radial Documentation

Release 0.5a

Leonardo A. dos Santos

Aug 09, 2017

Contents:

`radial` is a tool to work with the radial velocities of stars that have objects orbiting them, such as planets or stellar companions. The code is completely object-oriented and designed to be used with Python scripts and Jupyter Notebooks. If you find any bugs or have questions and suggestions about the code or the documentation, feel free to submit an issue on GitHub. `radial` is fully compatible with Python 3, and may have incompatibility issues with Python 2.

CHAPTER 1

Installation

`radial` currently works with Python 3; compatibility with Python 2 was not tested. The current version has the following dependencies:

- numpy
- scipy
- matplotlib
- astropy
- lmfit
- emcee
- corner

In order to install the software, download the source code and run:

```
python setup.py install
```

or:

```
python setup.py develop
```

if you intend on developing the code as you use it.

CHAPTER 2

Examples

The following examples illustrate the usage of `radial` to compute radial velocities and fitting orbital parameters to observed data.

The basics of radial velocities

We can almost completely characterize the orbits of massive bodies around a star using a set of five orbital parameters for each body. Different parametrization schemes use a different set of five parameters, depending on what kind of optimization is needed, but for this tutorial we will use the one from Murray & Correia (2010):

- K: radial velocity semi-amplitude
- T: orbital period
- e: eccentricity of the orbit
- ω: argument of periapse
- t_0: time of periapse passage

These parameters exist for each body orbiting a star.

In this notebook, we will play around with the package `radial` to simulate radial velocity curves for a star orbited by bodies with different orbital parameters.

```
from radial import body
import astropy.units as u
import numpy as np
import matplotlib.pyplot as plt
```

Let's start with a very familiar object: the Sun.

Note: if you are too lazy to convert units like me, I recommend using the `astropy.units` module.

```
sun = body.MainStar(mass=1 * u.solMass, name='Sun')
```

Now let's setup a couple of companions for the Sun. How about Earth and Jupiter?

Thei radial velocity semi-amplitude is not easily accessible from online databases, but we can compute them using the following equation:

$$K = \frac{m \sin I}{m + M} \frac{2\pi}{T} \frac{a}{\sqrt{1 - e^2}},$$

where m is the companion mass, M is the mass of the main star, I is the inclination angle between the reference plane and the axis of the orbit (let's consider $I = \pi/2$ in this example) and a is the semi-major axis of the orbit. All these parameters are easily accessible to us.

```
def compute_k(mass, period, semia, ecc, i=np.pi * u.rad):
    return mass / (mass + 1 * u.solMass) * 2 * np.pi / period * semia / (1 - ecc **␣
↪2) ** 0.5

# Computing K for the Earth
mass_e = 1 * u.earthMass
semia_e = 1.00000011 * u.AU
period_e = 1 * u.yr
ecc_e = 0.01671022
k_e = compute_k(mass_e, period_e, semia_e, ecc_e)

# Computing K for Jupiter
mass_j = 1 * u.jupiterMass
semia_j = 5.2026 * u.AU
period_j = 11.8618 * u.yr
ecc_j = 0.048498
k_j = compute_k(mass_j, period_j, semia_j, ecc_j)

# Setting up the companions
earth = body.Companion(main_star=sun,
                       k = k_e,
                       period_orb=period_e,
                       t_0=2457758.01181 * u.d,     # Time of periastron passage,␣
↪Julian Date
                       omega=114.207 * u.deg,       # Argument of periapsis/
↪perihelion
                       ecc=ecc_e)

jupiter = body.Companion(main_star=sun,
                         k=k_j,
                         period_orb=period_j,
                         t_0=2455636.95833 * u.d,
                         omega=273.867 * u.deg,
                         ecc=ecc_j)
```

The next step is to setup the Solar System with the Sun and its planetary companions. We need to state what is the time window that we want to simulate in Julian Dates.

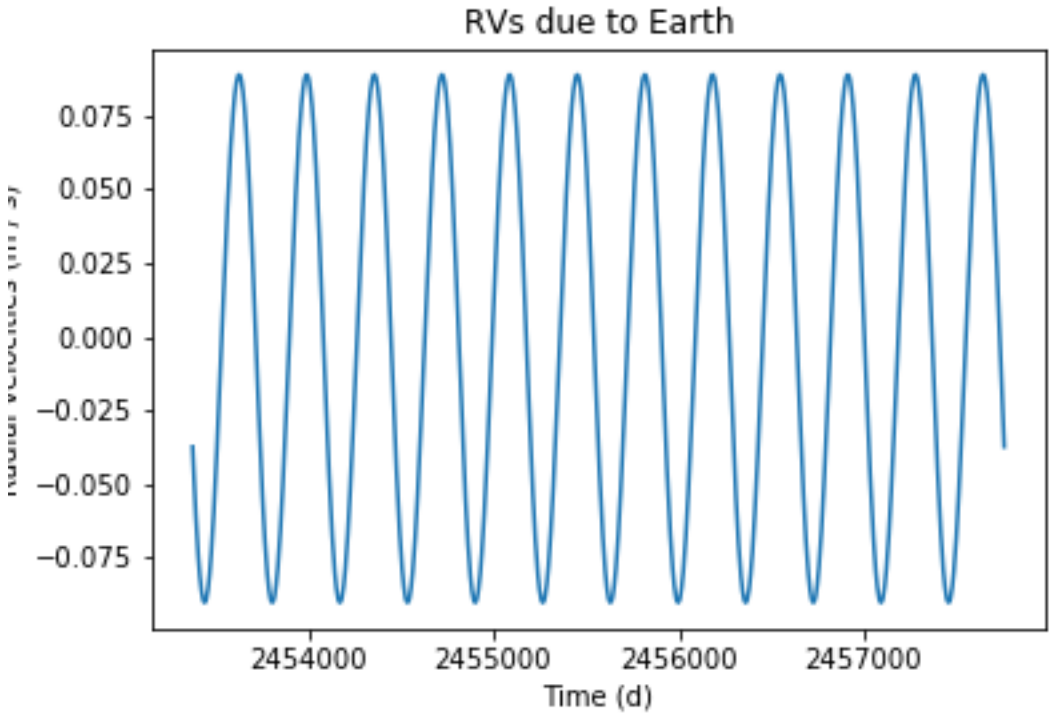
```
time = np.linspace(2453375, 2457758, 1000) * u.d    # ~12 years

# The Solar System
sys = body.System(main_star=sun,
                  companion=[earth, jupiter],
                  time=time)
```

Now to compute the radial velocities of the Sun due to Earth and Jupiter:

```
sys.compute_rv()
```

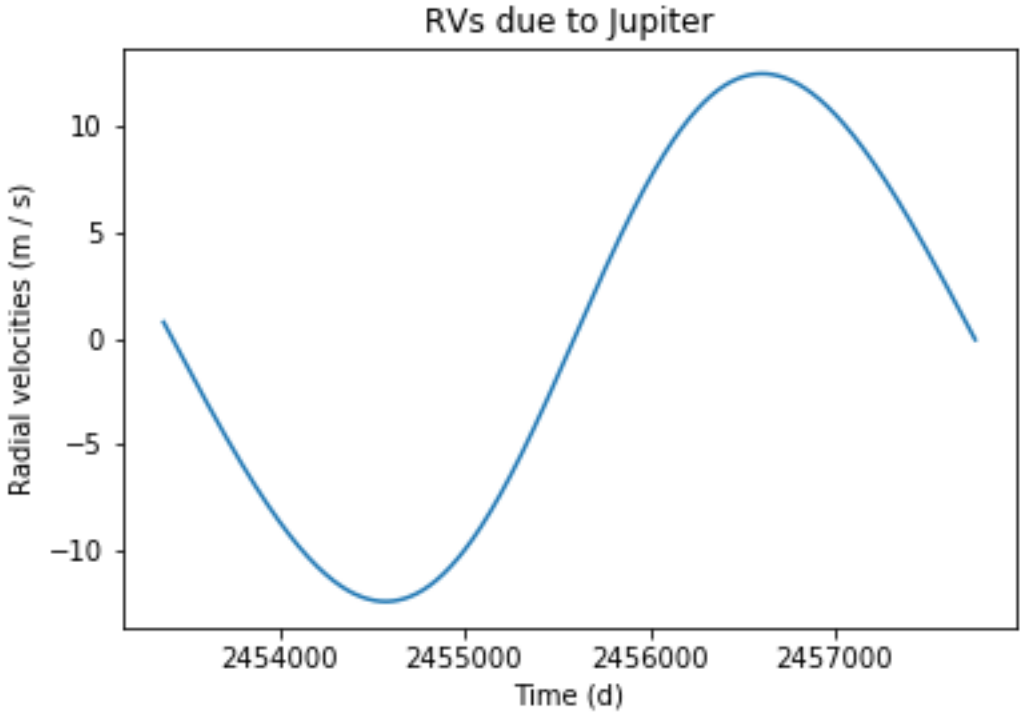
With the radial velocities finally computed, it's time to plot them. Let's take a look at the RVs caused by the Earth on the Sun:

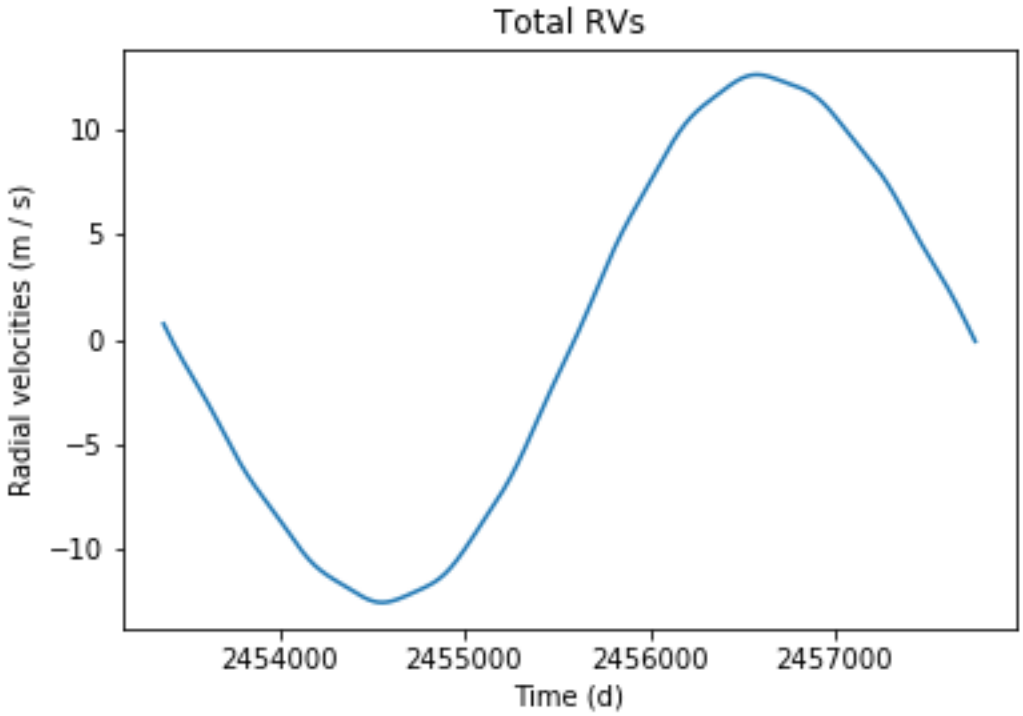
```
sys.plot_rv(0, 'RVs due to Earth')
plt.show()
```



And the total RVs (including Jupiter) are shown below:

```
sys.plot_rv(1, 'RVs due to Jupiter')
sys.plot_rv(plot_title='Total RVs')
plt.show()
```


RVs due to Jupiter
Radial velocities (m / s)
10
5
0
−5
−10
2454000
2455000
2456000
2457000
Time (d)


Total RVs
Radial velocities (m / s)
10
5
0
−5
−10
2454000
2455000
2456000
2457000
Time (d)

The binary star HIP 67620

Our objective in this notebook is to use radial velocity data of the solar twin HIP 67620 to estimate the projected mass, separation and other orbital parameters of its companion. We start by importing the necessary packages. Notice that we will specifically import the modules `orbit`, `estimate`, and `dataset` from the `radial` package.

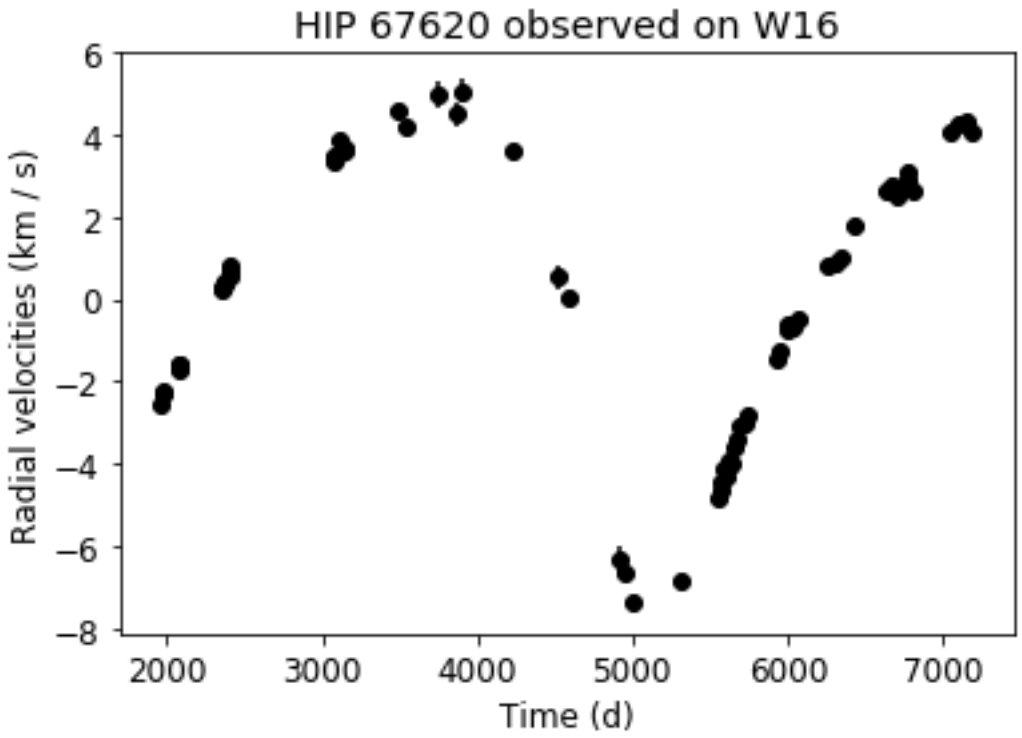
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.pylab as pylab
import astropy.units as u
from radial import estimate, dataset
```

We then extract the data from the text files located in the `tests` folder. They will be stored in `RVDataSet` objects, which are defined in the `dataset` module.

```
harps = dataset.RVDataSet(file='../../tests/HIP67620_HARPS.dat', # File name
                          t_offset=-2.45E6,                  # Time offset (units of␣
↪days)
                          rv_offset='subtract_mean',         # RV offset
                          instrument_name='HARPS',
                          target_name='HIP 67620',
                          skiprows=1,    # Number of rows to skip in the data file
                          t_col=5,       # Column corresponding to time in the data␣
↪file
                          rv_col=6,      # Column corresponding to RVs
                          rv_unc_col=7   # Column corresponding to RV ucnertainties
                         )
aat = dataset.RVDataSet(file='../../tests/HIP67620_AAT.dat', t_offset=-2.45E6, rv_
↪offset='subtract_mean',
                          instrument_name='AATPS', target_name='HIP 67620', delimiter=
↪',')
w16 = dataset.RVDataSet(file='../../tests/HIP67620_WF16.dat', t_offset=-5E4, rv_
↪offset='subtract_mean',
                        instrument_name='W16', target_name='HIP 67620', t_col=1,
                        rv_col=3, rv_unc_col=4)
```

We can visualize the radial velocities by running the function `plot()` of a given `dataset` object. For instance:

```
w16.plot()
```



Now that we have the data, how do we estimate the orbital parameters of the system? We use the methods and functions inside the *estimate* module. But first, we need to provide an initial guess for the orbital parameters. They are:

- `k`: radial velocity semi-amplitude K (in m/s)
- `period`: orbital period T (in days)
- `t0`: time of periastron passage t_0 (in days)
- `omega`: argument of periapse ω (in radians)
- `ecc`: eccentricity of the orbit e
- `gamma_X`: RV offset γ of the dataset number X (in m/s)

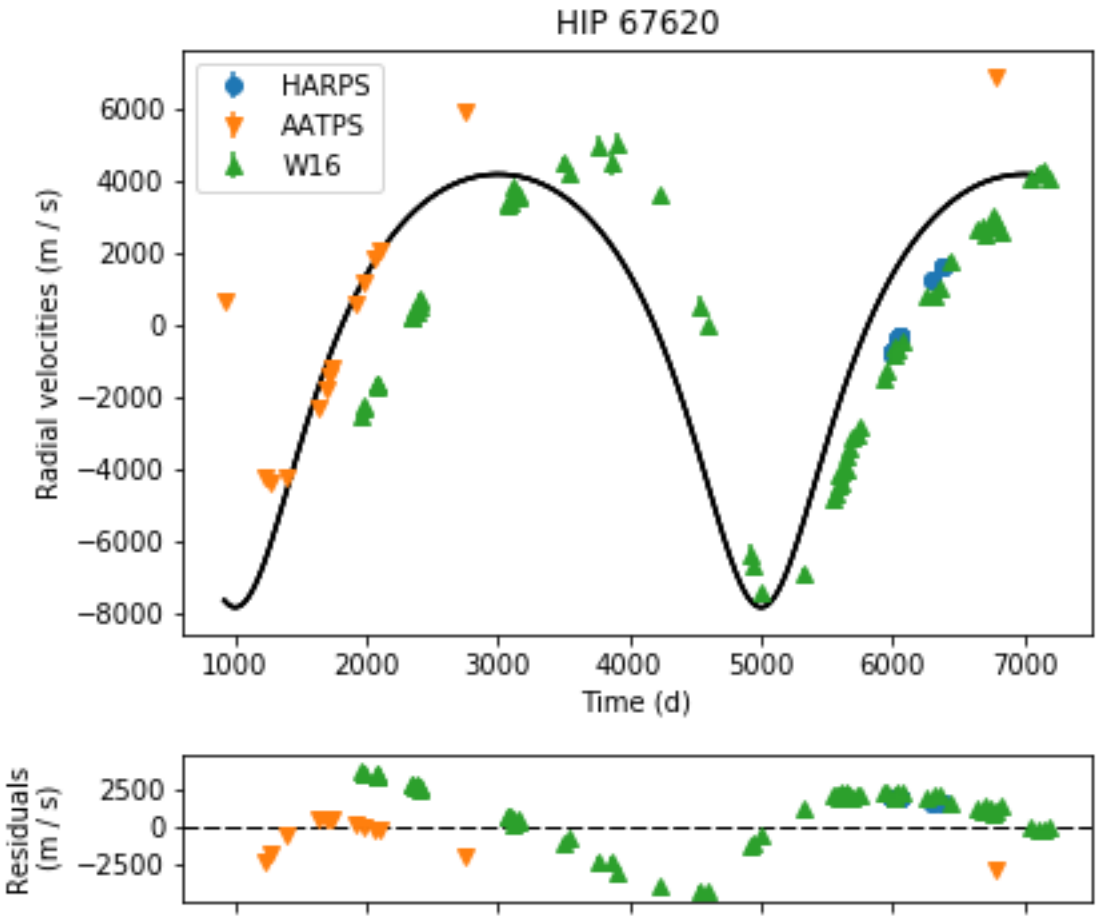
A first guess is usually an educated guess based on either a periodogram and/or simple visual inspection of the data.

```
# guess is a dictionary, which is a special type of "list" in python
# Instead of being indexed by a number, the items in a dictionary
# are indexed by a key (which is a string)
guess = {'k': 6000,
         'period': 4000,
         't0': 5000,
         'omega': 180 * np.pi / 180,
         'ecc': 0.3,
         'gamma_0': 0,
         'gamma_1': 0,
         'gamma_2': 0}
```

Now we need to instantiate a `FullOrbit` object with the datasets and our guess, as well as the parametrization option we want to use. Then, we plot it.

```
estim = estimate.FullOrbit(datasets=[harps, aat, w16],
                           guess=guess,
```

```
                                parametrization='mc10')
plot = estim.plot_rvs(plot_guess=True, fold=False, legend_loc=2)
plt.show()
```



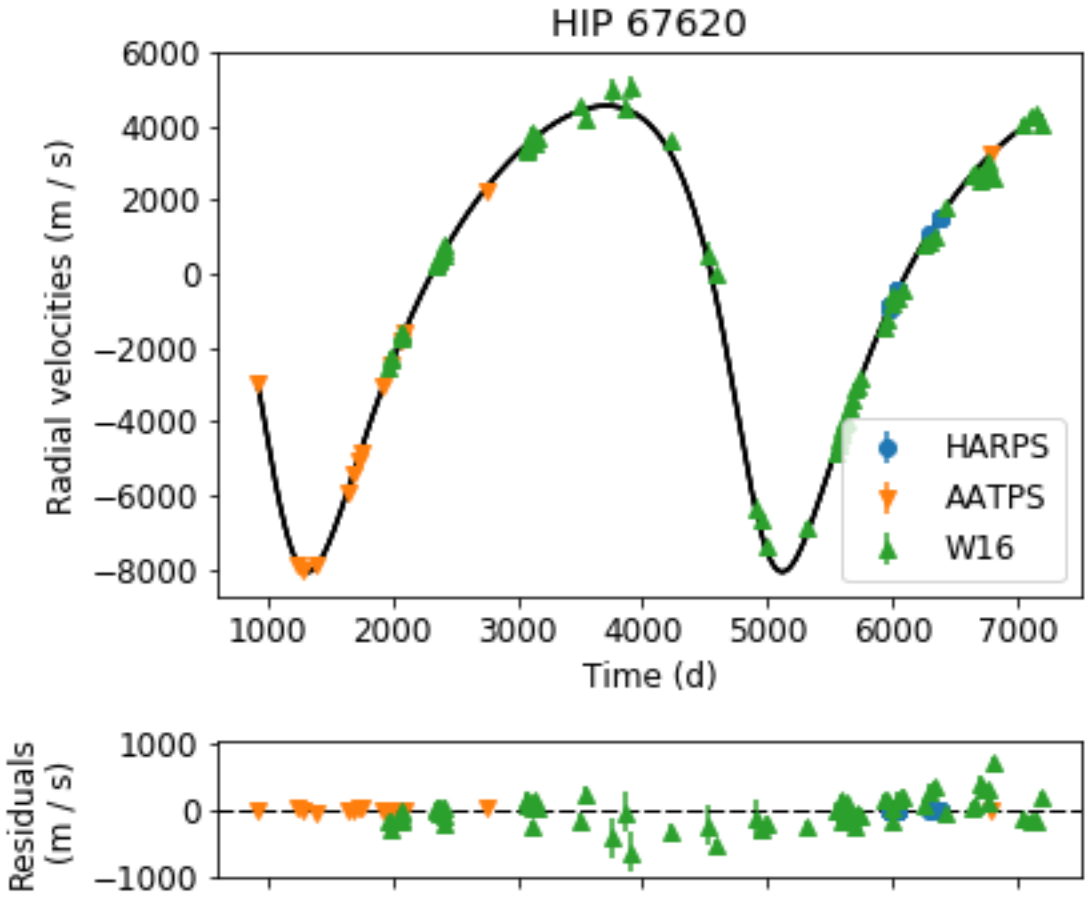
We estimate the orbital parameters of the system using the Nelder-Mead optimization algorithm implemented in the `lmfit` package. This will compute the best solution or, in other words, the one that minimizes the residuals of the fit.

It is probable that the first solutions are not good, and that is fine. Just run the estimation a couple of times until you get the satisfactory result.

```
result = estim.lmfit_orbit(update_guess=True)
```

Now let's plot the solution we obtained.

```
pylab.rcParams['font.size'] = 12
fig, gs = estim.plot_rvs(plot_guess=True, fold=False, legend_loc=4)
```



If the result looks good, that is great: we have the best solution of the orbit. However, we still need to estimate uncertainties for the orbital parameters. We do that using `emcee`. This is a Markov-Chain Monte Carlo (MCMC) simulation, in which we simulate a bunch of sets of orbital parameters that could still fit the data given the uncertainties of the observations, but are a little bit off from the best solution. They will make up the uncertainties of the fit.

This simulation starts from the best solution and do random walks across the parameter space. We will provide the number of *walkers* (`nwalkers`) for the MCMC simulation, as well as the number of *steps* (`nsteps`) that each one will take.

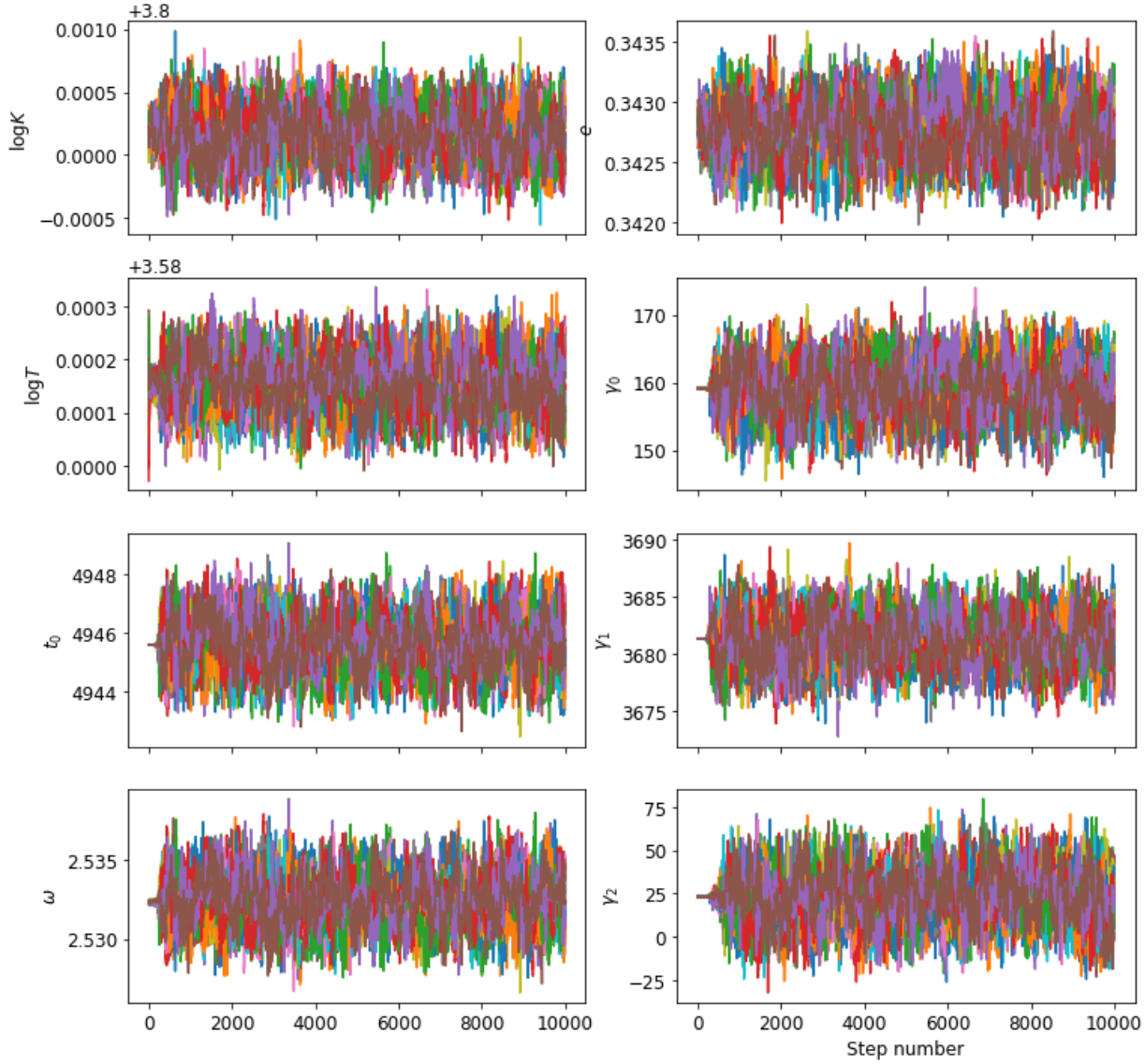
How do we know the number of walkers and steps to use? As a general rule of thumb, it is recommended to use at least 2 times the number of parameters for the number of walkers, and as many steps as it takes for the simulation to converge.

Note: We can use multiprocessing in `emcee` to make the calculations somewhat faster. For that, we need to provide the number of processing threads (in the parameter `nthreads`) of your computer. Most laptops have 2 or 4 threads.

```
estim.emcee_orbit(nwalkers=16, nsteps=10000, nthreads=1)
```

With that done, we plot the walkers to see how the simulation went.

```
estim.plot_emcee_sampler()
```

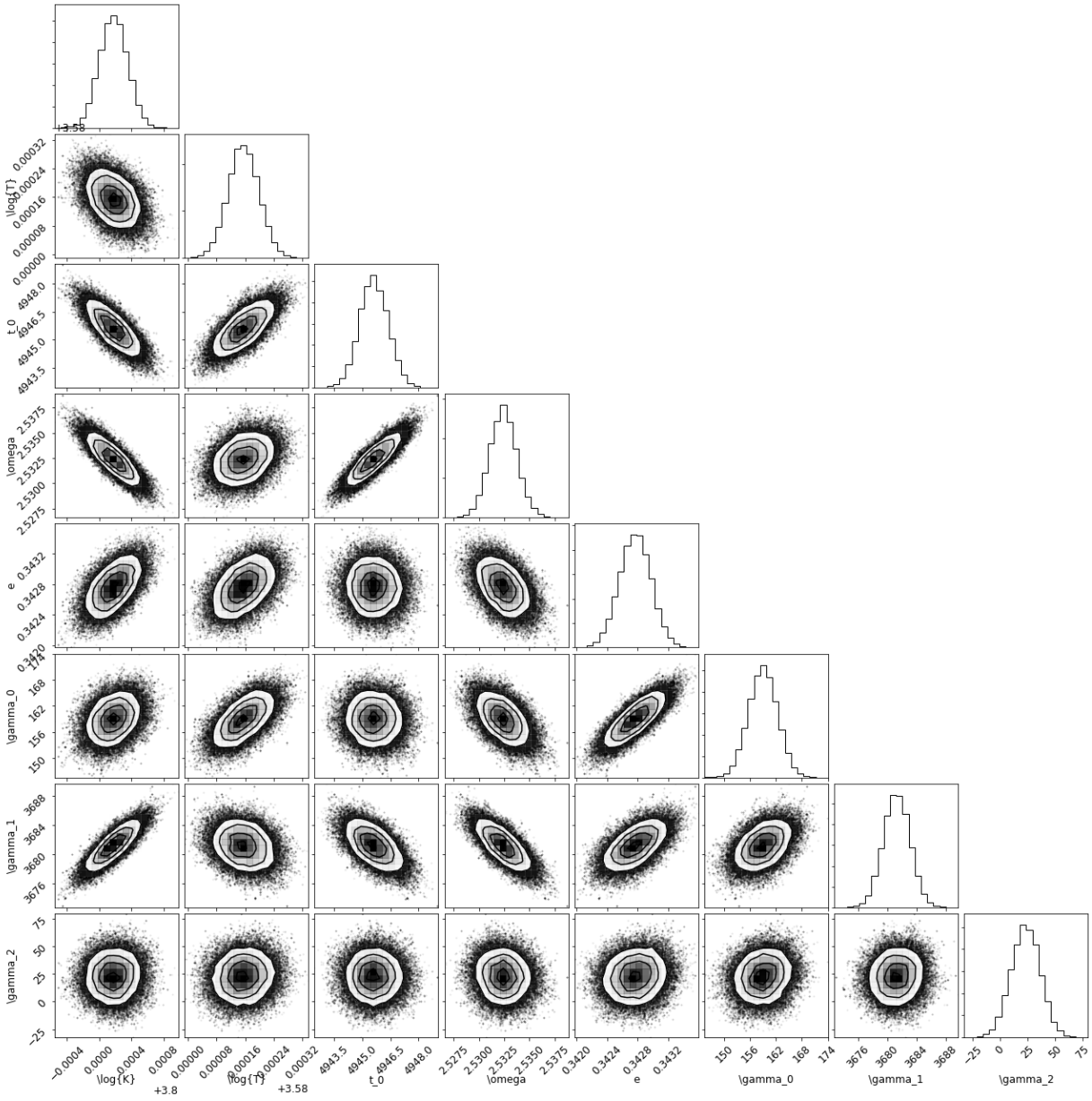


Let's cut the beginning of the simulation (the first 500 steps) because they correspond to the *burn-in* phase.

```
estim.make_chains(500)
```

Now we use a corner plot to analyze the posterior distributions of the parameters, as well as the correlations between them.

```
fig = estim.plot_corner()
plt.show()
```


\log{K}
+3.8
\log{T}
+3.58
t_0
\omega
e
\gamma_0
\gamma_1
\gamma_2

CHAPTER 3

API

radial.dataset module

class `radial.dataset.`**`RVDataSet`**(*file*, *t_col=0*, *rv_col=1*, *rv_unc_col=2*, *skiprows=0*, *delimiter=None*, *t_offset=None*, *rv_offset=None*, *t_unit=None*, *rv_unit=None*, *instrument_name=None*, *target_name=None*, *other_meta=None*)

Bases: `object`

Read and store the data and information of the radial velocities dataset in an intelligible manner. This class utilizes the power and convenience of `astropy` units and tables.

Parameters

- **`file`** (`str`) – Name of the file that contains the radial velocities data.
- **`t_col`** (`int`, optional) – Column number of the data file that corresponds to time. Default is 0.
- **`rv_col`** (`int`, optional) – Column number of the data file that corresponds to radial velocities. Default is 1.
- **`rv_unc_col`** (`int`, optional) – Column number of the data file that corresponds to radial velocity uncertainties. Default is 2.
- **`skiprows`** (`int`, optional) – Number of rows to skip from the data file. Default is 0.
- **`delimiter`** (`str` or `None`, optional) – String that is used to separate the columns in the data file. If `None`, uses the default value from `numpy.loadtxt`. Default is `None`.
- **`t_offset`** (`float`, `astropy.units.Quantity` or `None`, optional) – Numerical offset to be summed to the time array. If `None`, no offset is applied. Default is `None`.
- **`rv_offset`** (`str`, `float`, `astropy.units.Quantity` or `None`,) – optional Numerical offset to be summed to the radial velocities array. If `None`, no offset is applied. `str` options are 'subtract_median' and 'subtract_mean' (self-explanatory). Default is `None`.

- **t_unit** (astropy.units or None, optional) – The unit of the time array, in astropy.units. If None, uses days. Default is None.
- **rv_unit** (astropy.units or None, optional) – The unit of the radial velocities and uncertainties arrays, in astropy.units. If None, uses km/s. Default is None.
- **instrument_name** (str or None, optional) – Name of the instrument, which will be saved in the metadata. Default is None.
- **target_name** (str or None, optional) – Name of the observed target, which will be saved in the metadata. Default is None.
- **other_meta** (dict or None, optional) – Other metadata to be saved in the table. If None, no addition is made. Default is None.

plot()
Plot the data set.

radial.estimate module

class radial.estimate.**FullOrbit**(*datasets*, *guess*, *use_add_sigma=False*, *parametrization=u'mc10'*)
Bases: object

A class that computes the orbital parameters of a binary system given its radial velocities (and their uncertainties) in function of time. This class is optimized for time series that contain at least one full or almost full orbital period.

Parameters

- **datasets** (sequence or radial.dataset.RVDataSet) – A list of RVDataSet objects or one RVDataSet object that contains the data to be fit. If a sequence is passed, the order that the data sets in the sequence will dictate which instrumental parameter (gamma, sigma) index correspond to each data set.
- **guess** (dict) – First guess of the orbital parameters. The keywords must match to the names of the parameters to be fit. These names are: 'k', 'period', 't0', 'omega', 'ecc', 'gamma_X', 'sigma_X' (and so forth), where 'X' is the index of the data set.
- **use_add_sigma** (bool, optional) – If True, the code will use additional parameter to estimate an extra uncertainty term for each RV data set. Default is False.
- **parametrization** (str, optional) – The parametrization for the orbital parameter search. Currently available options are 'mc10' and 'exofast'. Default is 'mc10'.

compute_dynamics(*main_body_mass=1.0*)
Compute the mass and semi-major axis of the companion defined by the orbital parameters.

Parameters **main_body_mass** (float, optional) – The mass of the main body which the companion orbits, in units of solar masses. Default is 1.0.

emcee_orbit(*nwalkers=20*, *nsteps=1000*, *p_scale=2.0*, *nthreads=1*, *ballsizes=None*)
Calculates samples of parameters that best fit the signal rv.

Parameters

- **nwalkers** (int) – Number of walkers
- **nsteps** (int) – Number of burning-in steps
- **p_scale** (float, optional) – The proposal scale parameter. Default is 2.0.

- **nthreads** (int) – Number of threads in your machine
- **ballsizes** (dict) – The one-dimensional size of the volume from which to generate a first position to start the chain.

Returns **sampler** – The resulting sampler object.

Return type emcee.EnsembleSampler

lmfit_orbit(*vary=None*, *verbose=True*, *update_guess=False*, *minimize_mode=u'Nelder'*)
Perform a fit to the radial velocities datasets using lmfit.minimize.

Parameters

- **vary** (dict or None, optional) – Dictionary with keywords corresponding to each parameter, and entries that are True if the parameter is to be left to vary, or False if the parameter is to be fixed in the value provided by the guess. If None, all parameters will vary. Default is None.
- **verbose** (bool, optional) – If True, print output in the screen. Default is False.
- **update_guess** (bool, optional) – If True, updates ~estimate.FullOrbit.guess with the estimated values from the minimization. Default is False.
- **minimize_mode** (str, optional) – The minimization algorithm string. See the documentation of lmfit.minimize for a list of options available. Default is 'Nelder'.

Returns **result** – The resulting MinimizerResult object.

Return type lmfit.MinimizerResult

lnlike(*theta*)
Log-likelihood of a given set of parameters to adequately describe the observed data.

Parameters **theta** (dict or lmfit.Parameters) – The orbital parameters.

Returns **sum_res** – The log-likelihood.

Return type scalar

lnprob(*theta_list*)
This function calculates the ln of the probabilities to be used in the MCMC estimation.

Parameters **theta_list** (*sequence*) –

Returns

- *The probability of the signal rv being the result of a model with the*
- *parameters theta*

lomb_scargle(*dset_index*, *freqs*)
Compute a Lomb-Scargle periodogram for a given data set using scipy.signal.lombscargle.

Parameters

- **dset_index** (int) – Index of the data set to have the periodogram calculated for.
- **freqs** (array_like) – Angular frequencies for output periodogram.

Returns

- **pgram** (array_like) – Lomb-Scargle periodogram.
- **fig** (matplotlib.pyplot.figure) – Periodogram plot.

make_chains(*ncut*, *outfile=None*)
Make a chains object that represent the posterior distributions of the orbital parameters.

Parameters

- **ncut** (int) – Number of points of the burn-in phase to be ignored.
- **outfile** (str or None) – A string containing the path to the file where the chains will be saved. This is useful when you do not want to keep running emcee frequently. If None, no output file is produced. Default is None.

Returns **emcee_chains** – The chains of the emcee run, with the burn-in phase removed.

Return type numpy.ndarray

plot_corner()
Produce a corner (a.k.a. triangle) plot of the posterior distributions of the orbital parameters estimated with emcee.

Returns

Return type fig

plot_emcee_sampler(*outfile=None*, *n_cols=2*, *fig_size=(12, 12)*)
Plot the emcee sampler so that the user can check for convergence.

Parameters

- **outfile** (str or None, optional) – Name of the output image file to be saved. If None, then no output file is produced, and the plot is displayed on screen. Default is None.
- **n_cols** (int, optional) – Number of columns of the plot. Default is 2.
- **fig_size** (*tuple, optional*) – Sizes of each panel of the plot, where the first element of the tuple corresponds to the x-direction size, and the second element corresponds to the y-direction size. Default is (12, 12).

plot_rvs(*legend_loc=None*, *symbols=None*, *plot_guess=False*, *plot_samples=False*, *fold=False*, *numpoints=1000*)
Plot the data sets.

Parameters

- **legend_loc** (int or None, optional) – Location of the legend. If None, use the default from matplotlib. Default is None.
- **symbols** (sequence or None, optional) – List of symbols for each data set in the plot. If None, use the default list from matplotlib markers. Default is None.
- **plot_guess** (bool, optional) – If True, also plots the guess as a black curve, and the RVs of each data set is shifted by its respective gamma value.
- **plot_samples** (bool, optional) – If True, also plots the samples obtained with the emcee estimation. Default is False.
- **fold** (bool, optional) – If True, plot the radial velocities by folding them around the estimated orbital period. Default is False.
- **numpoints** (int, optional) – Number of points to compute the radial velocities curve. Default is 1000.

prepare_params(*theta*, *vary_param=None*)
Prepare a lmfit.Parameters object to be used in the lmfit estimation.

Parameters

- **theta** (dict) – The current orbital parameters.

- **vary_param** (dict) – Dictionary that says which parameters will vary in the estimation. By default, all parameters vary. A parameter can be fixed by setting its key to False.

Returns **params** – The lmfit.Parameters object for the estimation.

Return type lmfit.Parameters

print_emcee_result(*main_star_mass=None*, *mass_sigma=None*)

radial.body module

class radial.body.**Companion**(*k=None*, *period_orb=None*, *t_0=None*, *omega=None*, *ecc=None*, *msini=None*, *semi_a=None*, *name=None*, *main_star=None*, *mass=None*, *sini=None*)

Bases: object

The massive companion class. It can be either a binary star, an exoplanet, maybe even a black hole! It can be anything that has a mass and orbits another massive object. General relativity effects are not implemented yet.

Parameters

- **k** (astropy.units.Quantity or None, optional) – The radial velocity semi-amplitude. Default is None.
- **period_orb** (astropy.units.Quantity or None, optional) – The orbital period. Default is None.
- **t_0** (astropy.units.Quantity or None, optional) – The time of periastron passage. Default is None.
- **omega** (astropy.units.Quantity or None, optional) – Argument of periapse. Default is None.
- **ecc** (float or None, optional) – Eccentricity of the orbit. Default is None.
- **msini** (astropy.units.Quantity or None, optional) – Mass of the companion multiplied by sine of the inclination of the orbital plane in relation to the line of sight. Default is None.
- **semi_a** (astropy.units.Quantity or None, optional) – Semi-major axis of the orbit. Default is None.
- **name** (str or None, optional) – Name of the companion. Default is None.

class radial.body.**MainStar**(*mass*, *name=None*)

Bases: object

The main star of a given system.

mass [astropy.units.Quantity] The mass of the star.

name [str or None] The name of the star. Default is None.

class radial.body.**System**(*main_star*, *companion*, *time=None*, *name=None*, *dataset=None*)

Bases: object

The star-companions system class.

Parameters

- **main_star** (radial.object.MainStar) – The main star of the system.

- **companion** (*list*) – Python list containing the all the `radial.object.Companion` of the system.
- **time** (`astropy.units.Quantity` or `None`) – A scalar or `numpy.ndarray` containing the times in which the radial velocities are measured. Default is `None`.
- **name** (`str` or `None`, optional) – Name of the system. Default is `None`.
- **dataset** (sequence, `radial.dataset.RVDataSet` or `None`, optional) – A list of `RVDataSet` objects or one `RVDataSet` object that contains the data to be fit. Default is `None`.

compute_rv()
Compute the radial velocities of the main star, both the individual RVs (corresponding to each companion) and the total RVs.

mass_func()
Compute the mass functions of all the companions of the system. This method will also compute the msini and semi_a of the companions and save the values in their respective parameters.

plot_rv(*companion_index=None*, *plot_title=None*)

Parameters **companion_index** (`int` or `None`) – The companion index indicates which set of radial velocities will be plotted. If `None`, then the total radial velocities are plotted. Default is `None`.

Returns

- *fig*
- *ax*

radial.orbit module

class `radial.orbit.`**BinarySystem**(*k*, *period*, *t0*, *omega=None*, *ecc=None*, *sqe_cosw=None*, *sqe_sinw=None*, *gamma=0*)

Bases: `object`

A class that computes the radial velocities given the orbital parameters of the binary system.

Parameters

- **k** (*scalar*) – The radial velocity semi-amplitude K in m / s.
- **period** (*scalar*) – The orbital period in days.
- **t0** (*scalar*) – Time of pariastron passage in days.
- **omega** (scalar or `None`, optional) – Argument of periapse in radians. If `None`, both `sqe_cosw` and `sqe_sinw` will be required. Default is `None`.
- **ecc** (scalar or `None`, optional) – Eccentricity of the orbit. If `None`, both `sqe_cosw` and `sqe_sinw` will be required. Default is `None`.
- **sqe_cosw** (scalar or `None`, optional) – The square root of the eccentricity multiplied by the cosine of the argument of periapse. If `None`, both `omega` and `ecc` will be required. Default is `None`.
- **sqe_sinw** (scalar or `None`, optional) – The square root of the eccentricity multiplied by the sine of the argument of periapse. If `None`, both `omega` and `ecc` will be required. Default is `None`.

- **gamma** (scalar or None, optional) – Proper motion of the barycenter in m / s. Default is 0.

get_rvs(*ts*)
Computes the radial velocity given the orbital parameters.

Parameters **ts** (scalar or numpy.ndarray) – Time in days.

Returns **rvs** – Radial velocities

Return type scalar or numpy.ndarray

kep_eq(*e_ano*, *m_ano*)
The Kepler equation.

Parameters

- **e_ano** (*scalar*) – Eccentric anomaly in radians.
- **m_ano** (*scalar*) – Mean anomaly in radians.

Returns **kep** – Value of E-e*sin(E)-M

Return type scalar

rv_eq(*f*)
The radial velocities equation.

Parameters **f** (scalar or numpy.ndarray) – True anomaly in radians.

Returns **rvs** – Radial velocity

Return type scalar or numpy.ndarray

radial.prior module

radial.prior.**flat**(*theta*, *parametrization*)
Computes a flat prior probability for a given set of parameters theta.

Parameters **theta** (dict) – The orbital and instrumental parameters. This dictionary must have keywords identical to the parameter names (which depend on the number of data sets, the parametrization option and if the additional uncertainties are also being estimated.

Returns **prob** – The prior probability for a given set of orbital and instrumental parameters.

Return type float

radial.rv_model module

radial.rv_model.**exofast**(*t*, *log_k*, *log_period*, *t0*, *sqe_cosw*, *sqe_sinw*, *gamma*)
The radial velocities model from EXOFAST (Eastman et al. 2013).

Parameters

- **t** (*scalar*) – Time in days.
- **log_k** (*scalar*) – Base-10 logarithm of the radial velocity semi-amplitude in dex(m / s).
- **log_period** (*scalar*) – Base-10 logarithm of the orbital period in dex(d).
- **t0** (*scalar*) – Time of pariastron passage in days.
- **sqe_cosw** (*scalar*) – sqrt(ecc) * cos(omega).

- **sqe_sinw** (*scalar*) – sqrt(ecc) * sin(omega).
- **gamma** (*scalar*) – Instrumental radial velocity offset in m / s.

Returns **rvs** – Radial velocity in m / s.

Return type scalar

radial.rv_model.**mc10**(*t*, *log_k*, *log_period*, *t0*, *omega*, *ecc*, *gamma*)

The radial velocities model from Murray & Correia 2010.

Parameters

- **t** (*scalar*) – Time in days.
- **log_k** (*scalar*) – Base-10 logarithm of the radial velocity semi-amplitude in dex(m / s).
- **log_period** (*scalar*) – Base-10 logarithm of the orbital period in dex(d).
- **t0** (*scalar*) – Time of pariastron passage in days.
- **omega** (*scalar*) – Argument of periapse in radians.
- **ecc** (*scalar*) – Base-10 logarithm of the eccentricity of the orbit.
- **gamma** (*scalar*) – Instrumental radial velocity offset in m / s.

Returns **rvs** – Radial velocity in m / s.

Return type scalar

Python Module Index

r

Index