. Note that the dr2 and edr3 source IDs are not necessarily the same.
- **hiplogprob** (*orbitize.hipparcos.HipLogProb*) – object containing all info relevant to Hipparcos IAD fitting
- **dr** (*str*) – either 'dr2' or 'edr3'
- **query** (*bool*) – if True, queries the Gaia database for astrometry of the target (requires an internet connection). If False, uses user-input astrometric values (runs without internet).
- **gaia_data** (*dict*) – see *query* keyword above. If *query* set to False, then user must supply a dictionary of Gaia astometry in the following form:

  gaia_data = {
  'ra': 139.4 # RA in degrees 'dec': 139.4 # Dec in degrees 'ra_error': 0.004 # RA error in mas 'dec_error': 0.004 # Dec error in mas

  }

Written: Sarah Blunt, 2021

compute_lnlike(*raoff_model*, *deoff_model*, *samples*, *param_idx*)

Computes the log likelihood of an orbit model with respect to a single Gaia astrometric point. This is added to the likelihoods calculated with respect to other data types in `sampler._logl()`.

Parameters

- **raoff_model** (*np.array of float*) – 2xM primary RA offsets from the barycenter incurred from orbital motion of companions (i.e. not from parallactic motion), where M is the number of orbits being tested, and raoff_model[0,:] are position predictions at the Hipparcos epoch, and raoff_model[1,:] are position predictions at the Gaia epoch
- **deoff_model** (*np.array of float*) – 2xM primary decl offsets from the barycenter incurred from orbital motion of companions (i.e. not from parallactic motion), where M is the number of orbits being tested, and deoff_model[0,:] are position predictions at the Hipparcos epoch, and deoff_model[1,:] are position predictions at the Gaia epoch
- **samples** (*np.array of float*) – R-dimensional array of fitting parameters, where R is the number of parameters being fit. Must be in the same order documented in `System`.
- **param_idx** – a dictionary matching fitting parameter labels to their indices in an array of fitting parameters (generally set to System.basis.param_idx).

Returns

array of length M, where M is the number of input
orbits, representing the log likelihood of each orbit with respect to the Gaia position measurement.

Return type
np.array of float

class orbitize.gaia.**HGCALogProb**(*hip_id*, *hiplogprob*, *gost_filepath*, *hgca_filepath=None*)

Class to compute the log probability of an orbit with respect to the proper motion anomalies derived jointly from Gaia and Hipparcos using the HGCA catalogs produced by Brandt (2018, 2021). With this class, both Gaia and Hipparcos data will be considered. Do not need to pass in the Hipparcos class into System!

Required auxiliary data:

- HGCA of acceleration (either DR2 or EDR3)
- Hipparcos IAD file (see orbitize.hipparcos for more info)

- Gaia Observation Forecast Tool (GOST) CSV output (https://gaia.esac.esa.int/gost/).

For GOST, after entering the target name and resolving its coordinates, use 2014-07-26T00:00:00 as the start date. For the end date, use 2016-05-23T00:00:00 for DR2 and 2017-05-28T00:00:00 for EDR3.

Parameters

- **hip_id** (*int*) – the Hipparcos source ID of the object you're fitting.
- **hiplogprob** (*orbitize.hipparcos.HipLogProb*) – object containing all info relevant to Hipparcos IAD fitting
- **gost_filepath** (*str*) – path to CSV file outputted by GOST
- **hgca_filepath** (*str*) – path to HGCA catalog FITS file. If None, will download and store in orbitize.DATADIR

Written: Jason Wang, 2022

compute_lnlike(*raoff_model*, *deoff_model*, *samples*, *param_idx*)

Computes the log likelihood of an orbit model with respect to a single Gaia astrometric point. This is added to the likelihoods calculated with respect to other data types in `sampler._logl()`.

Parameters

- **raoff_model** (*np.array of float*) – NxM primary RA offsets from the barycenter incurred from orbital motion of companions (i.e. not from parallactic motion), where N is the number of epochs of combined from Hipparcos and Gaia and M is the number of orbits being tested, and raoff_model[0,:] are position predictions at the Hipparcos epoch, and raoff_model[1,:] are position predictions at the Gaia epoch
- **deoff_model** (*np.array of float*) – NxM primary decl offsets from the barycenter incurred from orbital motion of companions (i.e. not from parallactic motion), where N is the number of epochs of combined from Hipparcos and Gaia and M is the number of orbits being tested, and deoff_model[0,:] are position predictions at the Hipparcos epoch, and deoff_model[1,:] are position predictions at the Gaia epoch
- **samples** (*np.array of float*) – R-dimensional array of fitting parameters, where R is the number of parameters being fit. Must be in the same order documented in `System`.
- **param_idx** – a dictionary matching fitting parameter labels to their indices in an array of fitting parameters (generally set to System.basis.param_idx).

Returns

array of length M, where M is the number of input
orbits, representing the log likelihood of each orbit with respect to the Gaia position measurement.

Return type
np.array of float

2.5.4 Hipparcos API Module

class orbitize.hipparcos.**HipparcosLogProb**(*path_to_iad_file*, *hip_num*, *num_secondary_bodies*, *alphadec0_epoch=1991.25*, *renormalize_errors=False*)

Class to compute the log probability of an orbit with respect to the Hipparcos Intermediate Astrometric Data (IAD). If using a DVD file, queries Vizier for all metadata relevant to the IAD, and reads in the IAD themselves from a specified location. Follows Nielsen+ 2020 (studying the orbit of beta Pic b).

Fitting the Hipparcos IAD requires fitting for the following five parameters. They are added to the vector of fitting parameters in system.py, but are described here for completeness. See Nielsen+ 2020 for more detail.

- **alpha0: RA offset from the reported Hipparcos position at a particular** epoch (usually 1991.25) [mas]
- **delta0: Dec offset from the reported Hipparcos position at a particular** epoch (usually 1991.25) [mas]
- pm_ra: RA proper motion [mas/yr]
- pm_dec: Dec proper motion [mas/yr]
- plx: parallax [mas]

Note: In orbitize, it is possible to perform a fit to just the Hipparcos IAD, but not to just the Gaia astrometric data.

Parameters

- **path_to_iad_file** (*str*) – location of IAD file to be used in your fit. See the Hipparcos tutorial for a walkthrough of how to download these files.
- **hip_num** (*str*) – Hipparcos ID of star. Available on Simbad. Should have zeros in the prefix if number is <100,000. (i.e. 27321 should be passed in as '027321').
- **num_secondary_bodies** (*int*) – number of companions in the system
- **alphadec0_epoch** (*float*) – epoch (in decimal year) that the fitting parameters alpha0 and delta0 are defined relative to (see above).
- **renormalize_errors** (*bool*) – if True, normalize the scan errors to get chisq_red = 1, following Nielsen+ 2020 (eq 10). In general, this should be False, but it's helpful for testing. Check out *orbitize.hipparcos.nielsen_iad_refitting_test()* for an example using this renormalization.

Written: Sarah Blunt & Rob de Rosa, 2021

compute_lnlike(*raoff_model*, *deoff_model*, *samples*, *param_idx*)

Computes the log likelihood of an orbit model with respect to the Hipparcos IAD. This is added to the likelihoods calculated with respect to other data types in `sampler._logl()`.

Parameters

- **raoff_model** (*np.array of float*) – M-dimensional array of primary RA offsets from the barycenter incurred from orbital motion of companions (i.e. not from parallactic motion), where M is the number of epochs of IAD scan data.
- **deoff_model** (*np.array of float*) – M-dimensional array of primary RA offsets from the barycenter incurred from orbital motion of companions (i.e. not from parallactic motion), where M is the number of epochs of IAD scan data.

- **samples** (*np.array of float*) – R-dimensional array of fitting parameters, where R is the number of parameters being fit. Must be in the same order documented in `System`.
- **param_idx** – a dictionary matching fitting parameter labels to their indices in an array of fitting parameters (generally set to System.basis.param_idx).

Returns

array of length M, where M is the number of input
orbits, representing the log likelihood of each orbit with respect to the Hipparcos IAD.

Return type
np.array of float

class orbitize.hipparcos.**PMPlx_Motion**(*epochs_mjd*, *alpha0*, *delta0*, *alphadec0_epoch=1991.25*)

Class to compute the predicted position at an array of epochs given a parallax and proper motion model (NO orbital motion is added in this class).

Parameters

- **times_mjd** (*np.array of float*) – times (in mjd) at which we have absolute astrometric measurements
- **alpha0** (*float*) – measured RA position (in degrees) of the object at alphadec0_epoch (see below).
- **delta0** (*float*) – measured Dec position (in degrees) of the object at alphadec0_epoch (see below).
- **alphadec0_epoch** (*float*) – a (fixed) reference time. For stars with Hipparcos data, this should generally be 1991.25, but you can define it however you want. Absolute astrometric data (passed in via an orbitize! data table) should be defined as offsets from the reported position of the object at this epoch (with propagated uncertainties). For example, if you have two absolute astrometric measurements taken with GRAVITY, as well as a Hipparcos-derived position (at epoch 1991.25), alphadec0_epoch should be 1991.25, and you should pass in absolute astrometry in terms of mas *offset* from the Hipparcos catalog position, with propagated errors of your measurement and the Hipparcos measurement.

compute_astrometric_model(*samples*, *param_idx*, *epochs=None*)

Compute the astrometric prediction at self.epochs_mjd from parallax and proper motion alone, given an array of model parameters (no orbital motion is added here).

Parameters

- **samples** (*np.array of float*) – Length R array of fitting parameters, where R is the number of parameters being fit. Must be in the same order documented in `System`.
- **param_idx** – a dictionary matching fitting parameter labels to their indices in an array of fitting parameters (generally set to System.basis.param_idx).
- **epochs** – if None, use self.epochs for astrometric predictions. Otherwise, use this array passed in [in decimalyear].

Returns

- **float: predicted RA*cos(delta0) position offsets from the measured**
  position at alphadec0_epoch, calculated for each input epoch [mas]
- **float: predicted Dec position offsets from the measured position**
  at alphadec0_epoch, calculated for each input epoch [mas]

Return type
tuple of

orbitize.hipparcos.**nielsen_iad_refitting_test**(*iad_file*, *hip_num='027321'*, *saveplot='bPic_IADrefit.png'*, *burn_steps=100*, *mcmc_steps=5000*)

Reproduce the IAD refitting test from Nielsen+ 2020 (end of Section 3.1). The default MCMC parameters are what you'd want to run before using the IAD for a new system. This fit uses 100 walkers.

Parameters

- **iad_loc** (*str*) – path to the IAD file.
- **hip_num** (*str*) – Hipparcos ID of star. Available on Simbad. Should have zeros in the prefix if number is <100,000. (i.e. 27321 should be passed in as '027321').
- **saveplot** (*str*) – what to save the summary plot as. If None, don't make a plot
- **burn_steps** (*int*) – number of MCMC burn-in steps to run.
- **mcmc_steps** (*int*) – number of MCMC production steps to run.

Returns

numpy.array of float: n_steps x 5 array of posterior samples

orbitize.hipparcos.HipparcosLogProb: the object storing relevant metadata for the performed Hipparcos IAD fit

Return type
tuple

2.5.5 Kepler Solver

This module solves for the orbit of the planet given Keplerian parameters.

orbitize.kepler.**calc_orbit**(*epochs*, *sma*, *ecc*, *inc*, *aop*, *pan*, *tau*, *plx*, *mtot*, *mass_for_Kamp=None*, *tau_ref_epoch=58849*, *tolerance=1e-09*, *max_iter=100*, *use_c=True*, *use_gpu=False*)

Returns the separation and radial velocity of the body given array of orbital parameters (size n_orbs) at given epochs (array of size n_dates)

Based on orbit solvers from James Graham and Rob De Rosa. Adapted by Jason Wang and Henry Ngo.

Parameters

- **epochs** (*np.array*) – MJD times for which we want the positions of the planet
- **sma** (*np.array*) – semi-major axis of orbit [au]
- **ecc** (*np.array*) – eccentricity of the orbit [0,1]
- **inc** (*np.array*) – inclination [radians]
- **aop** (*np.array*) – argument of periastron [radians]
- **pan** (*np.array*) – longitude of the ascending node [radians]
- **tau** (*np.array*) – epoch of periastron passage in fraction of orbital period past MJD=0 [0,1]
- **plx** (*np.array*) – parallax [mas]
- **mtot** (*np.array*) – total mass of the two-body orbit (M_* + M_planet) [Solar masses]
- **mass_for_Kamp** (*np.array, optional*) – mass of the body that causes the RV signal. For example, if you want to return the stellar RV, this is the planet mass. If you want to return the

planetary RV, this is the stellar mass. [Solar masses]. For planet mass ~ 0, mass_for_Kamp ~ M_tot, and function returns planetary RV (default).

- **tau_ref_epoch** (*float, optional*) – reference date that tau is defined with respect to (i.e., tau=0)
- **tolerance** (*float, optional*) – absolute tolerance of iterative computation. Defaults to 1e-9.
- **max_iter** (*int, optional*) – maximum number of iterations before switching. Defaults to 100.
- **use_c** (*bool, optional*) – Use the C solver if configured. Defaults to True
- **use_gpu** (*bool, optional*) – Use the GPU solver if configured. Defaults to False

Returns

raoff (np.array): array-like (n_dates x n_orbs) of RA offsets between the bodies (origin is at the other body) [mas]

deoff (np.array): array-like (n_dates x n_orbs) of Dec offsets between the bodies [mas]

vz (np.array): array-like (n_dates x n_orbs) of radial velocity of one of the bodies (see *mass_for_Kamp* description) [km/s]

Return type
3-tuple

Written: Jason Wang, Henry Ngo, 2018

orbitize.kepler.**tau_to_manom**(*date*, *sma*, *mtot*, *tau*, *tau_ref_epoch*)

Gets the mean anomlay

Parameters

- **date** (*float or np.array*) – MJD
- **sma** (*float*) – semi major axis (AU)
- **mtot** (*float*) – total mass (M_sun)
- **tau** (*float*) – epoch of periastron, in units of the orbital period
- **tau_ref_epoch** (*float*) – reference epoch for tau

Returns
mean anomaly on that date [0, 2pi)

Return type
float or np.array

2.5.6 Log(Likelihood)

orbitize.lnlike.**chi2_lnlike**(*data*, *errors*, *corrs*, *model*, *jitter*, *seppa_indices*, *chi2_type='standard'*)

Compute Log of the chi2 Likelihood

Args:

data (np.array): Nobsx2 array of data, where data[:,0] = sep/RA/RV for every epoch, and data[:,1] = corresponding pa/DEC/np.nan.

errors (np.array): Nobsx2 array of errors for each data point. Same format as data.

corrs (np.array): Nobs array of Pearson correlation coeffs
between the two quantities. If there is none, can be None.

model (np.array): Nobsx2xM array of model predictions, where M is the number of orbits being compared against the data. If M is 1, `model` can be 2 dimensional. jitter (np.array): Nobsx2xM array of jitter values to add to errors.

Elements of array should be 0 for for all data other than stellar rvs.

seppa_indices (list): list of epoch numbers whose observations are
given in sep/PA. This list is located in System.seppa.

chi2_type (string): the format of chi2 to use is either 'standard' or 'log'

Returns:
np.array: Nobsx2xM array of chi-squared values.

Note: (1) **Example**: We have 8 epochs of data for a system. OFTI returns an array of 10,000 sets of orbital parameters. The `model` input for this function should be an array of dimension 8 x 2 x 10,000.

(2) Chi2_log: redefining chi-sqaured in log scale may give a more stable optimization. This works on separation and position angle data (seppa) not right ascension and declination (radec) data, but it is possible to convert between the two within Orbitize! using the function 'orbitize.system'radec2seppa' (see docuemntation). This implementation defines sep chi-squared in log scale, and defines pa chi-sq using complex phase representation. log sep chisq = (log sep - log sep_true)^2 / (sep_sigma / sep_true)^2 pa chisq = 2 * (1 - cos(pa-pa_true))/pa_sigma^2

---

i

orbitize.lnlike.**chi2_norm_term**(*errors*, *corrs*)

Return only the normalization term of the Gaussian likelihood:

$$-log(\sqrt{(2\pi * error^2)})$$

or

$$-0.5 * (log(det(C)) + N * log(2\pi))$$

Parameters

- **errors** (*np.array*) – Nobsx2 array of errors for each data point. Same format as `data`.
- **corrs** (*np.array*) – Nobs array of Pearson correlation coeffs between the two quantities. If there is none, can be None.

Returns
sum of the normalization terms

Return type
float

2.5.7 N-body Backend

orbitize.nbody.**calc_orbit**(*epochs*, *sma*, *ecc*, *inc*, *aop*, *pan*, *tau*, *plx*, *mtot*, *tau_ref_epoch=58849*, *m_pl=None*, *output_star=False*, *integrator='ias15'*)

Solves for position for a set of input orbital elements using rebound.

Parameters

- **epochs** (*np.array*) – MJD times for which we want the positions of the planet
- **sma** (*np.array*) – semi-major axis array of secondary bodies. For three planets, this should look like: np.array([sma1, sma2, sma3]) [au]
- **ecc** (*np.array*) – eccentricity of the orbits (same format as sma) [0,1]
- **inc** (*np.array*) – inclinations (same format as sma) [radians]
- **aop** (*np.array*) – arguments of periastron (same format as sma) [radians]
- **pan** (*np.array*) – longitudes of the ascending node (same format as sma) [radians]
- **tau** (*np.array*) – epochs of periastron passage in fraction of orbital period past MJD=0 (same format as sma) [0,1]
- **plx** (*float*) – parallax [mas]
- **mtot** (*float*) – total mass of the two-body orbit (M_* + M_planet) [Solar masses]
- **tau_ref_epoch** (*float, optional*) – reference date that tau is defined with respect to
- **m_pl** (*np.array, optional*) – masss of the planets (same format as sma) [solar masses]
- **output_star** (*bool*) – if True, also return the position of the star relative to the barycenter.
- **integrator** (*str*) – value to set for rebound.sim.integrator. Default "ias15"

Returns

raoff (np.array): array-like (n_dates x n_bodies x n_orbs) of RA offsets between the bodies (origin is at the other body) [mas]

deoff (np.array): array-like (n_dates x n_bodies x n_orbs) of Dec offsets between the bodies [mas]

vz (np.array): array-like (n_dates x n_bodies x n_orbs) of radial velocity of one of the bodies (see *mass_for_Kamp* description) [km/s]

Return type
3-tuple

2.5.8 Plotting Methods

orbitize.plot.**plot_corner**(*results*, *param_list=None*, ***corner_kwargs*)

Make a corner plot of posterior on orbit fit from any sampler

Parameters

- **param_list** (*list of strings*) – each entry is a name of a parameter to include. Valid strings:

```
sma1: semimajor axis
ecc1: eccentricity
inc1: inclination
aop1: argument of periastron
pan1: position angle of nodes
tau1: epoch of periastron passage, expressed as fraction of orbital␣
↪period
per1: period
K1: stellar radial velocity semi-amplitude
[repeat for 2, 3, 4, etc if multiple objects]
plx:  parallax
pm_ra: RA proper motion
pm_dec: Dec proper motion
alpha0: primary offset from reported Hipparcos RA @ alphadec0_epoch␣
↪(generally 1991.25)
delta0: primary offset from reported Hipparcos Dec @ alphadec0_epoch␣
↪(generally 1991.25)
gamma: rv offset
sigma: rv jitter
mi: mass of individual body i, for i = 0, 1, 2, ... (only if fit_
↪secondary_mass == True)
mtot: total mass (only if fit_secondary_mass == False)
```

- ****corner_kwargs** – any remaining keyword args are sent to `corner.corner`. See here. Note: default axis labels used unless overwritten by user input.

Returns
corner plot

Return type
`matplotlib.pyplot.Figure`

Note: **Example**: Use `param_list = ['sma1,ecc1,inc1,sma2,ecc2,inc2']` to only plot posteriors for semimajor axis, eccentricity and inclination of the first two companions

---

Written: Henry Ngo, 2018

`orbitize.plot.`**`plot_orbits`**(*results*, *object_to_plot=1*, *start_mjd=51544.0*, *num_orbits_to_plot=100*, *num_epochs_to_plot=100*, *square_plot=True*, *show_colorbar=True*, *cmap=<matplotlib.colors.LinearSegmentedColormap object>*, *sep_pa_color='lightgrey'*, *sep_pa_end_year=2025.0*, *cbar_param='Epoch [year]'*, *mod180=False*, *rv_time_series=False*, *plot_astrometry=True*, *plot_astrometry_insts=False*, *plot_errorbars=True*, *rv_time_series2=False*, *primary_instrument_name=None*, *fontsize=20*, *fig=None*)

Plots one orbital period for a select number of fitted orbits for a given object, with line segments colored according to time

Parameters

- **`object_to_plot`** (*int*) – which object to plot (default: 1)
- **`start_mjd`** (*float*) – MJD in which to start plotting orbits (default: 51544, the year 2000)
- **`num_orbits_to_plot`** (*int*) – number of orbits to plot (default: 100)
- **`num_epochs_to_plot`** (*int*) – number of points to plot per orbit (default: 100)

- **square_plot** (*Boolean*) – Aspect ratio is always equal, but if square_plot is True (default), then the axes will be square, otherwise, white space padding is used
- **show_colorbar** (*Boolean*) – Displays colorbar to the right of the plot [True]
- **cmap** (*matplotlib.cm.ColorMap*) – color map to use for making orbit tracks (default: modified Purples_r)
- **sep_pa_color** (*string*) – any valid matplotlib color string, used to set the color of the orbit tracks in the Sep/PA panels (default: 'lightgrey').
- **sep_pa_end_year** (*float*) – decimal year specifying when to stop plotting orbit tracks in the Sep/PA panels (default: 2025.0).
- **cbar_param** (*string*) – options are the following: 'Epoch [year]', 'sma1', 'ecc1', 'inc1', 'aop1', 'pan1', 'tau1', 'plx. Number can be switched out. Default is Epoch [year].
- **mod180** (*Bool*) – if True, PA will be plotted in range [180, 540]. Useful for plotting short arcs with PAs that cross 360 deg during observations (default: False)
- **rv_time_series** (*Boolean*) – if fitting for secondary mass using MCMC for rv fitting and want to display time series, set to True.
- **plot_astrometry** (*Boolean*) – set to True by default. Plots the astrometric data.
- **plot_astrometry_insts** (*Boolean*) – set to False by default. Plots the astrometric data by instruments.
- **fig** (*matplotlib.pyplot.Figure*) – optionally include a predefined Figure object to plot the orbit on. Most users will not need this keyword.

Returns
the orbit plot if input is valid, None otherwise

Return type
matplotlib.pyplot.Figure

(written): Henry Ngo, Sarah Blunt, 2018 Additions by Malena Rice, 2019 Additions by Dino Hsu, 2023

orbitize.plot.**plot_propermotion**(*results*, *system*, *object_to_plot=1*, *start_mjd=44239.0*, *periods_to_plot=1*, *end_year=2030.0*, *alpha=0.05*, *num_orbits_to_plot=100*, *num_epochs_to_plot=100*, *show_colorbar=True*, *cmap=<matplotlib.colors.LinearSegmentedColormap object>*, *cbar_param=None*, *tight_layout=False*)

Plots the proper motion of a host star as induced by a companion for one orbital period for a select number of fitted orbits for a given object, with line segments colored according to a given parameter (most informative is usually mass of companion)

Important Note: These plotted trajectories aren't what are fitting in the likelihood evaluation for the HGCA runs. The implementation forward models the Hip/Gaia measurements per epoch and infers the differential proper motions. This plot is given only for the purposes of an approximate visualization.

Parameters

- **system** (*object*) – orbitize.system object with a HGCALogProb passed to system.gaia
- **object_to_plot** (*int*) – which object to plot (default: 1)
- **start_mjd** (*float*) – MJD in which to start plotting orbits (default: 51544, the year 2000)
- **periods_to_plot** (*int*) – number of periods to plot (default: 1)
- **end_year** (*float*) – decimal year specifying when to stop plotting orbit tracks in the Sep/PA panels (default: 2025.0).

- **alpha** (*float*) – transparency of lines (default: 0.05)
- **num_orbits_to_plot** (*int*) – number of orbits to plot (default: 100)
- **num_epochs_to_plot** (*int*) – number of points to plot per orbit (default: 100)
- **show_colorbar** (*Boolean*) – Displays colorbar to the right of the plot [True]
- **cmap** (*matplotlib.cm.ColorMap*) – color map to use for making orbit tracks (default: modified Purples_r)
- **cbar_param** (*string*) – options are the following: 'sma1', 'ecc1', 'inc1', 'aop1', 'pan1', 'tau1', 'plx', 'm0', 'm1', etc. Number can be switched out. Default is None.
- **tight_layout** (*bool*) – apply plt.tight_layout function?
- **fig** (*matplotlib.pyplot.Figure*) – optionally include a predefined Figure object to plot the orbit on. Most users will not need this keyword.

Returns
the orbit plot if input is valid, None otherwise

Return type
matplotlib.pyplot.Figure

(written): William Balmer (2023), based on plot_orbits by Sarah Blunt and Henry Ngo

orbitize.plot.**plot_residuals**(*my_results*, *object_to_plot=1*, *start_mjd=51544*, *num_orbits_to_plot=100*, *num_epochs_to_plot=100*, *sep_pa_color='lightgrey'*, *sep_pa_end_year=2025.0*, *cbar_param='Epoch [year]'*, *mod180=False*)

Plots sep/PA residuals for a set of orbits

Parameters

- **my_results** (*orbitiez.results.Results*) – results to plot
- **object_to_plot** (*int*) – which object to plot (default: 1)
- **start_mjd** (*float*) – MJD in which to start plotting orbits (default: 51544, the year 2000)
- **num_orbits_to_plot** (*int*) – number of orbits to plot (default: 100)
- **num_epochs_to_plot** (*int*) – number of points to plot per orbit (default: 100)
- **sep_pa_color** (*string*) – any valid matplotlib color string, used to set the color of the orbit tracks in the Sep/PA panels (default: 'lightgrey').
- **sep_pa_end_year** (*float*) – decimal year specifying when to stop plotting orbit tracks in the Sep/PA panels (default: 2025.0).
- **cbar_param** (*string*) – options are the following: 'Epoch [year]', 'sma1', 'ecc1', 'inc1', 'aop1', 'pan1', 'tau1', 'plx. Number can be switched out. Default is Epoch [year].
- **mod180** (*Bool*) – if True, PA will be plotted in range [180, 540]. Useful for plotting short arcs with PAs that cross 360 deg during observations (default: False)

Returns
the residual plots

Return type
matplotlib.pyplot.Figure

2.5.9 Priors

class orbitize.priors.**GaussianPrior**(*mu*, *sigma*, *no_negatives=True*)

Gaussian prior.

$$log(p(x|\sigma,\mu)) \propto \frac{(x-\mu)}{\sigma}$$

Parameters

- **mu** (*float*) – mean of the distribution
- **sigma** (*float*) – standard deviation of the distribution
- **no_negatives** (*bool*) – if True, only positive values will be drawn from
- **prior** (*this*) –
- **0** (*and the probability of negative values will be*) –
- **(default** – True).

(written) Sarah Blunt, 2018

compute_lnprob(*element_array*)

Compute log(probability) of an array of numbers wrt a Gaussian distibution. Negative numbers return a probability of -inf.

Parameters
element_array (*float or np.array of float*) – array of numbers. We want the probability of drawing each of these from the appopriate Gaussian distribution

Returns
array of log(probability) values, corresponding to the probability of drawing each of the numbers in the input *element_array*.

Return type
numpy array of float

draw_samples(*num_samples*)

Draw positive samples from a Gaussian distribution. Negative samples will not be returned.

Parameters
num_samples (*float*) – the number of samples to generate

Returns
samples drawn from the appropriate Gaussian distribution. Array has length *num_samples*.

Return type
numpy array of float

transform_samples(*u*)

Transform uniform 1D samples, u, to samples drawn from a Gaussian distribution.

Parameters
u (*array of floats*) – list of samples with values 0 < u < 1.

Returns
1D u samples transformed to a Gaussian distribution.

Return type
numpy array of floats

class orbitize.priors.**KDEPrior**(*gaussian_kde*, *total_params*, *bounds=[]*, *log_scale_arr=[]*)

Gaussian kernel density estimation (KDE) prior. This class is a wrapper for scipy.stats.gaussian_kde.

Parameters

- **gaussian_kde** (*scipy.stats.gaussian_kde*) – scipy KDE object containing the KDE defined by the user
- **total_params** (*float*) – number of parameters in the KDE
- **bounds** (*array_like of bool, optional*) – bounds for the KDE out of which the prob returned is -Inf
- **bounds** – if True for a parameter the parameter is fit to the KDE in log-scale

Written: Jorge LLop-Sayson, Sarah Blunt (2021)

compute_lnprob(*element_array*)

Compute log(probability) of an array of numbers wrt a the defined KDE. Negative numbers return a probability of -inf.

Parameters

element_array (*float or np.array of float*) – array of numbers. We want the probability of drawing each of these from the KDE.

Returns

array of log(probability) values, corresponding to the probability of drawing each of the numbers in the input *element_array*.

Return type

numpy array of float

draw_samples(*num_samples*)

Draw positive samples from the KDE. Negative samples will not be returned.

Parameters

num_samples (*float*) – the number of samples to generate.

Returns

samples drawn from the KDE distribution. Array has length *num_samples*.

Return type

numpy array of float

increment_param_num()

Increment the index to evaluate the appropriate parameter.

class orbitize.priors.**LinearPrior**(*m*, *b*)

Draw samples from the probability distribution:

$$p(x) \propto mx + b$$

where m is negative, b is positive, and the range is [0,-b/m].

Parameters

- **m** (*float*) – slope of line. Must be negative.
- **b** (*float*) – y intercept of line. Must be positive.

draw_samples(*num_samples*)

Draw samples from a descending linear distribution.

Parameters
num_samples (*float*) – the number of samples to generate

Returns
samples ranging from [0, -b/m) as floats.

Return type
np.array

transform_samples(*u*)

Transform uniform 1D samples, u, to samples drawn from a Linear distribution.

Parameters
u (*array of floats*) – list of samples with values 0 < u < 1.

Returns
1D u samples transformed to a Linear distribution.

Return type
numpy array of floats

class orbitize.priors.**LogUniformPrior**(*minval*, *maxval*)

This is the probability distribution $p(x) \propto 1/x$

The __init__ method should take in a "min" and "max" value of the distribution, which correspond to the domain of the prior. (If this is not implemented, the prior has a singularity at 0 and infinite integrated probability).

Parameters

- **minval** (*float*) – the lower bound of this distribution
- **maxval** (*float*) – the upper bound of this distribution

compute_lnprob(*element_array*)

Compute the prior probability of each element given that its drawn from a Log-Uniofrm prior

Parameters
element_array (*float or np.array of float*) – array of paramters to compute the prior probability of

Returns
array of prior probabilities

Return type
np.array

draw_samples(*num_samples*)

Draw samples from this 1/x distribution.

Parameters
num_samples (*float*) – the number of samples to generate

Returns
samples ranging from [minval, maxval) as floats.

Return type
np.array

`transform_samples(u)`

Transform uniform 1D samples, u, to samples drawn from a Log Uniform distribution.

Parameters
u (*array of floats*) – list of samples with values 0 < u < 1.

Returns
1D u samples transformed to a Log Uniform distribution.

Return type
numpy array of floats

`class orbitize.priors.NearestNDInterpPrior(interp_fct, total_params)`

Nearest Neighbor interp. This class is a wrapper for scipy.interpolate.NearestNDInterpolator.

Parameters

- **interp_fct** (*scipy.interpolate.NearestNDInterpolator*) – scipy Interpolator object containing the NDInterpolator defined by the user
- **total_params** (*float*) – number of parameters

Written: Jorge LLop-Sayson (2021)

`compute_lnprob(element_array)`

Compute log(probability) of an array of numbers wrt a the defined ND interpolator. Negative numbers return a probability of -inf.

Parameters
element_array (*float or np.array of float*) – array of numbers. We want the probability of drawing each of these from the ND interpolator.

Returns
array of log(probability) values, corresponding to the probability of drawing each of the numbers in the input *element_array*.

Return type
numpy array of float

`draw_samples(num_samples)`

Draw positive samples from the ND interpolator. Negative samples will not be returned.

Parameters
num_samples (*float*) – the number of samples to generate.

Returns
samples drawn from the ND interpolator distribution. Array has length *num_samples*.

Return type
numpy array of float

`increment_param_num()`

Increment the index to evaluate the appropriate parameter.

`class orbitize.priors.ObsPrior(epochs, ra_err, dec_err, mtot, period_lims=(0, inf), tp_lims=(-inf, inf), tau_ref_epoch=58849)`

Implements the observation-based priors described in O'Neil+ 2018 (https://ui.adsabs.harvard.edu/abs/2019AJ....158....4O/abstra

Parameters

- **epochs** (*np.array of float*) – array of epochs at which observations are taken [mjd]
- **ra_err** (*np.array of float*) – RA errors of observations [mas]

- **dec_err** (*np.array of float*) – decl errors of observations [mas]
- **mtot** (*float*) – total mass of system [Msol]
- **period_lims** (*2-tuple of float*) – optional lower and upper prior limits for the orbital period [yr]
- **tp_lims** (*2-tuple of float*) – optional lower and upper prior limits for the time of periastron passage [mjd]
- **tau_ref_epoch** (*float*) – epoch [mjd] tau is defined relative to.

Note: This implementation is designed to be mathematically identical to the implementation in O'Neil+ 2018. There are several limitations of our implementation, in particular:

1. *ObsPrior* only works with MCMC (not OFTI)
2. *ObsPrior* only works with relative astrometry (i.e. you can't use RVs or other data types)
3. *ObsPrior* only works when the input astrometry is given in RA/decl. format (i.e. not sep/PA)
4. *ObsPrior* assumes total mass (*mtot*) and parallax (*plx*) are fixed.
5. *ObsPrior* only works for systems with one secondary object (no multi-planet systems)
6. You must use *ObsPrior* with the *orbitize.basis.ObsPriors* orbital basis.

None of these are inherent limitations of the observation-based technique, so let us know if you have a science case that would benefit from implementing one or more of these things!

---

draw_samples(*num_samples*)

Draws *num_samples* samples from uniform distributions in log(per), ecc, and tp. This is used for initializing the MCMC walkers.

> **Warning:** The behavior of orbitize.priors.ObsPrior.draw_samples() is different from the draw_samples() methods of other Prior objects, which draws random samples from the prior itself.

class orbitize.priors.**Prior**

Abstract base class for prior objects. All prior objects should inherit from this class.

Written: Sarah Blunt, 2018

class orbitize.priors.**SinPrior**

This is the probability distribution $p(x) \propto sin(x)$

The domain of this prior is [0,pi].

compute_lnprob(*element_array*)

Compute the prior probability of each element given that its drawn from a sine prior

Parameters
element_array (*float or np.array of float*) – array of paramters to compute the prior probability of

Returns
array of prior probabilities

Return type
np.array

draw_samples(*num_samples*)

Draw samples from a Sine distribution.

Parameters
num_samples (*float*) – the number of samples to generate

Returns
samples ranging from [0, pi) as floats.

Return type
np.array

transform_samples(*u*)

Transform uniform 1D samples, u, to samples drawn from a Sine distribution.

Parameters
u (*array of floats*) – list of samples with values 0 < u < 1.

Returns
1D u samples transformed to a Sine distribution.

Return type
numpy array of floats

class orbitize.priors.**UniformPrior**(*minval*, *maxval*)

This is the probability distribution p(x) propto constant.

Parameters

- **minval** (*float*) – the lower bound of the uniform prior
- **maxval** (*float*) – the upper bound of the uniform prior

compute_lnprob(*element_array*)

Compute the prior probability of each element given that its drawn from this uniform prior

Parameters
element_array (*float or np.array of float*) – array of paramters to compute the prior probability of

Returns
array of prior probabilities

Return type
np.array

draw_samples(*num_samples*)

Draw samples from this uniform distribution.

Parameters
num_samples (*float*) – the number of samples to generate

Returns
samples ranging from [0, pi) as floats.

Return type
np.array

transform_samples(*u*)

Transform uniform 1D samples, u, to samples drawn from a uniform distribution.

Parameters
u (*array of floats*) – list of samples with values 0 < u < 1.

Returns
1D u samples transformed to a uniform distribution.

Return type
numpy array of floats

`orbitize.priors.all_lnpriors(params, priors)`

Calculates log(prior probability) of a set of parameters and a list of priors

Parameters

- **params** (*np.array*) – size of N parameters
- **priors** (*list*) – list of N prior objects corresponding to each parameter

Returns
prior probability of this set of parameters

Return type
float

2.5.10 Read Input

Module to read user input from files and create standardized input for orbitize

`orbitize.read_input.read_file(filename)`

Reads data from any file for use in orbitize readable by `astropy.io.ascii.read()`, including csv format. See the astropy docs.

There are two ways to provide input data to orbitize.

The first way is to provide astrometric measurements, shown with the following example.

Example of an orbitize-readable .csv input file:

```
epoch,object,raoff,raoff_err,decoff,decoff_err,radec_corr,sep,sep_err,pa,pa_err,rv,
↪rv_err
1234,1,0.010,0.005,0.50,0.05,0.025,,,,,
1235,1,,,,,1.0,0.005,89.0,0.1,,
1236,1,,,,,1.0,0.005,89.3,0.3,,
1237,0,,,,,,,,,10,0.1
```

Each row must have `epoch` (in MJD=JD-2400000.5) and `object`. Objects are numbered with integers, where the primary/central object is `0`. If you have, for example, one RV measurement of a star and three astrometric measurements of an orbiting planet, you should put `0` in the `object` column for the RV point, and `1` in the columns for the astrometric measurements.

Each line must also have at least one of the following sets of valid measurements:

- RA and DEC offsets [mas], or
- sep [mas] and PA [degrees East of NCP], or
- RV measurement [km/s]

Note: Columns with no data can be omitted (e.g. if only separation and PA are given, the raoff, deoff, and rv columns can be excluded).

If more than one valid set is given (e.g. RV measurement and astrometric measurement taken at the same epoch), `read_file()` will generate a separate output row for each valid set.

For RA/Dec and Sep/PA, you can also specify a correlation term. This is useful when your error ellipse is tilted with respect to the RA/Dec or Sep/PA. The correlation term is the Pearson correlation coefficient (), which corresponds to the normalized off diagonal term of the covariance matrix. Let's define the covariance matrix as `C = [[C_11, C_12], [C_12, C_22]]`. Here C_11 = quant1_err^2 and C_22 = quant2_err^2 and C_12 is the off diagonal term. Then = `C_12/(sqrt(C_11)sqrt(C_22))`. Essentially it is the covariance normalized by the variance. As such, -1 1. You can specify either as radec_corr or seppa_corr. By definition, both are dimensionless, but one will correspond to RA/Dec and the other to Sep/PA. An example of real world data that reports correlations is this GRAVITY paper where table 2 reports the correlation values and figure 4 shows how the error ellipses are tilted.

Alternatively, you can also supply a data file with the columns already corresponding to the orbitize format (see the example in description of what this method returns). This may be useful if you are wanting to use the output of the *write_orbitize_input* method.

Note: RV measurements of objects that are not the primary should be relative to the barycenter RV. For example, if the barycenter has a RV of 20 +/- 1 km/s, and you've measured an absolute RV for the secondary of 15 +/- 2 km/s, you should input an RV of -5.0 +/- 2.2 for object 1.

Note: When providing data with columns in the orbitize format, there should be no empty cells. As in the example below, when quant2 is not applicable, the cell should contain nan.

Parameters

filename (*str or astropy.table.Table*) – Input filename or the actual table object

Returns

Table containing orbitize-readable input for given object. For the example input above:

```
 epoch  object  quant1 quant1_err  quant2 quant2_err quant12_corr quant_
↪type
float64  int   float64  float64   float64  float64     float64      ␣
↪str5
------- ------ ------- ---------- ------- ---------- ------------ ------
↪----
1234.0      1    0.01      0.005     0.5       0.05      0.025       ␣
↪radec
1235.0      1     1.0      0.005    89.0        0.1        nan       ␣
↪seppa
1236.0      1     1.0      0.005    89.3        0.3        nan       ␣
↪seppa
1237.0      0    10.0        0.1     nan        nan        nan         ␣
↪ rv
```

where `quant_type` is one of "radec", "seppa", or "rv".

If `quant_type` is "radec" or "seppa", the units of quant are mas and degrees, if `quant_type` is "rv", the units of quant are km/s

Return type

astropy.Table

Written: Henry Ngo, 2018

Updated: Vighnesh Nagpal, Jason Wang (2020-2021)

orbitize.read_input.**write_orbitize_input**(*table*, *output_filename*, *file_type='csv'*)

Writes orbitize-readable input as an ASCII file

Parameters

- **table** (*astropy.Table*) – Table containing orbitize-readable input for given object, as generated by the read functions in this module.
- **output_filename** (*str*) – csv file to write to
- **file_type** (*str*) – Any valid write format for astropy.io.ascii. See the astropy docs. Defaults to csv.

(written) Henry Ngo, 2018

2.5.11 Results

class orbitize.results.**Results**(*system=None*, *sampler_name=None*, *post=None*, *lnlike=None*, *version_number=None*, *curr_pos=None*)

A class to store accepted orbital configurations from the sampler

Parameters

- **system** (orbitize.system.System) – System object used to do the fit.
- **sampler_name** (*string*) – name of sampler class that generated these results (default: None).
- **post** (*np.array of float*) – MxN array of orbital parameters (posterior output from orbit-fitting process), where M is the number of orbits generated, and N is the number of varying orbital parameters in the fit (default: None).
- **lnlike** (*np.array of float*) – M array of log-likelihoods corresponding to the orbits described in post (default: None).
- **version_number** (*str*) – version of orbitize that produced these results.
- **data** (*astropy.table.Table*) – output from orbitize.read_input.read_file()
- **curr_pos** (*np.array of float*) – for MCMC only. A multi-D array of the current walker positions that is used for restarting a MCMC sampler.

Written: Henry Ngo, Sarah Blunt, 2018

API Update: Sarah Blunt, 2021

add_samples(*orbital_params*, *lnlikes*, *curr_pos=None*)

Add accepted orbits, their likelihoods, and the orbitize version number to the results

Parameters

- **orbital_params** (*np.array*) – add sets of orbital params (could be multiple) to results
- **lnlike** (*np.array*) – add corresponding lnlike values to results
- **curr_pos** (*np.array of float*) – for MCMC only. A multi-D array of the current walker positions

Written: Henry Ngo, 2018

API Update: Sarah Blunt, 2021

load_results(*filename*, *append=False*)

Populate the `results.Results` object with data from a datafile

Parameters

- **filename** (*string*) – filepath where data is saved
- **append** (*boolean*) – if True, then new data is added to existing object. If False (default), new data overwrites existing object

See the `save_results()` method in this module for information on how the data is structured.

Written: Henry Ngo, 2018

API Update: Sarah Blunt, 2021

plot_corner(*param_list=None*, ***corner_kwargs*)

Wrapper for orbitize.plot.plot_corner

plot_orbits(*object_to_plot=1*, *start_mjd=51544.0*, *num_orbits_to_plot=100*, *num_epochs_to_plot=100*, *square_plot=True*, *show_colorbar=True*, *cmap=<matplotlib.colors.LinearSegmentedColormap object>*, *sep_pa_color='lightgrey'*, *sep_pa_end_year=2025.0*, *cbar_param='Epoch [year]'*, *mod180=False*, *rv_time_series=False*, *plot_astrometry=True*, *plot_astrometry_insts=False*, *fig=None*)

Wrapper for orbitize.plot.plot_orbits

plot_propermotion(*object_to_plot=1*, *start_mjd=44239.0*, *periods_to_plot=1*, *end_year=2030.0*, *alpha=0.05*, *num_orbits_to_plot=100*, *num_epochs_to_plot=100*, *show_colorbar=True*, *cmap=<matplotlib.colors.LinearSegmentedColormap object>*, *cbar_param=None*)

Wrapper for orbitize.plot.plot_propermotion

print_results()

Prints median and 68% credible intervals alongside fitting labels

save_results(*filename*)

Save results.Results object to an hdf5 file

Parameters
filename (*string*) – filepath to save to

Save attributes from the `results.Results` object.

`sampler_name`, `tau_ref_epcoh`, `version_number` are attributes of the root group. `post`, `lnlike`, and `parameter_labels` are datasets that are members of the root group.

Written: Henry Ngo, 2018

API Update: Sarah Blunt, 2021

2.5.12 Sampler

class orbitize.sampler.**MCMC**(*system*, *num_temps=20*, *num_walkers=1000*, *num_threads=1*, *chi2_type='standard'*, *like='chi2_lnlike'*, *custom_lnlike=None*, *prev_result_filename=None*)

MCMC sampler. Supports either parallel tempering or just regular MCMC. Parallel tempering will be run if `num_temps` > 1 Parallel-Tempered MCMC Sampler uses ptemcee, a fork of the emcee Affine-infariant sampler Affine-Invariant Ensemble MCMC Sampler uses emcee.

> **Warning:** may not work well for multi-modal distributions

Parameters

- **system** (system.System) – system.System object
- **num_temps** (*int*) – number of temperatures to run the sampler at. Parallel tempering will be used if num_temps > 1 (default=20)
- **num_walkers** (*int*) – number of walkers at each temperature (default=1000)
- **num_threads** (*int*) – number of threads to use for parallelization (default=1)
- **chi2_type** (*str, optional*) – either "standard", or "log"
- **like** (*str*) – name of likelihood function in `lnlike.py`
- **custom_lnlike** (*func*) – ability to include an addition custom likelihood function in the fit. The function looks like `clnlikes = custon_lnlike(params)` where `params` is a RxM array of fitting parameters, where R is the number of orbital paramters (can be passed in system.compute_model()), and M is the number of orbits we need model predictions for. It returns `clnlikes` which is an array of length M, or it can be a single float if M = 1.
- **prev_result_filename** (*str*) – if passed a filename to an HDF5 file containing a orbitize.Result data, MCMC will restart from where it left off.

Written: Jason Wang, Henry Ngo, 2018

check_prior_support()

Review the positions of all MCMC walkers, to verify that they are supported by the prior space. This function will raise a descriptive ValueError if any positions lie outside prior support. Otherwise, it will return nothing.

(written): Adam Smith, 2021

chop_chains(*burn*, *trim=0*)

Permanently removes steps from beginning (and/or end) of chains from the Results object. Also updates *curr_pos* if steps are removed from the end of the chain.

Parameters

- **burn** (*int*) – The number of steps to remove from the beginning of the chains
- **trim** (*int*) – The number of steps to remove from the end of the chians (optional)

> **Warning:** Does not update bookkeeping arrays within *MCMC* sampler object.

(written): Henry Ngo, 2019

examine_chains(*param_list=None*, *walker_list=None*, *n_walkers=None*, *step_range=None*, *transparency=1*)

Plots position of walkers at each step from Results object. Returns list of figures, one per parameter :param param_list: List of strings of parameters to plot (e.g. "sma1")

If None (default), all parameters are plotted

Parameters

- **walker_list** – List or array of walker numbers to plot If None (default), all walkers are plotted
- **n_walkers** (*int*) – Randomly select *n_walkers* to plot Overrides walker_list if this is set If None (default), walkers selected as per *walker_list*
- **step_range** (*array or tuple*) – Start and end values of step numbers to plot If None (default), all the steps are plotted
- **transparency** (*int or float*) – Determines visibility of the plotted function If 1 (default) results plot at 100% opacity

Returns
Walker position plot for each parameter selected

Return type
List of `matplotlib.pyplot.Figure` objects

(written): Henry Ngo, 2019

run_sampler(*total_orbits*, *burn_steps=0*, *thin=1*, *examine_chains=False*, *output_filename=None*, *periodic_save_freq=None*)

Runs PT MCMC sampler. Results are stored in `self.chain` and `self.lnlikes`. Results also added to `orbitize.results.Results` object (`self.results`)

Note: Can be run multiple times if you want to pause and inspect things. Each call will continue from the end state of the last execution.

Parameters

- **total_orbits** (*int*) – total number of accepted possible orbits that are desired. This equals `num_steps_per_walker` x `num_walkers`
- **burn_steps** (*int*) – optional paramter to tell sampler to discard certain number of steps at the beginning
- **thin** (*int*) – factor to thin the steps of each walker by to remove correlations in the walker steps
- **examine_chains** (*boolean*) – Displays plots of walkers at each step by running *examine_chains* after *total_orbits* sampled.
- **output_filename** (*str*) – Optional filepath for where results file can be saved.
- **periodic_save_freq** (*int*) – Optionally, save the current results into `output_filename` every nth step while running, where n is value passed into this variable.

Returns
the sampler used to run the MCMC

Return type
emcee.sampler object

validate_xyz_positions()

If using the XYZ basis, walkers might be initialized in an invalid region of parameter space. This function fixes that by replacing invalid positions by new randomly generated positions until all are valid.

class orbitize.sampler.**NestedSampler**(*system*)

Implements nested sampling using Dynesty package.

Thea McKenna, Sarah Blunt, & Lea Hirsch 2024

ptform(*u*)

Prior transform function.

Parameters
u (*array of floats*) – list of samples with values 0 < u < 1.

Returns

1D u samples transformed to a chosen Prior
Class distribution.

Return type
numpy array of floats

run_sampler(*static=False*, *bound='multi'*, *pfrac=1.0*, *num_threads=1*, *start_method='fork'*, *run_nested_kwargs={}*)

Runs the nested sampler from the Dynesty package.

Parameters

- **static** (*bool*) – True if using static nested sampling, False if using dynamic.
- **bound** (*str*) – Method used to approximately bound the prior using the current set of live points. Conditions the sampling methods used to propose new live points. See https://dynesty.readthedocs.io/en/latest/quickstart.html#bounding-options for complete list of options.
- **pfrac** (*float*) – posterior weight, between 0 and 1. Can only be altered for the Dynamic nested sampler, otherwise this keyword is unused.
- **num_threads** (*int*) – number of threads to use for parallelization (default=1)
- **start_method** (*str*) – multiprocessing start method. Default "fork," which won't work on all OS. Change to "spawn" if you get an error, and make sure you run your orbitize! script inside an if __name__=='__main__' condition to protect entry points.
- **run_nested_kwargs** (*dict*) – dictionary of keywords to be passed into dynesty.Sampler.run_nested()

Returns

numpy.array of float: posterior samples

int: number of iterations it took to converge

Return type
tuple

class orbitize.sampler.OFTI(*system*, *like='chi2_lnlike'*, *custom_lnlike=None*, *chi2_type='standard'*)

OFTI Sampler

Parameters

- **system** (system.System) – system.System object
- **like** (*string*) – name of likelihood function in lnlike.py
- **custom_lnlike** (*func*) – ability to include an addition custom likelihood function in the fit. The function looks like clnlikes = custon_lnlike(params) where params is a RxM array of fitting parameters, where R is the number of orbital paramters (can be passed in system.compute_model()), and M is the number of orbits we need model predictions for. It returns clnlikes which is an array of length M, or it can be a single float if M = 1.

Written: Isabel Angelo, Sarah Blunt, Logan Pearce, 2018

prepare_samples(*num_samples*)

Prepare some orbits for rejection sampling. This draws random orbits from priors, and performs scale & rotate.

Parameters

num_samples (*int*) – number of orbits to draw and scale & rotate for OFTI to run rejection sampling on

Returns

array of prepared samples. The first dimension has size of num_samples. This should be passed into OFTI.reject()

Return type

np.array

reject(*samples*)

Runs rejection sampling on some prepared samples.

Parameters

samples (*np.array*) – array of prepared samples. The first dimension has size num_samples. This should be the output of prepare_samples().

Returns

np.array: a subset of samples that are accepted based on the data.

np.array: the log likelihood values of the accepted orbits.

Return type

tuple

run_sampler(*total_orbits*, *num_samples=10000*, *num_cores=None*, *OFTI_warning=60.0*)

Runs OFTI in parallel on multiple cores until we get the number of total accepted orbits we want.

Parameters

- **total_orbits** (*int*) – total number of accepted orbits desired by user
- **num_samples** (*int*) – number of orbits to prepare for OFTI to run rejection sampling on. Defaults to 10000.
- **num_cores** (*int*) – the number of cores to run OFTI on. Defaults to number of cores availabe.
- **OFTI_warning** (*float*) – if OFTI doesn't accept a single orbit before this amount of time (in seconds), print a warning suggesting to try MCMC. If None, don't print a warning.

Returns
array of accepted orbits. Size: total_orbits.

Return type
np.array

Written by: Vighnesh Nagpal(2019)

class orbitize.sampler.**Sampler**(*system*, *like='chi2_lnlike'*, *custom_lnlike=None*, *chi2_type='standard'*)

Abstract base class for sampler objects. All sampler objects should inherit from this class.

Written: Sarah Blunt, 2018

2.5.13 System

class orbitize.system.**System**(*num_secondary_bodies*, *data_table*, *stellar_or_system_mass*, *plx*, *mass_err=0*, *plx_err=0*, *restrict_angle_ranges=False*, *tau_ref_epoch=58849*, *fit_secondary_mass=False*, *hipparcos_IAD=None*, *gaia=None*, *fitting_basis='Standard'*, *use_rebound=False*)

A class to store information about a system (data & priors) and calculate model predictions given a set of orbital parameters.

Parameters

- **num_secondary_bodies** (*int*) – number of secondary bodies in the system. Should be at least 1.
- **data_table** (*astropy.table.Table*) – output from orbitize.read_input.read_file()
- **stellar_or_system_mass** (*float*) – mass of the primary star (if fitting for dynamical masses of both components, for example when you have both astrometry and RVs) or total system mass (if fitting for total system mass only, as in the case of a vanilla 2-body fit using relative astrometry only) [M_sol]
- **plx** (*float*) – mean parallax of the system, in mas
- **mass_err** (*float, optional*) – uncertainty on stellar_or_system_mass, in M_sol
- **plx_err** (*float, optional*) – uncertainty on plx, in mas
- **restrict_angle_ranges** (*bool, optional*) – if True, restrict the ranges of the position angle of nodes to [0,180) to get rid of symmetric double-peaks for imaging-only datasets.
- **tau_ref_epoch** (*float, optional*) – reference epoch for defining tau (MJD). Default is 58849 (Jan 1, 2020).
- **fit_secondary_mass** (*bool, optional*) – if True, include the dynamical mass of the orbiting body as a fitted parameter. If this is set to False, stellar_or_system_mass is taken to be the total mass of the system. (default: False)
- **hipparcos_IAD** (orbitize.hipparcos.HipparcosLogProb) – an object containing information & precomputed values relevant to Hipparcos IAD fitting. See hipparcos.py for more details.
- **gaia** (orbitize.gaia.GaiaLogProb) – an object containing information & precomputed values relevant to Gaia astrometrry fitting. See gaia.py for more details.
- **fitting_basis** (*str*) – the name of the class corresponding to the fitting basis to be used. See basis.py for a list of implemented fitting bases.

- **use_rebound** (*bool*) – if True, use an n-body backend solver instead of a Keplerian solver.

Priors are initialized as a list of orbitize.priors.Prior objects and stored in the variable `System.sys_priors`. You should initialize this class, then overwrite priors you wish to customize. You can use the `System.param_idx` attribute to figure out which indices correspond to which fitting parameters. See the "changing priors" tutorial for more detail.

Written: Sarah Blunt, Henry Ngo, Jason Wang, 2018

compute_all_orbits(*params_arr*, *epochs=None*, *comp_rebound=False*)

Calls orbitize.kepler.calc_orbit and optionally accounts for multi-body interactions. Also computes total quantities like RV (without jitter/gamma)

Parameters

- **params_arr** (*np.array of float*) – RxM array of fitting parameters, where R is the number of parameters being fit, and M is the number of orbits we need model predictions for. Must be in the same order documented in `System()` above. If M=1, this can be a 1d array.
- **epochs** (*np.array of float*) – epochs (in mjd) at which to compute orbit predictions.
- **comp_rebound** (*bool, optional*) – A secondary optional input for use of N-body solver Rebound; by default, this will be set to false and a Kepler solver will be used instead.

Returns

raoff (np.array of float): N_epochs x N_bodies x N_orbits array of
RA offsets from barycenter at each epoch.

decoff (np.array of float): N_epochs x N_bodies x N_orbits array of
Dec offsets from barycenter at each epoch.

vz (np.array of float): N_epochs x N_bodies x N_orbits array of
radial velocities at each epoch.

Return type
tuple

compute_model(*params_arr*, *use_rebound=False*)

Compute model predictions for an array of fitting parameters. Calls the above compute_all_orbits() function, adds jitter/gamma to RV measurements, and propagates these predictions to a model array that can be subtracted from a data array to compute chi2.

Parameters

- **params_arr** (*np.array of float*) – RxM array of fitting parameters, where R is the number of parameters being fit, and M is the number of orbits we need model predictions for. Must be in the same order documented in `System()` above. If M=1, this can be a 1d array.
- **use_rebound** (*bool, optional*) – A secondary optional input for use of N-body solver Rebound; by default, this will be set to false and a Kepler solver will be used instead.

Returns

np.array of float: Nobsx2xM array model predictions. If M=1, this is
a 2d array, otherwise it is a 3d array.

np.array of float: Nobsx2xM array jitter predictions. If M=1, this is
a 2d array, otherwise it is a 3d array.

Return type
tuple of

convert_data_table_radec2seppa(*body_num=1*)

Converts rows of self.data_table given in radec to seppa. Note that self.input_table remains unchanged.

Parameters
body_num (*int*) – which object to convert (1 = first planet)

save(*hf*)

Saves the current object to an hdf5 file

Parameters
hf (*h5py._hl.files.File*) – a currently open hdf5 file in which to save the object.

orbitize.system.**generate_synthetic_data**(*orbit_frac*, *mtot*, *plx*, *ecc=0.5*, *inc=0.7853981633974483*, *argp=0.7853981633974483*, *lan=0.7853981633974483*, *tau=0.8*, *num_obs=4*, *unc=2*)

Generate an orbitize-table of synethic data

Parameters

- **orbit_frac** (*float*) – percentage of orbit covered by synthetic data
- **mtot** (*float*) – total mass of the system [M_sol]
- **plx** (*float*) – parallax of system [mas]
- **num_obs** (*int*) – number of observations to generate
- **unc** (*float*) – uncertainty on all simulated RA & Dec measurements [mas]

Returns

- *astropy.table.Table*: data table of generated synthetic data
- float: the semimajor axis of the generated data

Return type
2-tuple

orbitize.system.**radec2seppa**(*ra*, *dec*, *mod180=False*)

Convenience function for converting from right ascension/declination to separation/ position angle.

Parameters

- **ra** (*np.array of float*) – array of RA values, in mas
- **dec** (*np.array of float*) – array of Dec values, in mas
- **mod180** (*Bool*) – if True, output PA values will be given in range [180, 540) (useful for plotting short arcs with PAs that cross 360 during observations) (default: False)

Returns
(separation [mas], position angle [deg])

Return type
tuple of float

orbitize.system.**seppa2radec**(*sep*, *pa*)

Convenience function to convert sep/pa to ra/dec

Parameters

- **sep** (*np.array of float*) – array of separation in mas

- **pa** (*np.array of float*) – array of position angles in degrees

Returns
(ra [mas], dec [mas])

Return type
tuple

orbitize.system.**transform_errors**(*x1*, *x2*, *x1_err*, *x2_err*, *x12_corr*, *transform_func*, *nsamps=100000*)

Transform errors and covariances from one basis to another using a Monte Carlo apporach

Parameters

- **x1** (*float*) – planet location in first coordinate (e.g., RA, sep) before transformation
- **x2** (*float*) – planet location in the second coordinate (e.g., Dec, PA) before transformation)
- **x1_err** (*float*) – error in x1
- **x2_err** (*float*) – error in x2
- **x12_corr** (*float*) – correlation between x1 and x2
- **transform_func** (*function*) – function that transforms between (x1, x2) and (x1p, x2p) (the transformed coordinates). The function signature should look like: *x1p, x2p = transform_func(x1, x2)*
- **nsamps** – number of samples to draw more the Monte Carlo approach. More is slower but more accurate.

2.6 orbitize! Manual

2.6.1 Intro to orbitize!

orbitize! hinges on the two-body problem, which describes the paths of two bodies gravitationally bound to each other as a function of time, given parameters determining the position and velocity of both objects at a particular epoch. There are many basis sets (orbital bases) that can be used to describe an orbit, which can then be solved using Kepler's equation, but first it is important to be explicit about coordinate systems.

Note: For an interactive visualization to define and help users understand our coordinate system, you can check out this GitHub tutorial.

There is also a YouTube video with use and explanation of the coordinate system by Sarah Blunt.

In its "standard" mode, orbitize! assumes that the user only has relative astrometric data to fit. In the orbitize! coordinate system, relative R.A. and declination can be expressed as the following functions of orbital parameters

$$\Delta R.A. = \pi a(1 - ecosE)[cos^2\frac{i}{2}sin(f + \omega_p + \Omega) - sin^2\frac{i}{2}sin(f + \omega_p - \Omega)]$$

$$\Delta decl. = \pi a(1 - ecosE)[cos^2\frac{i}{2}cos(f + \omega_p + \Omega) + sin^2\frac{i}{2}cos(f + \omega_p - \Omega)]$$

where , , ω_p, , and are orbital parameters, and is the system parallax. f is the true anomaly, and E is the eccentric anomaly, which are related to elapsed time through Kepler's equation and Kepler's third law

$$M = 2\pi(\frac{t}{P} - (\tau - \tau_{ref}))$$

$$(\frac{P}{yr})^2 = (\frac{a}{au})^3(\frac{M_\odot}{M_{tot}})$$
$$M = E - esinE$$
$$f = 2tan^{-1}[\sqrt{\frac{1+e}{1-e}}tan\frac{E}{2}]$$

`orbitize!` employs two Kepler solvers to convert between mean and eccentric anomaly: one that is efficient for the highest eccentricities, and Newton's method, which in other cases is more efficient for solving for the average orbit. See Blunt et al. (2020) for more detail.

From scrutinizing the above sets of equations, one may observe a few important degeneracies:

1. Individual component masses do not show up anywhere in this equation set.

2. The degeneracy between semimajor axis , total mass $_{tot}$, and parallax . If we just had relative astrometric measurements and no external knowledge of the system parallax, we would not be able to distinguish between a system that has larger distance and larger semimajor axis (and therefore larger total mass, assuming a fixed period) from a system that has smaller distance, smaller semimajor axis, and smaller total mass.

3. The argument of periastron ω_p and the position angle of nodes . The above defined R.A. and decl. functions are invariant to the transformation:

$$\omega_p' = \omega_p + \pi$$
$$\Omega' = \Omega - \pi$$

which creates a 180° degeneracy between particular values of ω_p and , and a characteristic "double-peaked" structure in marginalized 1D posteriors of these parameters.

Solutions to breaking degeneracies 1 and 3 can be found in the next section.

2.6.2 Using Radial Velocities

In the `orbitize!` coordinate system, and relative to the system barycenter, the radial velocity of the planet due to the gravitational influence of the star is:

$$rv_p(f) = \sqrt{\frac{G}{(1-e^2)}}M_*sini(M_{tot})^{-1/2}a^{-1/2}(cos(\omega_p + f) + ecos\omega_p)$$
$$rv_*(f) = \sqrt{\frac{G}{(1-e^2)}}M_psini(M_{tot})^{-1/2}a^{-1/2}(cos(\omega_* + f) + ecos\omega_*)$$

where is the argument of periastron of the star's orbit, which is equal to + 180°.

In these equations, the individual component masses m and m enter. This means radial velocity measurements break the total mass degeneracy and enable measurements of individual component masses ("dynamical" masses). However it is crucial to keep in mind that radial velocities of a planet do not enable dynamical mass measurements of the planet itself, but of the star. Radial velocity measurements also break the / degeneracy discussed in the previous section, uniquely orienting the orbit in 3D space.

`orbitize!` can perform joint fits of RV and astrometric data in two different ways, which have complementary applications.

The first method is automatically triggered when an `orbitize!` user inputs radial velocity data. `orbitize!` automatically parses the data, sets up an appropriate model, then runs the user's Bayesian computation algorithm of choice to jointly constrain all free parameters in the fit. `orbitize!` can handle both primary and secondary RVs, and fits for the appropriate dynamical masses when RVs are present; when primary RVs are included, `orbitize!` fits for the dynamical masses of secondary objects, and vice versa. Instrumental nuisance parameters (RV zeropoint offset, , and

white noise jitter,) for each RV instrument are also included as additional free parameters in the fit if the user specifies different instrument names in the data file.

The second method of jointly fitting RV and astrometric data in `orbitize!` separates out the fitting of radial velocities and astrometry, enabling a user to fit "one at a time," and combine the results in a Bayesian framework. First, a user performs a fit to just the radial velocity data using, for example, radvel (but can be any radial velocity orbit-fitting code). The user then feeds the numerical posterior samples into `orbitize!` through the `orbitize.priors.KDEPrior` object. This prior creates a representation of the prior using kernel density estimation (kernel density estimation), which can then be used to generate random prior samples or compute the prior probability of a sample orbit. Importantly, this prior preserves covariances between input parameters, allowing `orbitize!` to use an accurate representation of the RV posterior to constrain the fit. This method can be referred to as the "posteriors as priors" method, since posteriors output from a RV fitting code are, through KDE sampling, being applied as priors in `orbitize!` .

More coming soon!

CHAPTER

THREE

CHANGELOG:

3.0.0 (2024-4-15)

- implementation of Hipparcos-Gaia catalog of accelerations fitting! (@semaphoreP)
- fit arbitrary absolute astrometry (@sblunt)
- implement O'Neil observation-based priors (@sblunt/@clarissardoo)
- discuss MCMC autocorrelation in MCMC tutorial (@michaelkmpoon)
- add time warning if OFTI doesn't accept an orbit in first 60 s (@michaelkmpoon)
- add first parts of orbitize! manual (@sofiacovarrubias/@sblunt)
- bugfix for rebound MCMC fits (issue #357; @sblunt)
- implementation of residual plotting method for orbit plots (@Saanikachoudhary and @semaphoreP)
- plot companion RVs (@chihchunhsu)
- add documentation about referencing issues when modifying priors to tutorial (@wcroberson)

2.2.2 (2023-06-30)

- tests now overwrite any generated text files (@sblunt)

2.2.1 (2023-06-28)

- tau_to_tp function now accepts array of after_date (@tomasstolker/@semaphoreP)

2.2.0 (2023-06-21)

- set up new CI system using GH actions (@sblunt)
- removed radvel as dependency, and moved radvel_utils subpackage to new dir (@sblunt). This is a breaking change for users of orbitize.radvel_utils.

2.1.4 (2023-06-20)

- unit tests hotfixes (@semaphoreP)
- use forked ptemcee (@sblunt)

2.1.3 (2023-02-07)

- Compatibility with numpy v1.24 (issue #330 and #331; @tomasstolker)

2.1.2 (2022-08-31)

- Bugfix for saving/loading fits using IAD (issue #324; @sblunt)

2.1.1 (2022-05-24)

- Hotfix for one of the log-chi2 unit tests (@sblunt)

2.1.0 (2022-05-24)

- Added a (more numerically stable) log-chi2 option for calculating likelihood (@Mireya-A and @lhirsch238)

2.0.1 (2022-04-22)

- Addressed plotting bugs: issues #316/#309, #314, #311 (@semaphoreP)
- Made Gaia module runnable without internet and added some Gaia/Hipparcos unit tests (@sblunt)

2.0.0 (2021-10-13)

This is the official release of orbitize! version 2.

Big changes:

- Fit Gaia positions (@sblunt)
- New plotting module & API (@sblunt)
- Relative planet RVs now officially supported & tested (@sblunt)
- GPU Kepler solver (@devincody)
- RV end-to-end test added (@vighnesh-nagpal)

Small changes:

- Hipparcos calculation bugfix (@sblunt)
- v1 results backwards compatibility bugfix (@sblunt)
- windows install docs update (@sblunt
- basis bugfix with new API (@TirthDS, @sblunt)
- handle Hipparcos 2021 data format (@sblunt)
- clarify API on mtot/mstar (@lhirsch238, @sblunt)

2.0b1 (2021-09-03)

This is the beta release of orbitize! version 2.

Big changes:

- N-body Kepler solver backend! (@sofiacovarrubias)
- Fitting in XYZ orbital basis! (@rferrerc)
- API for fitting in arbitrary new orbital bases! (@TirthDS)
- compute_all_orbits separated out, streamlining stellar astrometry & RV calculations (@sblunt)
- Hip IAD! (@sblunt)
- param_idx now used everywhere under the hood (system parsing updated) (@sblunt)
- KDE prior added (inspiration=training on RV fits) (@jorgellop)

Small changes:

- HD 4747 rv data file fix for the RV tutorial (@lhirsch238)
- Add check_prior_support to sampler.MCMC (@adj-smith)
- Update example generation code in MCMC v OFTI tutorial (@semaphoreP)
- Fixed plotting bug (issue #243) (@TirthDS)
- Expand FAQ section (@semaphoreP)

- use astropy tables in results (@semaphoreP)
- Expand converge section of MCMC tutorial (@semaphoreP)
- Deprecated functions and deprecation warnings officially removed (@semaphoreP)
- Fix logic in setting of track_planet_perturbs (@sblunt)
- Fix plotting error if orbital periods are > 1e9 days (@sblunt)
- Add method for printing results of a fit (@sblunt)

1.16.1 (2021-06-27)

- Fixed chop_chains() function to copy original data over when updating Results object (@TirthDS)

1.16.0 (2021-06-23)

- User-defined prior on PAN were not being applied if OFTI is used; fixed (@sblunt)
- Dates in HD 4747 data file were incorrect; fixed (@lhirsch238)

1.15.5 (2021-07-20)

- Addressed issue #177, giving *Results* and *Sampler* classes a parameter label array (@sblunt)
- Fixed a bug that was causing RA/Dec data points to display wrong in orbit plots (@sblunt)

1.15.4 (2021-06-18)

- Bugfix for issue #234 (@semaphoreP, @adj-smith)

1.15.3 (2021-06-07)

- Add codeastro mode to pytest that prints out a SECRET CODE if tests pass omgomg (@semaphoreP)

1.15.2 (2021-05-11)

- Fixed backwards-compatibility bug with version numbers and saving/loading (@semaphoreP, @wbalmer)

1.15.1 (2021-03-29)

- Fixed bug where users with Results objects from v<14.0 couldn't load using v>=14.0 (@semaphoreP, @wbalmer)
- Fixed order of Axes objects in Advanced Plotting tutorial (@wbalmer, @sblunt)

1.15.0 (2021-02-23)

- Handle covariances in input astrometry (@semaphoreP)

1.14.0 (2021-02-12)

- Version number now saved in results object (@hgallamore)
- Joint RV+astrometry fits can now handle different RV instruments! (@vighnesh-nagpal, @Rob685, @lhirsch238)
- New "FAQ" section added to docs (@semaphoreP)
- Bugfix for multiplanet code (@semaphoreP) introduced in PR #192
- now you can pass a preexisting Figure object into `results.plot_orbit` (@sblunt)
- colorbar label is now "Epoch [year]" (@sblunt)
- corner plot maker can now handle fixed parameters without crashing (@sblunt)

1.13.1 (2021-01-25)

- `compute_sep` in `radvel_utils` submodule now returns `mp` (@sblunt)

- `astropy._erfa` was deprecated (now in separate package). Dependencies updated. (@sblunt)

1.13.0 (2020-11-8)

- Added `radvel-utils` submodule which allows users to calculate projected separation posteriors given RadVel chains (@sblunt)
- Fixed a total mass/primary mass mixup bug that was causing problems for equal-mass binary RV+astrometry joint fits (@sblunt)
- Bugfix for multiplanet perturbation approximation: now only account for inner planets only when computing perturbations (@semaphoreP)

1.12.1 (2020-9-6)

- `tau_ref_epoch` is now set to Jan 1, 2020 throughout the code (@semaphoreP)
- `restrict_angle_ranges` keyword now works as expected for OFTI (@sblunt)

1.12.0 (2020-8-28)

- Compatibility with `emcee>=3` (@sblunt)

1.11.3 (2020-8-20)

- Save results section of OFTI tutorial now current (@rferrerc)
- Modifying MCMC initial positions tutorial documentation now uses correct orbital elements (@rferrerc)

1.11.2 (2020-8-10)

- Added transparency option for plotting MCMC chains (@sofiacovarrubias)
- Removed some redundant code (@MissingBrainException)

1.11.1 (2020-6-11)

- Fixed a string formatting bug causing corner plots to fail for RV+astrometry fits

1.11.0 (2020-4-14)

- Multiplanet support!
- Changes to directory structure of sample data files
- Fixed a bug that was causing corner plots to fail on loaded results objects

1.10.0 (2020-3-6)

- Joint RV + relative astrometry fitting capabilities!
- New tutorial added

1.9.0 (2020-1-24)

- Require astropy>=4
- Minor documentation upgrades
- **This is the first Python 2 noncompliant version**

1.8.0 (2020-1-24)

- Bugfixes related to numpy and astropy upgrades
- **This is the last version that will support Python 2**

1.7.0 (2019-11-10)

- Default corner plots now display angles in degrees instead of radians

- Add a keyword for plotting orbits that cross PA=360

1.6.0 (2019-10-1)

- Mikkola solver now implemented in C-Kepler solver
- Fixed a bug with parallel processing for OFTI
- Added orbit vizualisation jupyter nb show-me-the-orbit to docs/tutorials
- New methods for viewing/chopping MCMC chains
- Require `emcee<3` for now

1.5.0 (2019-9-9)

- Parallel processing for OFTI.
- Fixed a bug converting errors in RA/Dec to sep/PA in OFTI.
- OFTI and MCMC now both return likelihood, whereas before one returned posterior.
- Updated logic for restricting Omega and omega bounds.

1.4.0 (2019-7-15)

- API change to lay the groundwork for dynamical mass calculation.
- JeffreysPrior -> LogUniformPrior
- New tutorials.
- Added some informative error messages for input tables.
- Bugfixes.

1.3.1 (2019-6-19)

- Bugfix for RA/Dec inputs to the OFTI sampler (Issue #108).

1.3.0 (2019-6-4)

- Add ability to customize date of tau definition.
- Sampler now saves choice of tau reference with results.
- Default tau value is now Jan 1, 2020.
- Small bugfixes.

1.2.0 (2019-3-21)

- Remove unnecessary `astropy` date warnings.
- Add custom likelihood function.
- Add progress bar for `ptemcee` sampler.
- Add customizable color axis for orbit plots.
- Small bugfixes.

1.1.0 (2019-1-6)

- Add sep/PA panels to orbit plot.
- `GaussianPrior` now operates on only positive numbers by default.

1.0.2 (2018-12-4)

- Expand input reading functionality.

- Bugfixes for MCMC.

1.0.1 (2018-11-20)

- Bugfix for building on CentOS machines.

1.0.0 (2018-10-30)

- Initial release.

PYTHON MODULE INDEX

O

INDEX