radvel Documentation

Release 1.4.11

BJ Fulton, Erik Petigura, Sarah Blunt, and Evan Sinukoff

Oct 16, 2023

CONTENTS

Welcome to the documentation for `radvel`, a Python package for modeling of radial velocity time series data.

Please cite Fulton, Petigura, Blunt & Sinukoff (2018) and the following DOI if you make use of RadVel in your research.

Bug reports and feature requests should be posted to the GitHub issue tracker .

Code contributions are welcome and should be submitted as a pull request. Read the paper for further details regarding contributions.

Contents:

CHAPTER

ONE

GETTING STARTED

1.1 Installation

While it is possible to install `radvel` inside a minimal Python environment like that built-in to Mac OSX, we recommend first installing a scientific Python environment such as Anaconda or Miniconda.

Install `radvel` using pip:

```
$ pip install radvel
```

Make sure that `pdflatex` is installed and in your system's path. You can get `pdflatex` by installing the TexLive package or other LaTeX distributions. By default it is expected to be in your system's path, but you may specify a path to pdflatex using the `--latex-compiler` option at the `radvel report` step.

If you are running OSX, and want to perform Gaussian Process likelihood computations in parallel, you may need to perform some additional installation steps. See OSX-multiprocessing.

If you wish to use the celerite GP kernels you will also need to install celerite. See the celerite install instructions.

1.2 Example Fit

Test your installation by running through one of the included examples. We will use the `radvel` command line interface to execute a multi-planet, multi-instrument fit.

The `radvel` binary should have been automatically placed in your system's path by the `pip` command (see *Installation*). If your system can not find the `radvel` executable then try running `python setup.py install` from within the top-level `radvel` directory.

First lets look at `radvel --help` for the available options:

```
$ radvel --help
usage: RadVel [-h] [--version] {fit,plot,mcmc,derive,bic,table,report} ...

RadVel: The Radial Velocity Toolkit

optional arguments:
  -h, --help            show this help message and exit
  --version             Print version number and exit.

subcommands:
  {fit,plot,mcmc,derive,bic,table,report}
```

Here is an example workflow to run a simple fit using the included *HD164922.py* example configuration file. This example configuration file can be found in the `example_planets` subdirectory on the GitHub repository page.

Perform a maximum-likelihood fit. You almost always will need to do this first:

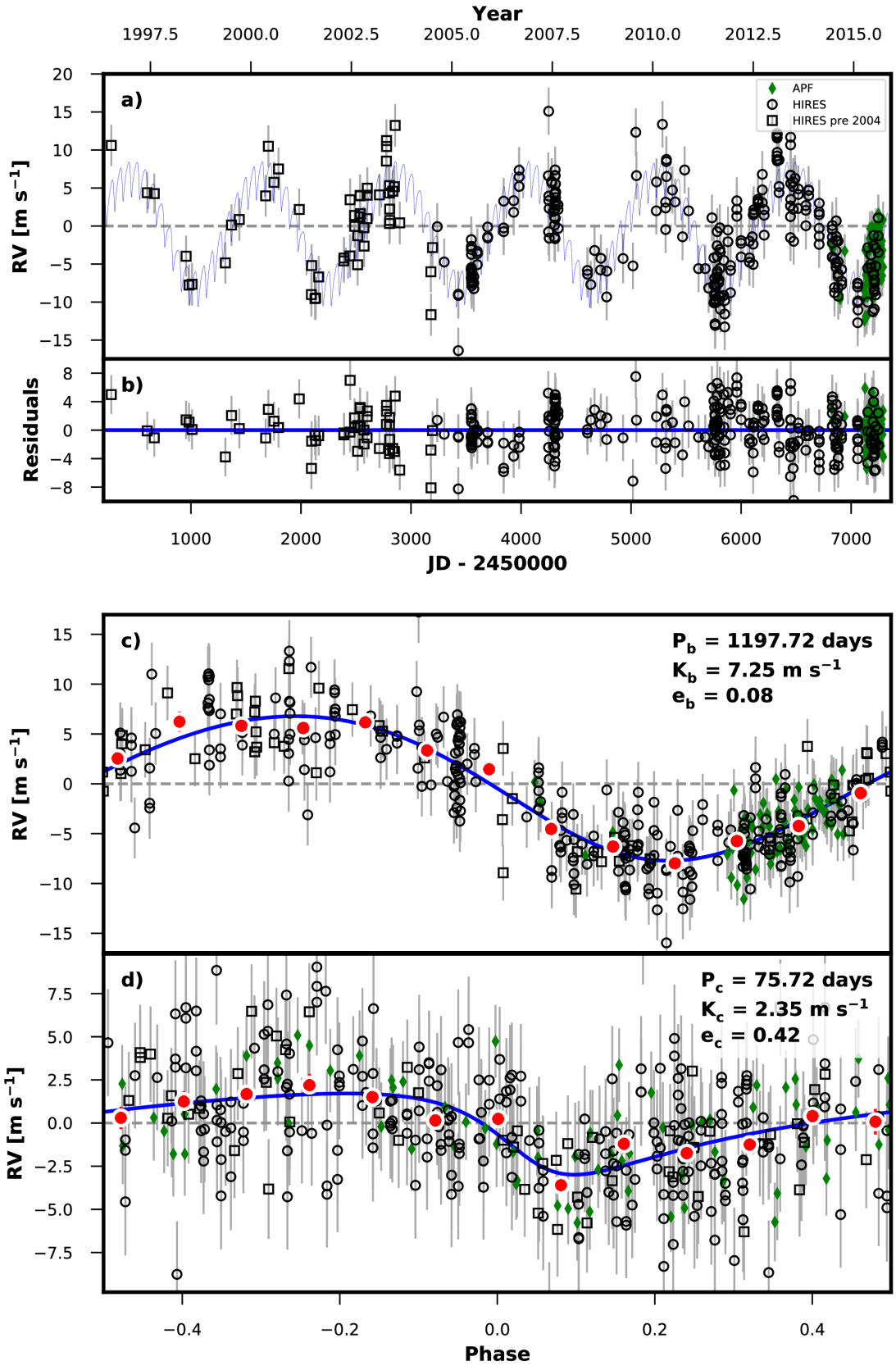
```
$ radvel fit -s /path/to/radvel/example_planets/HD164922.py
```

By default the results will be placed in a directory with the same name as your planet configuration file (without *.py*, e.g. *HD164922*). You may also specify an output directory using the `-o` flag.

After the maximum-likelihood fit is complete the directory should have been created and should contain one new file: *HD164922/HD164922_post_obj.pkl*. This is a `pickle` binary file that is not meant to be human-readable but lets make a plot of the best-fit solution contained in that file:

```
$ radvel plot -t rv -s /path/to/radvel/example_planets/HD164922.py
```

This should produce a plot named *HD164922_rv_multipanel.pdf* that looks something like this.


Year
1997.5
2000.0
2002.5
2005.0
2007.5
2010.0
2012.5
2015.0
a)
APF
HIRES
HIRES pre 2004
RV [m s^{-1}]
b)
Residuals
JD - 2450000
c)
P_b = 1197.72 days
K_b = 7.25 m s^{-1}
e_b = 0.08
d)
P_c = 75.72 days
K_c = 2.35 m s^{-1}
e_c = 0.42
Phase

Next lets perform the Markov-Chain Monte Carlo (MCMC) exploration to assess parameter uncertainties.

```
$ radvel mcmc -s /path/to/radvel/example_planets/HD164922.py
```

Once the MCMC chains finish running there will be another new file called *HD164922_mcmc_chains.csv.tar.bz2*. This is a compressed csv file containing the parameter values and likelihood at each step in the MCMC chains.

Now we can update the RV time series plot with the MCMC results and generate the full suite of plots.

```
$ radvel plot -t rv corner trend -s /path/to/radvel/example_planets/HD164922.py
```

We can summarize our analysis with the *radvel report* command.

```
$ radvel report -s /path/to/radvel/example_planets/HD164922.py
```

which creates a LaTeX document and corresponding PDF to summarize the results. Note that this command assembles values and plots that have been computed through other commands, if you want to update, rerun the previous commands before reruning *radvel report*

The report PDF will be saved as *HD164922_results.pdf*. It should contain a table reporting the parameter values and uncertainties, a table summarizing the priors, the RV time-series plot, and a corner plot showing the posterior distributions for all free parameters.

1.3 Optional Features

Combine the measured properties of the RV time-series with the properties of the host star defined in the setup file to derive physical parameters for the planetary system. Have a look at the *epic203771098.py* example setup file to see how to include stellar parameters.

```
$ radvel derive -s /path/to/radvel/example_planets/HD164922.py
```

Generate a corner plot for the derived parameters. This plot will also be included in the summary report if available.

```
$ radvel plot -t derived -s /path/to/radvel/example_planets/HD164922.py
```

Perform a model comparison testing models eliminating different sets of planets, their eccentricities, and RV trends. If this is run a new table will be included in the summary report.

```
$ radvel ic -t nplanets e trend -s /path/to/radvel/example_planets/HD164922.py
```

Generate and save only the TeX code for any/all of the tables.

```
$ radvel table -t params priors ic_compare derived -s /path/to/radvel/example_planets/
↪HD164922.py
```

CHAPTER

TWO

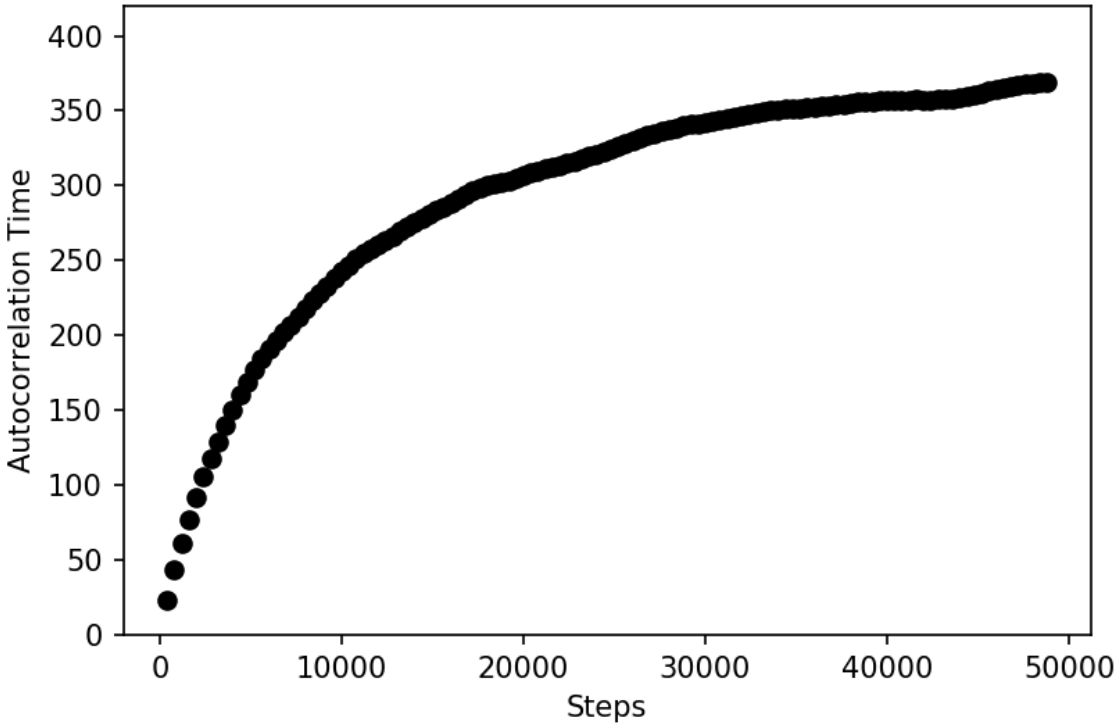
INTRODUCTION TO AUTOCORRELATION TIMES

2.1 Background

As of v1.3.0 there are two additional new convergence criterion in addition to *maxGR* and *minTz* used to evaluate whether or not the MCMC chains are converged/well-mixed. *MinAfactor* and *maxArchange* both rely on the autocorrelation time. While a more in-depth explanation of autocorrelation and its relation to convergence can be found in the documentation for emcee, we will provide a brief description here. The original RadVel convergence checks are also still in place and are described in Fulton et al. (2018).

Autocorrelation time is the number of steps the walker needs to take for the chain to "forget" its initial position. Therefore, it can be used to reduce the error in your MCMC run and to tell if it is well-mixed. Using `emcee3`, we can estimate this value for each parameter in an ensemble; if multiple ensembles are run, we combine their chains to receive the autocorrelation times.

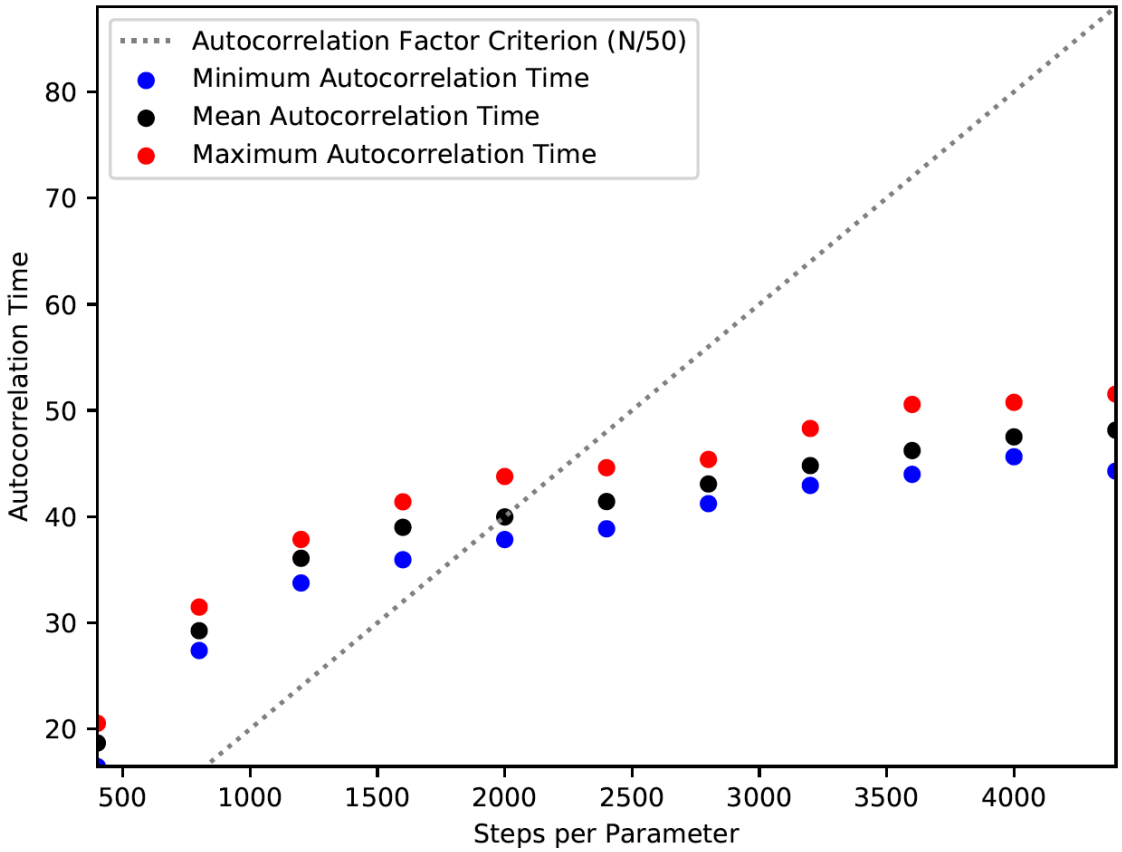
As the chain progresses, the estimated value of the autocorrelation time () becomes more accurate. In the plot below, you can see that the the estimated autocorrelation time quickly rises and then begins to plateau as it nears the true value.



2.2 minAfactor

The autocorrelation time can be used to evaluate whether or not a chain is sufficiently long to be considered well-mixed. Therefore, if N steps have been taken, N/ c, where c is what we call the autocorrelation factor. The minimum autocorrelation factor to consider the chains converged is represented by the criterion `minAfactor`. The default value for `minAfactor` is 40, however we find that higher values may be useful, particularly for shorter runs; alternate values can be specified using the `minAfactor` argument when calling the MCMC.

The autocorrelation factor is calculated for each parameter and the minimum of these values is returned in real time as the MCMC run progresses. Once the minimum autocorrelation factor is above `minAfactor`, this criterion for convergence is met. Whether or not `minAfactor` has been satisfied can be seen in the autocorrelation plot below. Once the maximum autocorrelation time has passed the dashed line labeled 'Autocorrelation Factor Criterion,' the chain is likely converged. After five consecutive status checks appear past the dashed line, the MCMC will halt if all other criterion have also been met.



2.3 maxArchange

While we want the autocorrelation factor to be sufficiently large, we want to make sure that it is being calculated with an accurate estimate of the autocorrelation time. We know our estimate for is accurate once it begins to plateau, allowing us to use the relative change in the autocorrelation time to infer whether or not the estimate is reliable. We calculate the relative change in autocorrelation time for each parameter between every convergence check. The largest of these values is also returned in real time and for the chain to be considered converged, it must fall below the criterion `maxArchange`.

The default value for `maxArchange` is .03, where we consider the autocorrelation time to be leveling off. Rarely should you need to increase the `maxArchange` argument when running *radvel mcmc*, but for more conservative criterion, you may want to decrease it, particularly for long chains with large autocorrelation times (in such cases, the relative change may be smaller, but has not reached its plateau).

CHAPTER

THREE

ADVANCED USAGE

These tutorials give some examples in the use of the underlying `radvel` API. They are also available as interactive iPython notebooks in the *docs/tutorials* subdirectory of the radvel package.

3.1 K2-24 Fitting & MCMC

Using the K2-24 (EPIC-203771098) dataset, we demonstrate how to use the radvel API to:

- perform a max-likelihood fit
- do an MCMC exploration of the posterior space
- plot the results

3.1.1 Circular Orbits

Perform some preliminary imports:

[ ]:
```
%matplotlib inline

import os

import matplotlib
import numpy as np
import pylab as pl
import pandas as pd
from scipy import optimize

import corner

import radvel
import radvel.likelihood
from radvel.plot import orbit_plots, mcmc_plots

matplotlib.rcParams['font.size'] = 14
```

Define a function that we will use to initialize the `radvel.Parameters` and `radvel.RVModel` objects

[2]:
```
def initialize_model():
    time_base = 2420
    params = radvel.Parameters(2,basis='per tc secosw sesinw logk') # number of planets␣
```

(continues on next page)

(continued from previous page)

```
↪= 2
    params['per1'] = radvel.Parameter(value=20.885258)
    params['tc1'] = radvel.Parameter(value=2072.79438)
    params['secosw1'] = radvel.Parameter(value=0.01)
    params['sesinw1'] = radvel.Parameter(value=0.01)
    params['logk1'] = radvel.Parameter(value=1.1)
    params['per2'] = radvel.Parameter(value=42.363011)
    params['tc2'] = radvel.Parameter(value=2082.62516)
    params['secosw2'] = radvel.Parameter(value=0.01)
    params['sesinw2'] = radvel.Parameter(value=0.01)
    params['logk2'] = radvel.Parameter(value=1.1)
    mod = radvel.RVModel(params, time_base=time_base)
    mod.params['dvdt'] = radvel.Parameter(value=-0.02)
    mod.params['curv'] = radvel.Parameter(value=0.01)
    return mod
```

Define a simple plotting function to display the data, model, and residuals

[3]:
```
def plot_results(like):
    fig = pl.figure(figsize=(12,4))
    fig = pl.gcf()
    fig.set_tight_layout(True)
    pl.errorbar(
        like.x, like.model(t)+like.residuals(),
        yerr=like.yerr, fmt='o'
        )
    pl.plot(ti, like.model(ti))
    pl.xlabel('Time')
    pl.ylabel('RV')
    pl.draw()
```

Load up the K2-24 data. In this example the RV data and parameter starting guesses are stored in an csv file

[4]:
```
path = os.path.join(radvel.DATADIR,'epic203771098.csv')
rv = pd.read_csv(path)

t = np.array(rv.t)
vel = np.array(rv.vel)
errvel = rv.errvel
ti = np.linspace(rv.t.iloc[0]-5,rv.t.iloc[-1]+5,100)
```

Fit the K2-24 RV data assuming:

1. circular orbits
2. fixed period, time of transit

Set initial guesses for the parameters. Setting vary=False and linear=True on the gamma parameters will cause them to be solved for analytically following the technique described here (Thanks Tim Brandt!). If you use this you will need to calculate the uncertainties on gammas manually following that derivation.

[5]:
```
mod = initialize_model()
like = radvel.likelihood.RVLikelihood(mod, t, vel, errvel)
```

(continues on next page)

(continued from previous page)

```
like.params['gamma'] = radvel.Parameter(value=0.1, vary=False, linear=True)
like.params['jit'] = radvel.Parameter(value=1.0)
```

Choose which parameters to vary or fix. By default, all `radvel.Parameter` objects will vary, so you only have to worry about setting the ones you want to hold fixed.

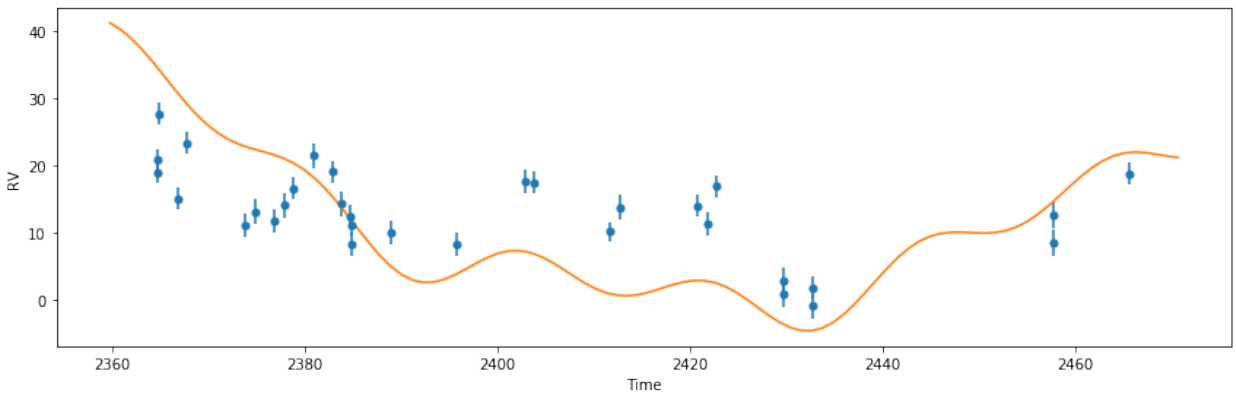
[6]:
```
like.params['secosw1'].vary = False
like.params['sesinw1'].vary = False
like.params['secosw2'].vary = False
like.params['sesinw2'].vary = False
like.params['per1'].vary = False
like.params['per2'].vary = False
like.params['tc1'].vary = False
like.params['tc2'].vary = False
print(like)
```

```
parameter                      value      vary
per1                         20.8853     False
tc1                          2072.79     False
secosw1                         0.01     False
sesinw1                         0.01     False
logk1                            1.1      True
per2                          42.363     False
tc2                          2082.63     False
secosw2                         0.01     False
sesinw2                         0.01     False
logk2                            1.1      True
dvdt                            -0.02      True
curv                            0.01      True
gamma                            0.1     False
jit                                1      True
```

Plot the initial model

[7]:
```
pl.figure()
plot_results(like)
```

```
<Figure size 432x288 with 0 Axes>
```



Well that solution doesn't look very good. Now lets try to optimize the parameters set to vary by maximizing the

likelihood.

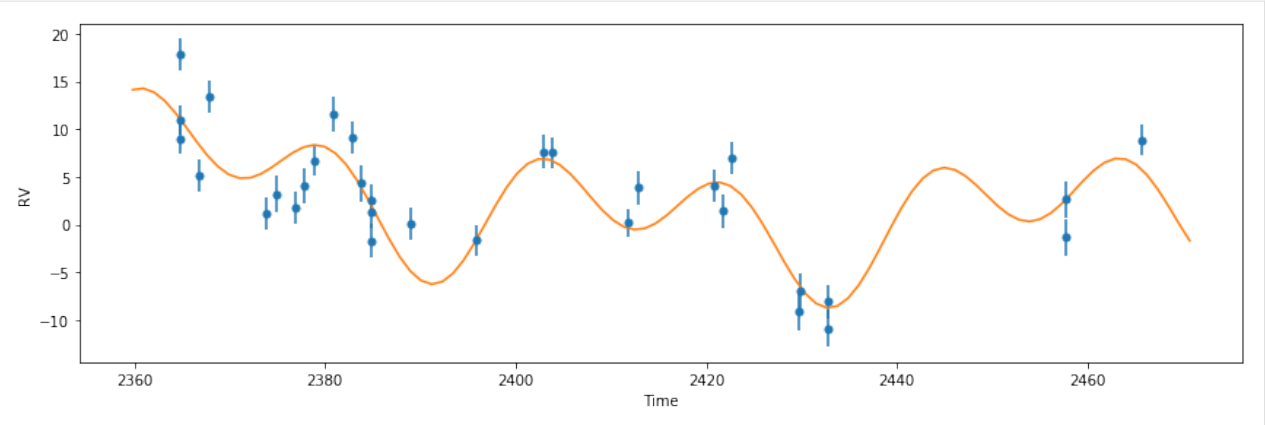
Initialize a `radvel.Posterior` object and add some priors

[8]:
```
post = radvel.posterior.Posterior(like)
post.priors += [radvel.prior.Gaussian( 'jit', np.log(3), 0.5)]
post.priors += [radvel.prior.Gaussian( 'logk2', np.log(5), 10)]
post.priors += [radvel.prior.Gaussian( 'logk1', np.log(5), 10)]
post.priors += [radvel.prior.Gaussian( 'gamma', 0, 10)]
```

Maximize the likelihood and print the updated posterior object

[9]:
```
res  = optimize.minimize(
    post.neglogprob_array,     # objective function is negative log likelihood
    post.get_vary_params(),    # initial variable parameters
    method='Nelder-Mead',           # Powell also works
    )

plot_results(like)             # plot best fit model
print(post)
```

```
parameter                      value      vary
per1                         20.8853     False
tc1                          2072.79     False
secosw1                         0.01     False
sesinw1                         0.01     False
logk1                        1.55037      True
per2                          42.363     False
tc2                          2082.63     False
secosw2                         0.01     False
sesinw2                         0.01     False
logk2                        1.37648      True
dvdt                      -0.0292189      True
curv                      0.00182259      True
gamma                       -3.99195     False
jit                          2.09753      True

Priors
------
Gaussian prior on jit, mu=1.0986122886681098, sigma=0.5
Gaussian prior on logk2, mu=1.6094379124341003, sigma=10
Gaussian prior on logk1, mu=1.6094379124341003, sigma=10
Gaussian prior on gamma, mu=0, sigma=10
```



That looks much better!

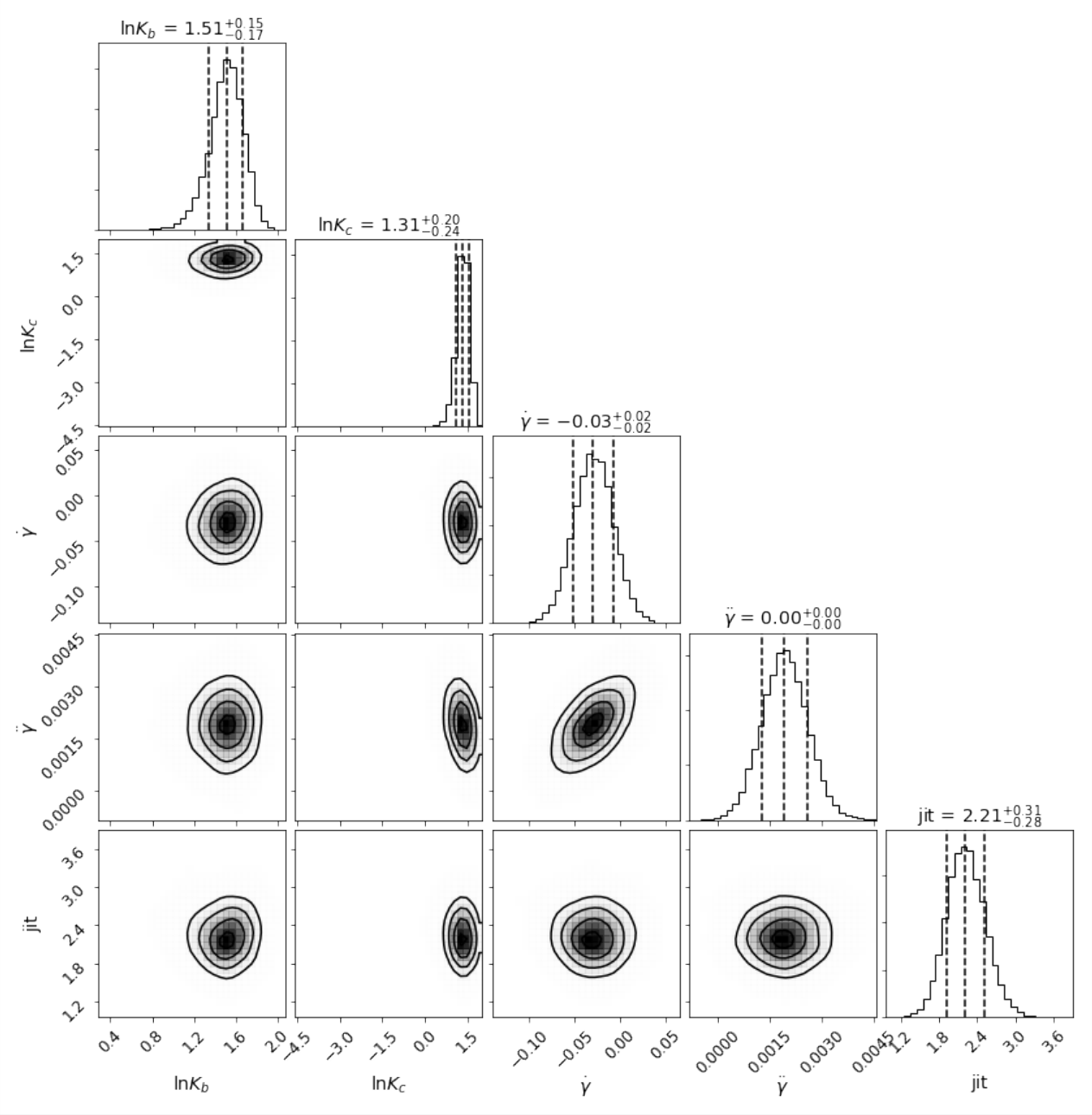
Now lets use Markov-Chain Monte Carlo (MCMC) to estimate the parameter uncertainties. In this example we will run 400 steps for the sake of speed but in practice you should let it run at least 10000 steps and ~50 walkers. If the chains converge before they reach the maximum number of allowed steps it will automatically stop.

[10]:
```
df = radvel.mcmc(post,nwalkers=20,nrun=400,savename='rawchains.h5')
```

```
8000/64000 (12.5%) steps complete; Running 12210.22 steps/s; Mean acceptance rate = 57.3
↪%; Min Auto Factor =  22; Max Auto Relative-Change =   inf; Min Tz = 1176.3; Max G-R =␣
↪1.024
Discarding burn-in now that the chains are marginally well-mixed

64000/64000 (100.0%) steps complete; Running 13032.47 steps/s; Mean acceptance rate = 55.
↪1%; Min Auto Factor =  50; Max Auto Relative-Change = 0.169; Min Tz = 4270.9; Max G-R␣
↪= 1.007

MCMC: WARNING: chains did not pass convergence tests. They are likely not well-mixed.
```

Now lets make a corner plot to display the posterior distributions.

[11]:
```
Corner = mcmc_plots.CornerPlot(post, df)
Corner.plot()
```



3.1.2 Eccentric Orbits

Allow `secosw` and `sesinw` parameters to vary

```
[12]: like.params['secosw1'].vary = True
      like.params['sesinw1'].vary = True
      like.params['secosw2'].vary = True
      like.params['sesinw2'].vary = True
```

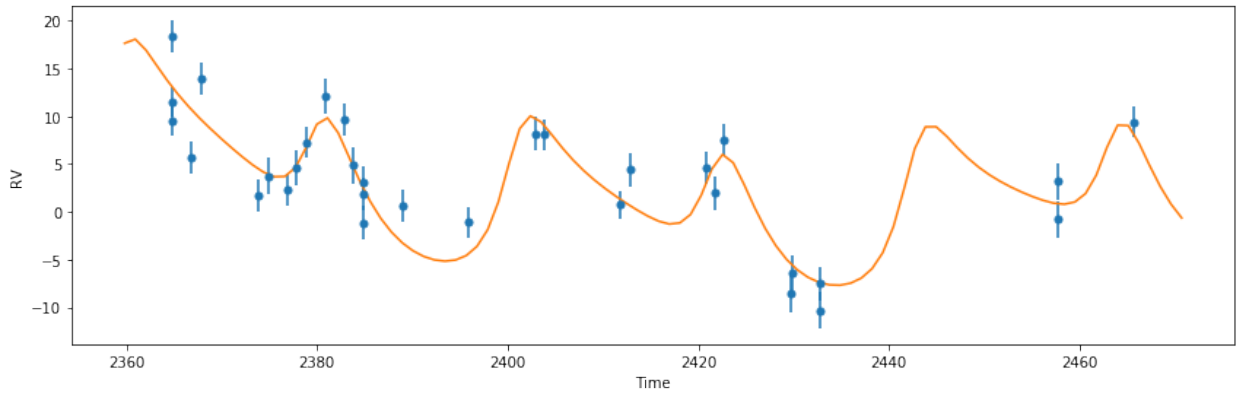
Add an `EccentricityPrior` to ensure that eccentricity stays below 1.0. In this example we will also add a Gaussian prior on the jitter (`jit`) parameter with a center at 2.0 m/s and a width of 0.1 m/s.

[13]:
```
post = radvel.posterior.Posterior(like)
post.priors += [radvel.prior.EccentricityPrior( 2 )]
post.priors += [radvel.prior.Gaussian( 'jit', np.log(2), np.log(0.1))]
```

Optimize the parameters by maximizing the likelihood and plot the result

[14]:
```
res  = optimize.minimize(
    post.neglogprob_array,
    post.get_vary_params(),
    method='Powell',)

plot_results(like)
print(post)
```

```
parameter                      value      vary
per1                         20.8853     False
tc1                          2072.79     False
secosw1                     0.398041      True
sesinw1                    -0.404209      True
logk1                        1.8115      True
per2                          42.363     False
tc2                          2082.63     False
secosw2                    -0.124394      True
sesinw2                      0.37035      True
logk2                        1.48072      True
dvdt                      -0.0297857      True
curv                     0.00206207      True
gamma                      -4.52471     False
jit                          1.9542      True

Priors
------
e1 constrained to be < 0.99
e2 constrained to be < 0.99
Gaussian prior on jit, mu=0.6931471805599453, sigma=-2.3025850929940455
```



Plot the final solution

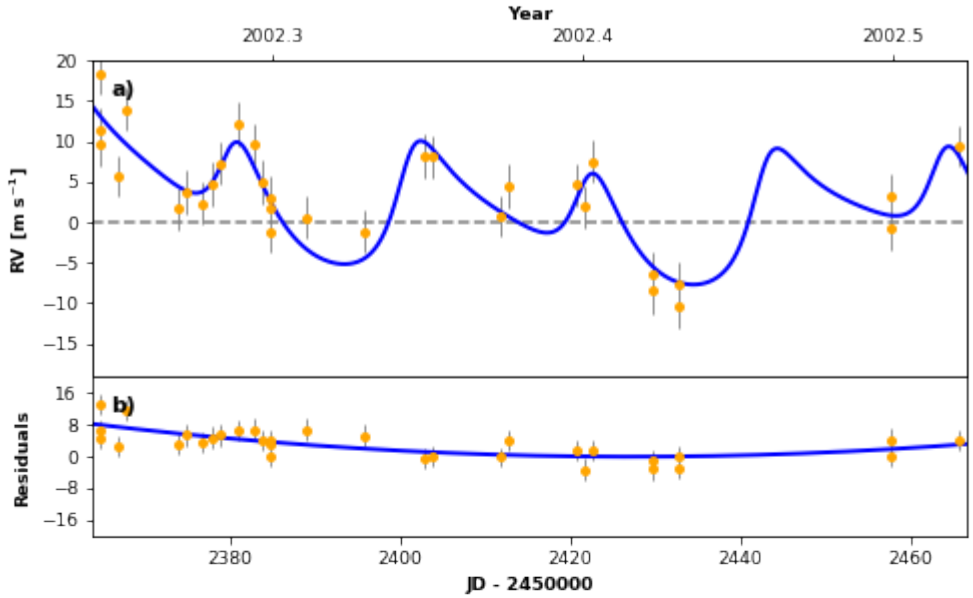
[15]:
```
RVPlot = orbit_plots.MultipanelPlot(post, legend=False)
```

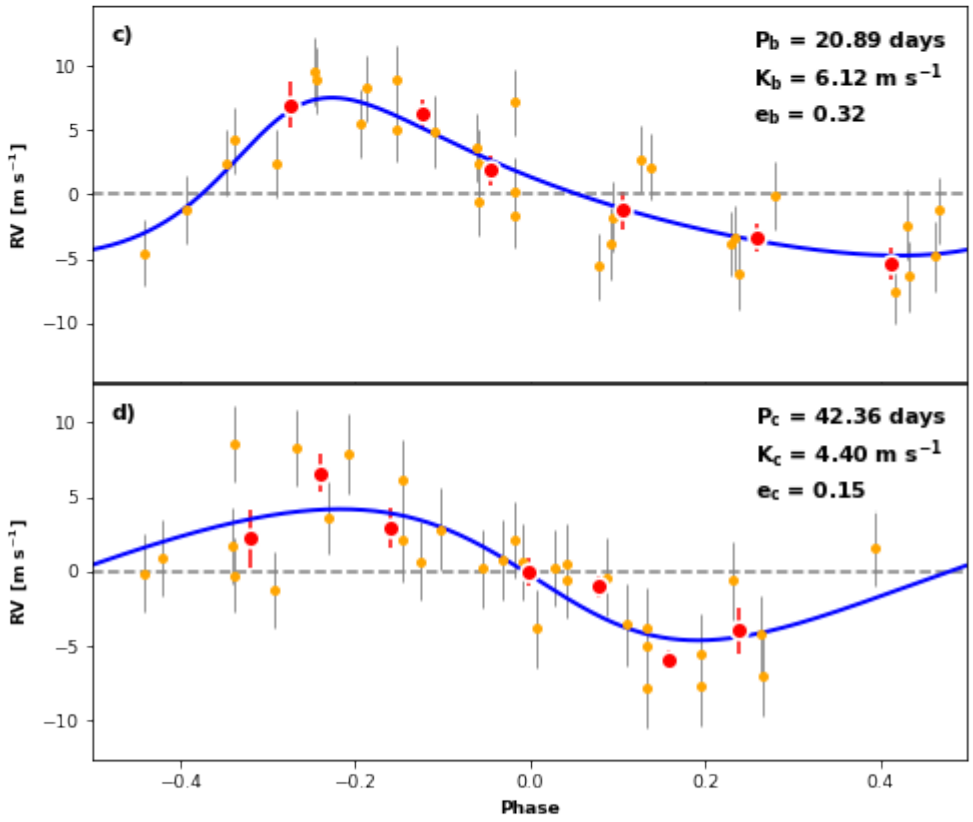
(continues on next page)

(continued from previous page)

```
RVPlot.plot_multipanel()
```

[15]:
```
(<Figure size 540x786.857 with 5 Axes>,
 [<matplotlib.axes._subplots.AxesSubplot at 0x7fc70affe908>,
  <matplotlib.axes._subplots.AxesSubplot at 0x7fc6d8861f28>,
  <matplotlib.axes._subplots.AxesSubplot at 0x7fc72a3f1f28>,
  <matplotlib.axes._subplots.AxesSubplot at 0x7fc70b03ea58>])
```


Year
2002.3
2002.4
2002.5
a)
RV [m s^{-1}]
b)
Residuals
JD - 2450000


c)
P_b = 20.89 days
K_b = 6.12 m s^{-1}
e_b = 0.32
d)
P_c = 42.36 days
K_c = 4.40 m s^{-1}
e_c = 0.15
RV [m s^{-1}]
Phase

3.2 Gaussian Process Fitting

by Sarah Blunt

3.2.1 Prerequisites

This tutorial assumes knowledge of the basic `radvel` API for χ^2 likelihood fitting. As such, please complete the following before beginning this tutorial: - radvel/docs/tutorials/164922_Fitting+MCMC.ipynb - radvel/docs/tutorials/K2-24_Fitting+MCMC.ipynb

This tutorial also assumes knowledge of Gaussian Processes (GPs) as applied to radial velocity (RV) timeseries modeling. Grunblatt et al. (2015) and Rajpaul et al. (2015) contain excellent introductions to this topic. Also check out "Gaussian Processes for Machine Learning," by Rasmussen & Williams, a free online textbook hosted at gaussianprocesses.org.

3.2.2 Objectives

Using the K2-131 (EPIC-228732031) dataset published in Dai et al. (2017), I will show how to: - perform a maximum a posteriori (MAP) fit using a quasi-periodic kernel GP regression to model stellar activity (with data from multiple telescopes) - do an MCMC exploration of the corresponding parameter space (with data from multiple telescopes)

3.2.3 Tutorial

Do some preliminary imports:

[1]:
```
import numpy as np
import pandas as pd
import os
import radvel
import radvel.likelihood
from radvel.plot import orbit_plots, mcmc_plots
from scipy import optimize

%matplotlib inline
```

Read in RV data from Dai et al. (2017):

[2]:
```
data = pd.read_csv(os.path.join(radvel.DATADIR,'k2-131.txt'), sep=' ')

t = np.array(data.time)
vel = np.array(data.mnvel)
errvel = np.array(data.errvel)
tel = np.array(data.tel)

telgrps = data.groupby('tel').groups
instnames = telgrps.keys()
```

We'll use a quasi-periodic covariance kernel in this fit. An element in the covariance matrix, C_{ij} is defined as follows:

$$C_{ij} = \eta_1^2 exp[-\frac{|t_i - t_j|^2}{\eta_2^2} - \frac{sin^2(\pi|t_i - t_j|/\eta_3)}{2\eta_4^2}]$$

Several other kernels are implemented in `radvel`. The code for all kernels lives in radvel/gp.py. Check out that file if you'd like to implement a new kernel.

Side Note: to see a list of all implemented kernels and examples of possible names for their associated hyperparameters…

[3]:
```
print(radvel.gp.KERNELS)
```

```
{'SqExp': ['gp_length', 'gp_amp'], 'Per': ['gp_per', 'gp_length', 'gp_amp'], 'QuasiPer':␣
↪['gp_per', 'gp_perlength', 'gp_explength', 'gp_amp'], 'Celerite': ['gp_B', 'gp_C', 'gp_
↪L', 'gp_Prot']}
```

Define the GP hyperparameters we will use in our fit:

[4]:
```
hnames = [
    'gp_amp', # eta_1; GP variability amplitude
    'gp_explength', # eta_2; GP non-periodic characteristic length
    'gp_per', # eta_3; GP variability period
    'gp_perlength', # eta_4; GP periodic characteristic length
]
```

Define some numbers (derived from photometry) that we will use in our priors on the GP hyperparameters:

[5]:
```
gp_explength_mean = 9.5*np.sqrt(2.) # sqrt(2)*tau in Dai+ 2017 [days]
gp_explength_unc = 1.0*np.sqrt(2.)
gp_perlength_mean = np.sqrt(1./(2.*3.32)) # sqrt(1/(2*gamma)) in Dai+ 2017
gp_perlength_unc = 0.019
gp_per_mean = 9.64 # T_bar in Dai+ 2017 [days]
gp_per_unc = 0.12

Porb = 0.3693038 # orbital period [days]
Porb_unc = 0.0000091
Tc = 2457582.9360 # [BJD]
Tc_unc = 0.0011
```

Dai et al. (2017) derive the above from photometry (see sect 7.2.1). I'm currently working on implementing joint modeling of RVs & photometry and RVs & activity indicators in `radvel`, so stay tuned if you'd like to use those features!

Initialize `radvel.Parameters` object:

[6]:
```
nplanets=1
params = radvel.Parameters(nplanets,basis='per tc secosw sesinw k')
```

Set initial guesses for each fitting parameter:

[7]:
```
params['per1'] = radvel.Parameter(value=Porb)
params['tc1'] = radvel.Parameter(value=Tc)
params['sesinw1'] = radvel.Parameter(value=0.,vary=False) # fix eccentricity = 0
params['secosw1'] = radvel.Parameter(value=0.,vary=False)
params['k1'] = radvel.Parameter(value=6.55)
params['dvdt'] = radvel.Parameter(value=0.,vary=False)
params['curv'] = radvel.Parameter(value=0.,vary=False)
```

Set initial guesses for GP hyperparameters:

[8]:
```
params['gp_amp'] = radvel.Parameter(value=25.0)
params['gp_explength'] = radvel.Parameter(value=gp_explength_mean)
params['gp_per'] = radvel.Parameter(value=gp_per_mean)
params['gp_perlength'] = radvel.Parameter(value=gp_perlength_mean)
```

Instantiate a `radvel.model.RVmodel` object, with `radvel.Parameters` object as attribute:

[9]:
```
gpmodel = radvel.model.RVModel(params)
```

Initialize `radvel.likelihood.GPLikelihood` objects (one for each telescope):

[10]:
```
jit_guesses = {'harps-n':0.5, 'pfs':5.0}

likes = []
def initialize(tel_suffix):

    # Instantiate a separate likelihood object for each instrument.
    # Each likelihood must use the same radvel.RVModel object.
    indices = telgrps[tel_suffix]
    like = radvel.likelihood.GPLikelihood(gpmodel, t[indices], vel[indices],
                                          errvel[indices], hnames, suffix='_'+tel_suffix,
                                          kernel_name="QuasiPer"
                                          )
    # Add in instrument parameters
    like.params['gamma_'+tel_suffix] = radvel.Parameter(value=np.mean(vel[indices]),␣
↪vary=False, linear=True)
    like.params['jit_'+tel_suffix] = radvel.Parameter(value=jit_guesses[tel_suffix],␣
↪vary=True)
    likes.append(like)


for tel in instnames:
    initialize(tel)
```

Instantiate a `radvel.likelihood.CompositeLikelihood` object that has both GP likelihoods as attributes:

[11]:
```
gplike = radvel.likelihood.CompositeLikelihood(likes)
```

Instantiate a `radvel.Posterior` object:

[12]:
```
gppost = radvel.posterior.Posterior(gplike)
```

Add in priors (see Dai et al. 2017 section 7.2):

[13]:
```
gppost.priors += [radvel.prior.Gaussian('per1', Porb, Porb_unc)]
gppost.priors += [radvel.prior.Gaussian('tc1', Tc, Tc_unc)]
gppost.priors += [radvel.prior.Jeffreys('k1', 0.01, 10.)] # min and max for Jeffrey's␣
↪priors estimated by Sarah
gppost.priors += [radvel.prior.Jeffreys('gp_amp', 0.01, 100.)]
gppost.priors += [radvel.prior.Jeffreys('jit_pfs', 0.01, 10.)]
gppost.priors += [radvel.prior.Jeffreys('jit_harps-n', 0.01,10.)]
gppost.priors += [radvel.prior.Gaussian('gp_explength', gp_explength_mean, gp_explength_
↪unc)]
gppost.priors += [radvel.prior.Gaussian('gp_per', gp_per_mean, gp_per_unc)]
```

(continues on next page)

(continued from previous page)

```
gppost.priors += [radvel.prior.Gaussian('gp_perlength', gp_perlength_mean, gp_perlength_
↪unc)]
```

Note: our prior on `'gp_perlength'` isn't equivalent to the one Dai et al. (2017) use because our formulations of the quasi-periodic kernel are slightly different. The results aren't really affected.

Do a MAP fit:

[14]:
```
res = optimize.minimize(
    gppost.neglogprob_array, gppost.get_vary_params(), method='Nelder-Mead',
    options=dict(maxiter=200, maxfev=100000, xatol=1e-8)
)

print(gppost)
```

```
parameter                      value      vary
per1                        0.369302       True
tc1                      2.45758e+06       True
secosw1                            0      False
sesinw1                            0      False
k1                           6.71302       True
dvdt                               0      False
curv                               0      False
gp_amp                       23.3362       True
gp_explength                 13.3721       True
gp_per                       9.62823       True
gp_perlength                0.381962       True
gamma_harps-n               -6695.13      False
jit_harps-n                      0.5      False
gamma_pfs                    1.66129      False
jit_pfs                            5      False

Priors
------
Gaussian prior on per1, mu=0.3693038, sigma=9.1e-06
Gaussian prior on tc1, mu=2457582.936, sigma=0.0011
Jeffrey's prior on k1, min=0.01, max=10.0
Jeffrey's prior on gp_amp, min=0.01, max=100.0
Jeffrey's prior on jit_pfs, min=0.01, max=10.0
Jeffrey's prior on jit_harps-n, min=0.01, max=10.0
Gaussian prior on gp_explength, mu=13.435028842544403, sigma=1.4142135623730951
Gaussian prior on gp_per, mu=9.64, sigma=0.12
Gaussian prior on gp_perlength, mu=0.38807526285316646, sigma=0.019
```

Explore the parameter space with MCMC:

[15]:
```
chains = radvel.mcmc(gppost,nrun=100,ensembles=3,savename='rawchains.h5')
```

```
15000/15000 (100.0%) steps complete; Running 1231.70 steps/s; Mean acceptance rate = 45.7
↪%; Min Auto Factor =  12; Max Auto Relative-Change = 0.978; Min Tz = 248.2; Max G-R =␣
↪1.012
Discarding burn-in now that the chains are marginally well-mixed
```

(continues on next page)

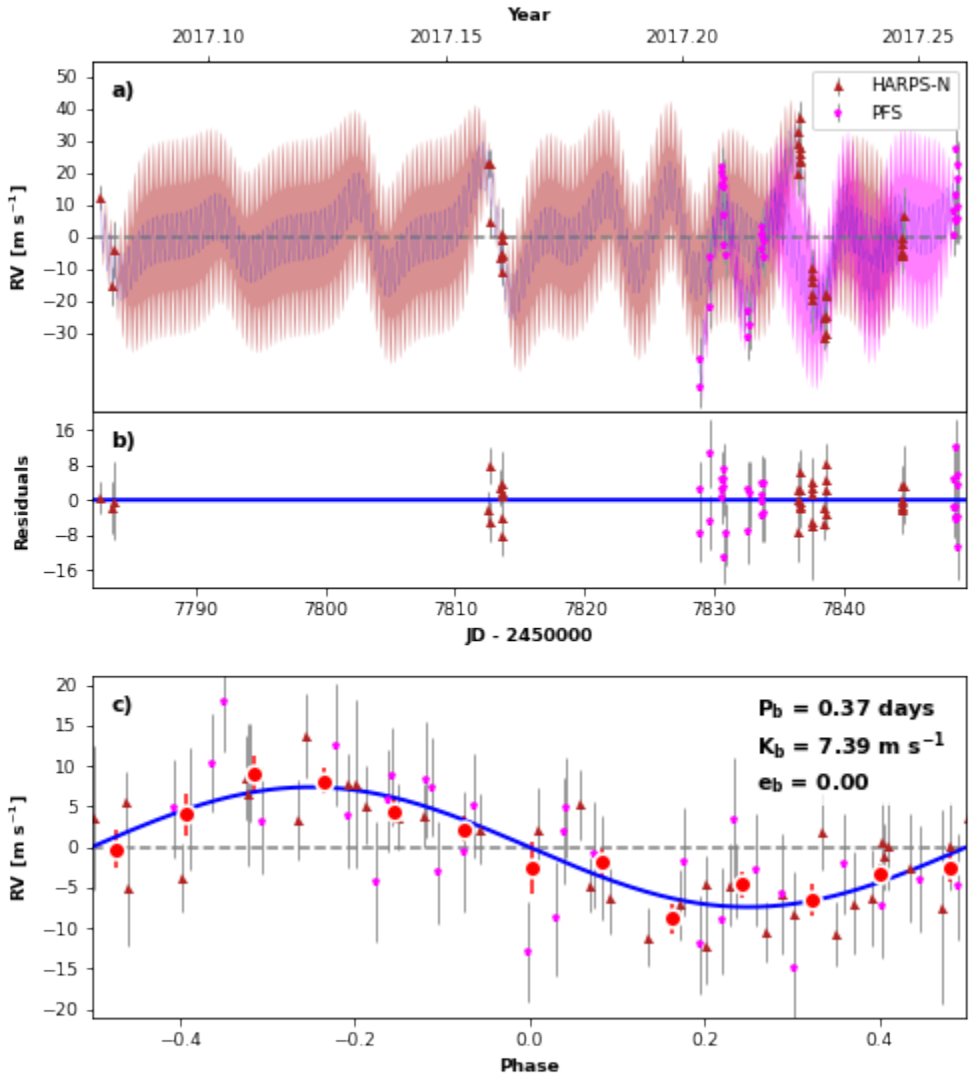
(continued from previous page)

```
MCMC: WARNING: chains did not pass convergence tests. They are likely not well-mixed.
```

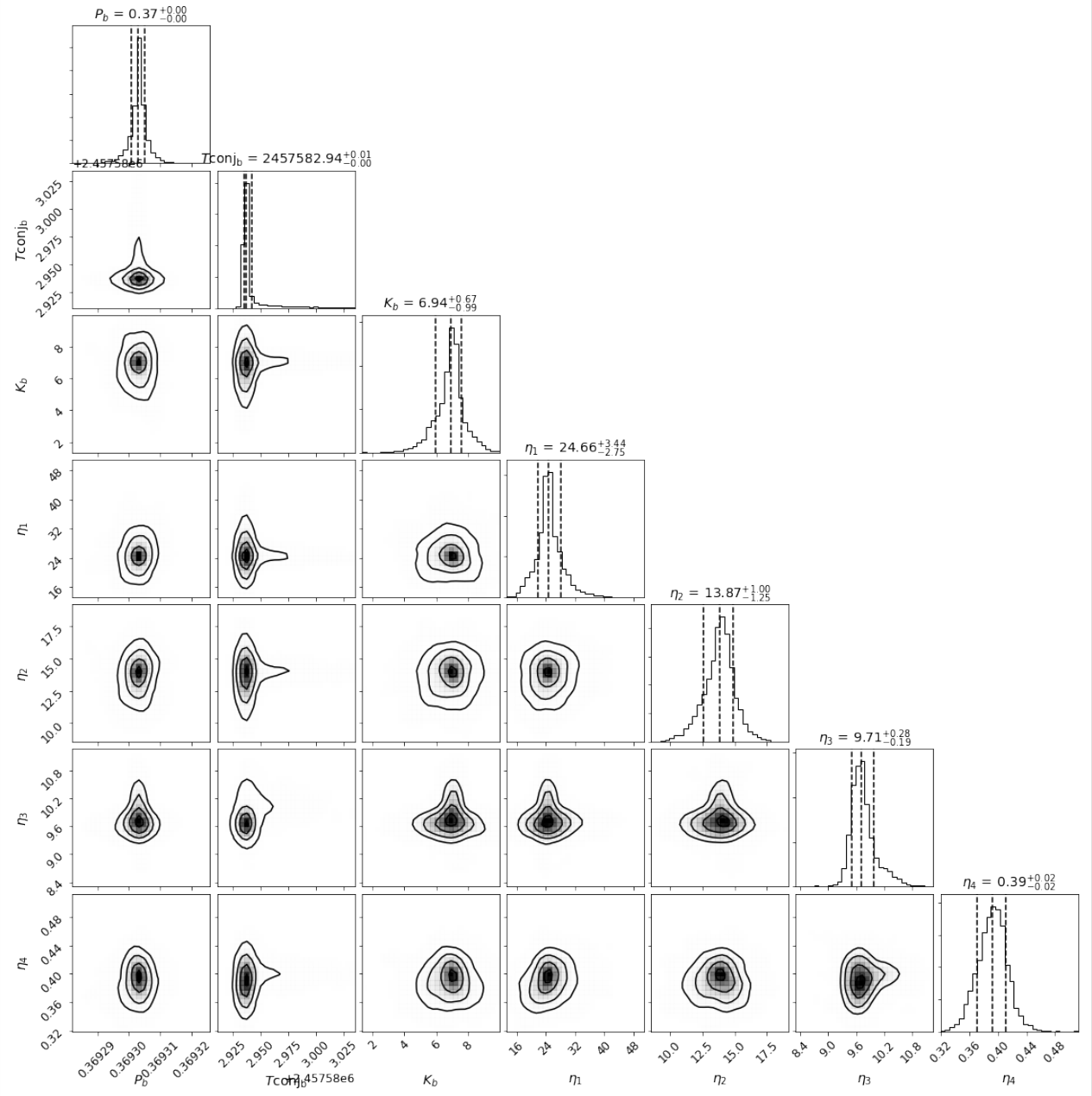
Note: for reliable results, run MCMC until the chains have converged. For this example, nrun=10000 should do the trick, but that would take a minute or two, and I won't presume to take up that much of your time with this tutorial.

Make some nice plots:

[16]:
```
# try switching some of these (optional) keywords to "True" to see what they do!
GPPlot = orbit_plots.GPMultipanelPlot(
    gppost,
    subtract_gp_mean_model=False,
    plot_likelihoods_separately=False,
    subtract_orbit_model=False
)
GPPlot.plot_multipanel()
```



[17]:
```
Corner = mcmc_plots.CornerPlot(gppost, chains) # posterior distributions
Corner.plot()
```



[18]:
```
quants = chains.quantile([0.159, 0.5, 0.841]) # median & 1sigma limits of posterior␣
↪distributions

for par in gppost.params.keys():
    if gppost.params[par].vary:
        med = quants[par][0.5]
        high = quants[par][0.841] - med
        low = med - quants[par][0.159]
        err = np.mean([high,low])
```

(continues on next page)

(continued from previous page)

```
        err = radvel.utils.round_sig(err)
        med, err, errhigh = radvel.utils.sigfig(med, err)
        print('{} : {} +/- {}'.format(par, med, err))
```

```
per1 : 0.3693031 +/- 2.1e-06
tc1 : 2457582.9366 +/- 0.0036
k1 : 6.94 +/- 0.83
gp_amp : 24.7 +/- 3.1
gp_explength : 13.9 +/- 1.1
gp_per : 9.71 +/- 0.24
gp_perlength : 0.39 +/- 0.02
```

Compare posterior characteristics with those of Dai et al. (2017):

per1 : 0.3693038 +/- 9.1e-06

tc1 : 2457582.936 +/- 0.0011

k1 : 6.6 +/- 1.5

gp_amp : 26.0 +/- 6.2

gp_explength : 11.6 +/- 2.3

gp_per : 9.68 +/- 0.15

gp_perlength : 0.35 +/- 0.02

gamma_harps-n : -6695 +/- 11

jit_harps-n : 2.0 +/- 1.5

gamma_pfs : -1 +/- 11

jit_pfs : 5.3 +/- 1.4

Thanks for going through this tutorial! As always, if you have any questions, feature requests, or problems, please file an issue on the `radvel` GitHub repo (github.com/California-Planet-Search/radvel).

3.3 Custom Models and Likelihoods

By fitting a basic lightcurve model with a radial velocity model, we demonstrate how to build custom models and likelihoods in RadVel versions 1.40 and later. Note that this is different from previous versions, now that parameters are stored in a `radvel.Vector` object.

Perform some preliminary imports:

[42]:
```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy import optimize

import corner

import radvel
import radvel.likelihood
```

(continues on next page)

(continued from previous page)

```
import batman

matplotlib.rcParams['font.size'] = 14
```

We begin by generating artifical transit data using the `batman <https://www.cfa.harvard.edu/~lkreidberg/batman/index.html>`__ package, described in Kreidberg (2015):

[43]:
```
x_trans = np.sort(np.random.uniform(low=2456296,high=2456307,size=170))
yerr_trans = 4e-3 #assuming we know the Gaussian uncertainty

p = batman.TransitParams()
p.t0 = 2456301.6
p.per = 200.31
p.rp = .121
p.a = 14.3
p.inc = 89.0
p.ecc = 0.0
p.w = 0
p.limb_dark = 'uniform'
p.u = []

m = batman.TransitModel(p, x_trans)
y_trans = m.light_curve(p)
y_trans += yerr_trans * np.random.randn(len(y_trans))
```

We can now generate corresponding simulated radial velocities using `radvel`:

[44]:
```
x_rv = np.sort(np.random.uniform(low=2456200,high=2457140,size=83))
yerr_rv = 10 #assuming we know the Gaussian uncertainty

synth_params = radvel.Parameters(1,basis='per tc e w k')
synth_params['per1'] = radvel.Parameter(value = 200.31)
synth_params['tc1'] = radvel.Parameter(value = 2456301.6)
synth_params['e1'] = radvel.Parameter(value = 0.0)
synth_params['w1'] = radvel.Parameter(value = 0.0)
synth_params['k1'] = radvel.Parameter(value = 39.1)

synth_params['dvdt'] = radvel.Parameter(value=0)
synth_params['curv'] = radvel.Parameter(value=0)

synth_model = radvel.RVModel(params=synth_params)
y_rv = synth_model(x_rv)
y_rv += yerr_rv * np.random.randn(len(y_rv))
```

Now we prepare for the analyis of our data. We can begin by defining the parameters that will be used in our models:

[45]:
```
params = radvel.Parameters(num_planets=1)
params['tc'] = radvel.Parameter(value=2456300)
params['per'] = radvel.Parameter(value=200)
params['a'] = radvel.Parameter(value=10)
params['rp'] = radvel.Parameter(value=0.08)
params['inc'] = radvel.Parameter(value=90)
```

(continues on next page)

(continued from previous page)

```
params['e'] = radvel.Parameter(value=0.0, vary=False, linear=False) #for simplicity, we␣
↪assume a circular orbit
params['w'] = radvel.Parameter(value=0.0, vary=False, linear=False)
params['k'] = radvel.Parameter(value=30)

params['jit_trans'] = radvel.Parameter(value=0.01)
params['gamma_trans'] = radvel.Parameter(value=0, vary=False) #Unless you construct your␣
↪own likelihood,
                                                              #you must provide both a␣
↪gamma and jitter term.
                                                              #Here we fix gamma at 0␣
↪because it doesn't contribute
                                                              #to the transit likelihood
params['jit_rv'] = radvel.Parameter(value=1.0)
params['gamma_rv'] = radvel.Parameter(value=0.0)
```

Next, we need to set up a dictionary that tells RadVel how to construct a `radvel.Vector` object from the `radvel.Parameters` object. Using these indices, the model is then able to quickly read parameter values from the vector. Each index corresponds to a row in the vector, and every parameter used needs to be assigned a unique index.

[46]:
```
indices = {
    'tc': 0,
    'per': 1,
    'rp': 2,
    'a': 3,
    'inc': 4,
    'e': 5,
    'w': 6,
    'k': 7,
    'dvdt': 8,
    'curv': 9,
    'jit_trans': 10,
    'gamma_trans':11,
    'jit_rv': 12,
    'gamma_rv': 13
}
```

Using the indices defined above, we will now provide a function that defines the lightcurve signal as a function of time and parameters:

[47]:
```
def lightcurve_calc(t, params, vector):

    pars = batman.TransitParams()
    pars.t0 = vector.vector[0][0]
    pars.per = vector.vector[1][0]
    pars.rp = vector.vector[2][0]
    pars.a = vector.vector[3][0]
    pars.inc = vector.vector[4][0]
    pars.ecc = vector.vector[5][0]
    pars.w = vector.vector[6][0]
    pars.limb_dark = "uniform"
    pars.u = []
```

(continues on next page)

(continued from previous page)

```
    m = batman.TransitModel(pars, t)

    flux = m.light_curve(pars)
    return flux
```

We now need to use the same indices to define the radial velocities as a function of the time and parameters. We cannot use the default vector construction because there are shared parameters between the transit model and the radial velocity model; they need to pull values from the same vector.

[48]:
```
def rv_calc(t, params, vector):

    per = vector.vector[1][0]
    tp = radvel.orbit.timetrans_to_timeperi(tc=vector.vector[0][0], per=vector.
↪vector[1][0],
                                            ecc=vector.vector[5][0], omega=vector.
↪vector[6][0])
    e = vector.vector[5][0]
    w = vector.vector[6][0]
    k = vector.vector[7][0]
    orbel_synth = np.array([per, tp, e, w, k])
    vel = radvel.kepler.rv_drive(t, orbel_synth)

    return vel
```

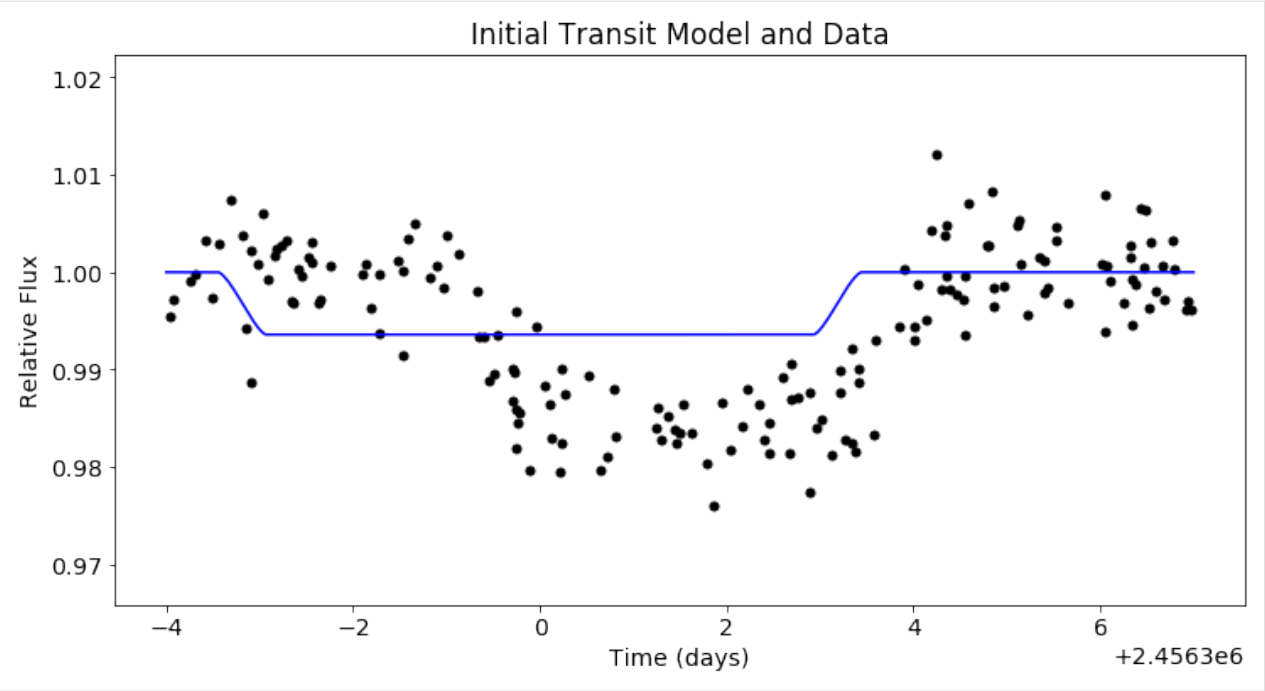
Using the functions that define our models, we can now construct `radvel.GeneralRVModel` objects. For the first model, we must override the default vector construction by calling `radvel.GeneralRVModel.vector.indices` and `radvel.GeneralRVModel.vector.dict_to_vector()`. For any additional models, we must set the vector equal to the initial model's vector, that way they are functions of the same parameters.

[50]:
```
mod_trans = radvel.GeneralRVModel(params, forward_model=lightcurve_calc)
mod_trans.vector.indices = indices
mod_trans.vector.dict_to_vector()

mod_rv = radvel.GeneralRVModel(params, forward_model=rv_calc)
mod_rv.vector = mod_trans.vector
```

We can now plot the data and initial models. When using custom parameters and models, built in plotting functions will rarely work.

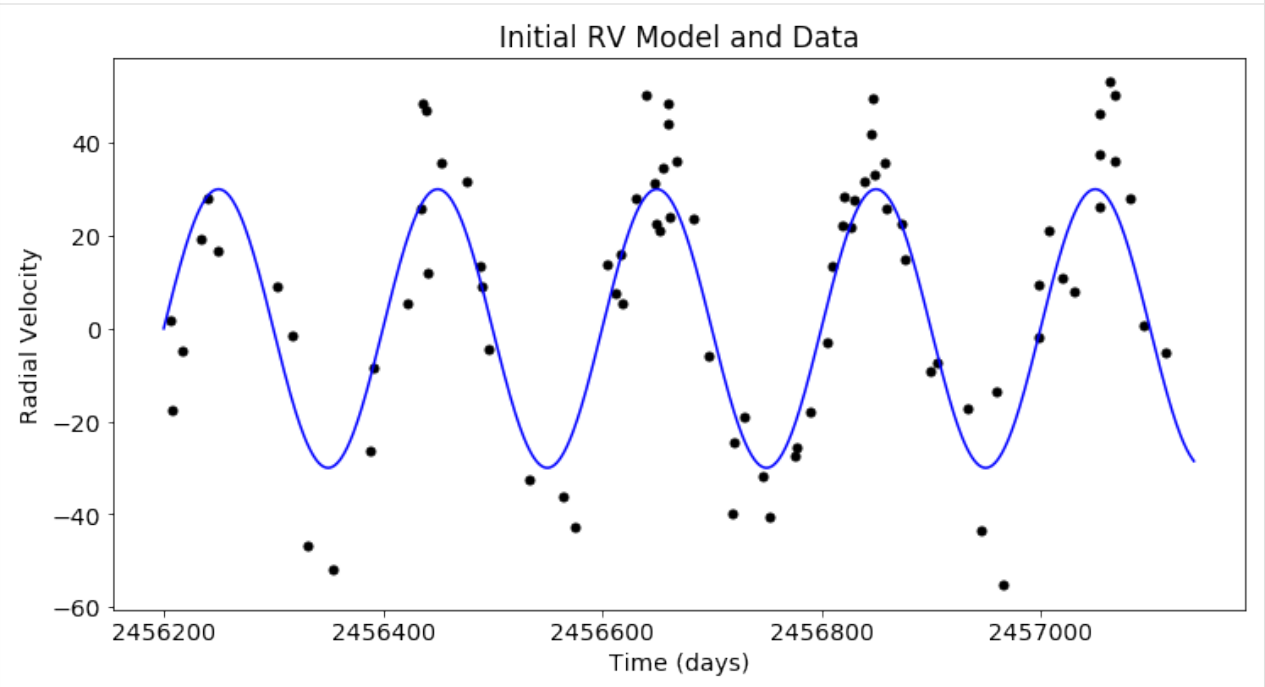
[29]:
```
t_trans = np.linspace(2456296, 2456307, 1000)
plt.figure(figsize=(12,6))
plt.scatter(x_trans, y_trans, c='black')
plt.plot(t_trans, mod_trans(t_trans), c='blue')
plt.xlabel('Time (days)')
plt.ylabel('Relative Flux')
plt.title('Initial Transit Model and Data')
```

[29]:
```
Text(0.5, 1.0, 'Initial Transit Model and Data')
```



[30]:
```
t_rv = np.linspace(2456200, 2457140, 1000)
plt.figure(figsize=(12,6))
plt.scatter(x_rv, y_rv, c='black')
plt.plot(t_rv, mod_rv(t_rv), c='blue')
plt.xlabel('Time (days)')
plt.ylabel('Radial Velocity')
plt.title('Initial RV Model and Data')
```

[30]:
```
Text(0.5, 1.0, 'Initial RV Model and Data')
```



Now that the models are ready, we need to set up our likelihood objects. Because we will be using a composite likelihood later on, it is easiest to use the `radvel.RVLikelihood` object. However, building off the generic `radvel.Likelihood` class is an option, allowing you to define your own methods and attributes.

[31]:
```
errors_trans = np.zeros(len(x_trans))
errors_trans.fill(yerr_trans)
like_trans = radvel.RVLikelihood(mod_trans, x_trans, y_trans, errors_trans, suffix='_
↪trans')

errors_rv = np.zeros(len(x_rv))
errors_rv.fill(yerr_rv)
like_rv = radvel.RVLikelihood(mod_rv, x_rv, y_rv, errors_rv, suffix='_rv')
```

Now that we have our individual likelihoods ready, we need to construct a composite likelihood:

[51]:
```
like = radvel.CompositeLikelihood([like_trans, like_rv])
```

Now we are ready to initialize the `radvel.Posterior` object. Note that most built in priors may be used on custom parameters.

[33]:
```
post = radvel.posterior.Posterior(like)
post.priors += [radvel.prior.HardBounds('rp',0,1)] #priors are useful to keep params in␣
↪physically possible boundaries
post.priors += [radvel.prior.HardBounds('a',5,30)]
```

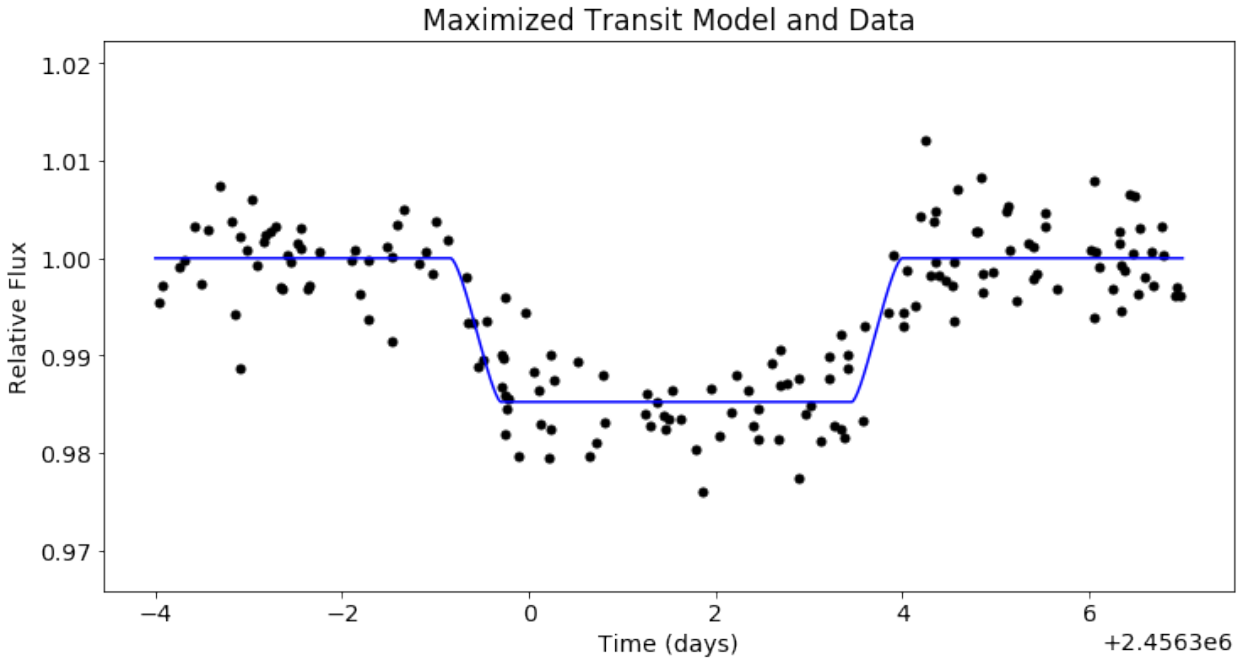
Maximize the likelihood, print the updated posterior object, and plot the newly fitted model:

[34]:
```
res  = optimize.minimize(
    post.neglogprob_array,
    post.get_vary_params(),
    method='Powell',
    )

print(post)
```

```
parameter                      value      vary
tc                         2.4563e+06       True
per                           199.918       True
a                             14.6554       True
rp                            0.12145       True
inc                           90.5981       True
e                                   0      False
w                                   0      False
k                             37.1623       True
jit_trans                -7.70543e-11       True
gamma_trans                         0      False
jit_rv                        4.85847       True
gamma_rv                     -2.06673       True

Priors
------
Bounded prior on rp, min=0, max=1
Bounded prior on a, min=5, max=30
```

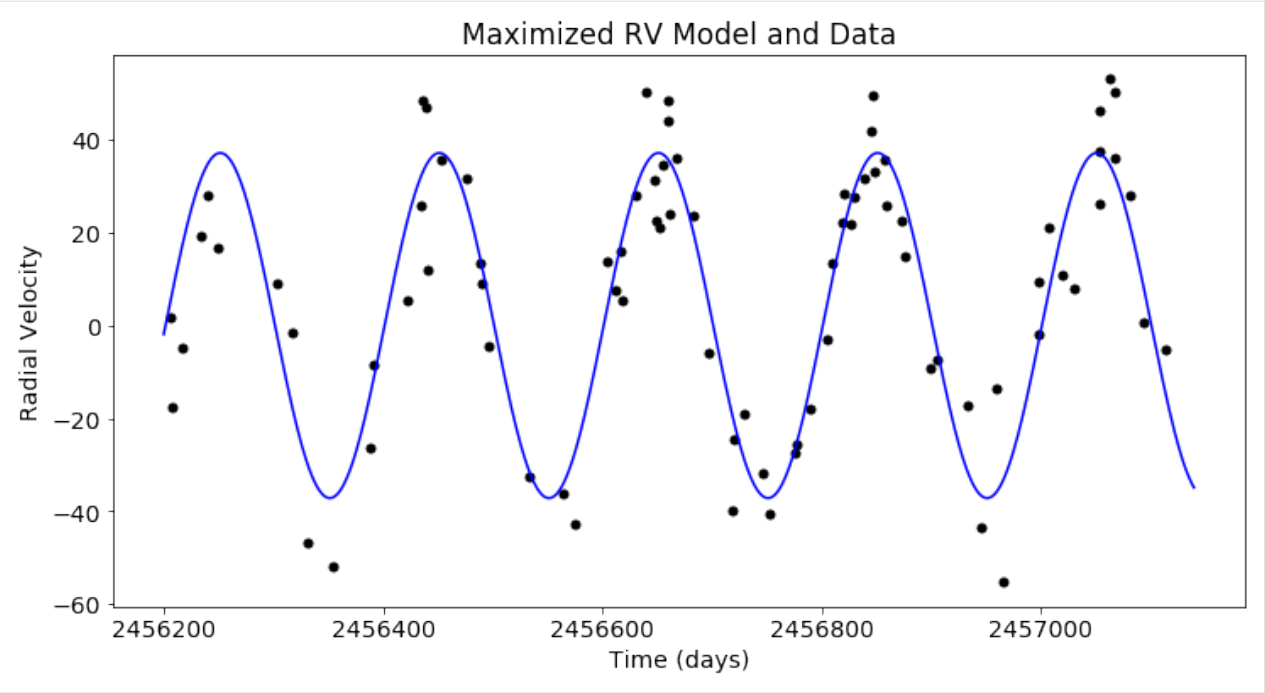
[35]:
```
plt.figure(figsize=(12,6))
plt.scatter(x_trans, y_trans, c='black')
plt.plot(t_trans, post.likelihood.like_list[0].model(t_trans), c='blue')
plt.xlabel('Time (days)')
plt.ylabel('Relative Flux')
plt.title('Maximized Transit Model and Data')
```

[35]:
```
Text(0.5, 1.0, 'Maximized Transit Model and Data')
```



[36]:
```
plt.figure(figsize=(12,6))
plt.scatter(x_rv, y_rv, c='black')
plt.plot(t_rv, post.likelihood.like_list[1].model(t_rv), c='blue')
plt.xlabel('Time (days)')
plt.ylabel('Radial Velocity')
plt.title('Maximized RV Model and Data')
```

[36]:
```
Text(0.5, 1.0, 'Maximized RV Model and Data')
```



Now lets use Markov-Chain Monte Carlo (MCMC) to estimate the parameter uncertainties. In this example we will run 500 steps for the sake of speed but in practice you should let it run at least 10000 steps and ~50 walkers.

[37]:
```
df = radvel.mcmc(post,nwalkers=50,nrun=500)
```

```
20000/200000 (10.0%) steps complete; Running 10117.35 steps/s; Mean acceptance rate = 36.
↪9%; Min Auto Factor =  19; Max Auto Relative-Change =   inf; Min Tz = 2691.6; Max G-R␣
↪= 1.010
Discarding burn-in now that the chains are marginally well-mixed

200000/200000 (100.0%) steps complete; Running 11106.51 steps/s; Mean acceptance rate =␣
↪22.8%; Min Auto Factor =  21; Max Auto Relative-Change = 0.121; Min Tz = 2962.5; Max G-
↪R = 1.010

MCMC: WARNING: chains did not pass convergence tests. They are likely not well-mixed.
```

Let's take a quick look at the parameter values and uncertainties. Additionally, we need to update the posterior for future plotting and other purposes.

[38]:
```
quants = df.quantile([0.159, 0.5, 0.841]) # median & 1sigma limits of posterior␣
↪distributions

par_array = []
for par in post.name_vary_params():
    med = quants[par][0.5]
    high = quants[par][0.841] - med
    low = med - quants[par][0.159]
    err = np.mean([high,low])
    err = radvel.utils.round_sig(err)
    par_array.append(med)
    med, err, errhigh = radvel.utils.sigfig(med, err)
```

(continues on next page)

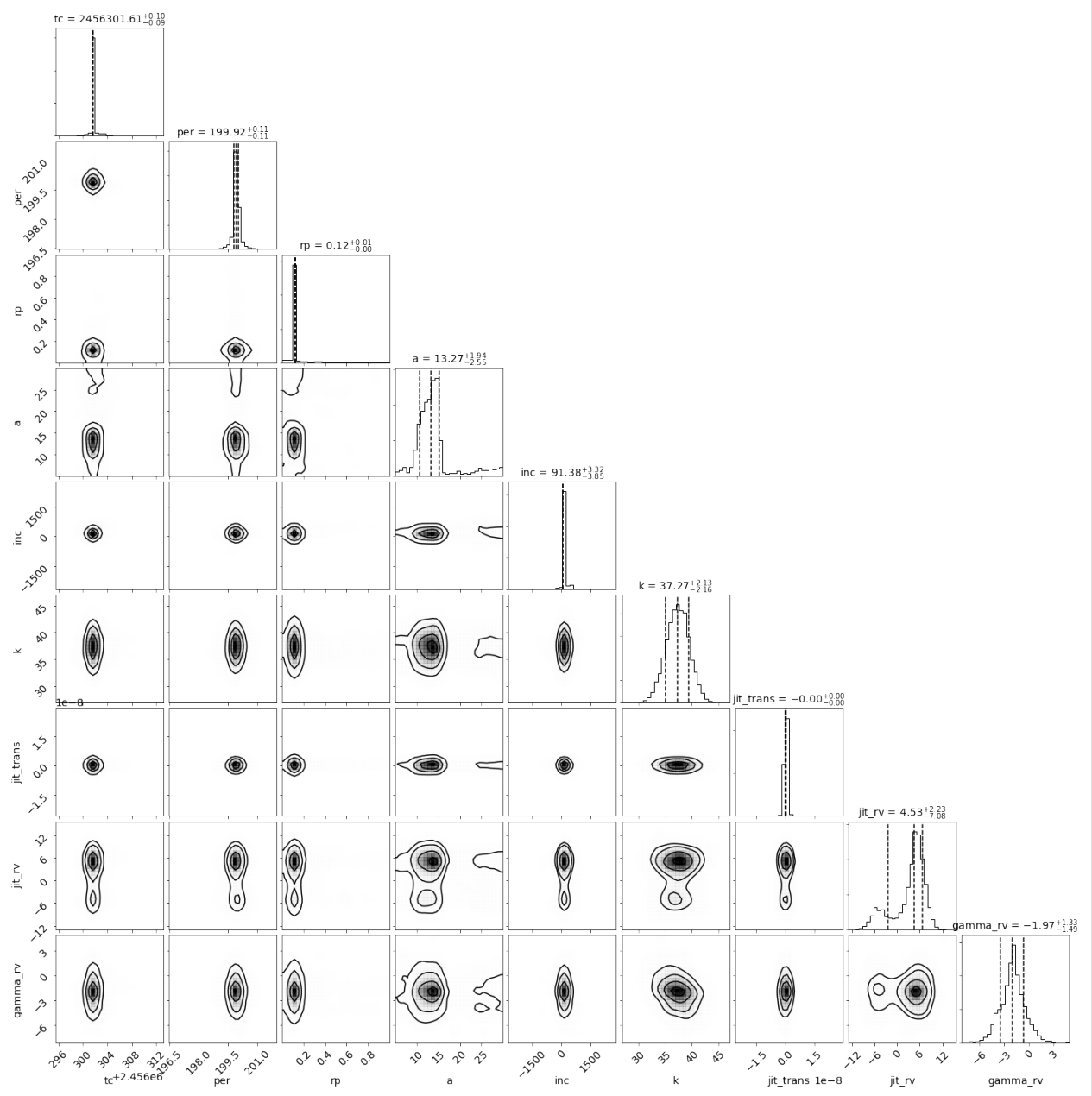
(continued from previous page)

```
    print('{} : {} +/- {}'.format(par, med, err))
post.set_vary_params(par_array)
```

```
tc : 2456301.614 +/- 0.091
per : 199.92 +/- 0.11
rp : 0.123 +/- 0.0048
a : 13.3 +/- 2.3
inc : 91.4 +/- 3.6
k : 37.3 +/- 2.2
jit_trans : -7e-11 +/- 2.1e-10
jit_rv : 4.5 +/- 4.7
gamma_rv : -2.0 +/- 1.4
```

Let's make a corner plot to display the posterior distributions:

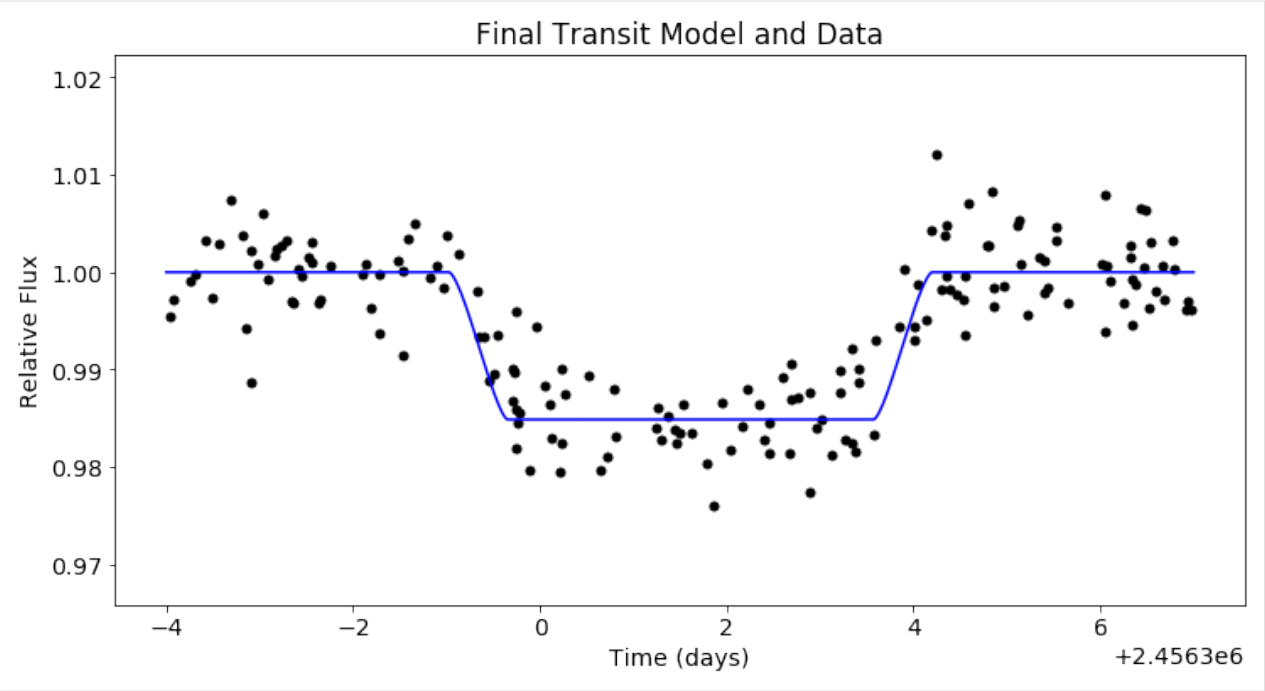
[39]:
```
_ = corner.corner(df[post.name_vary_params()], labels=post.name_vary_params(), label_
↪kwargs={"fontsize": 14},
            plot_datapoints=False, bins=30, quantiles=[0.16, 0.5, 0.84],
            show_titles=True, title_kwargs={"fontsize": 14}, smooth=True
        )
```



Finally, we can plot our MCMC model and data.

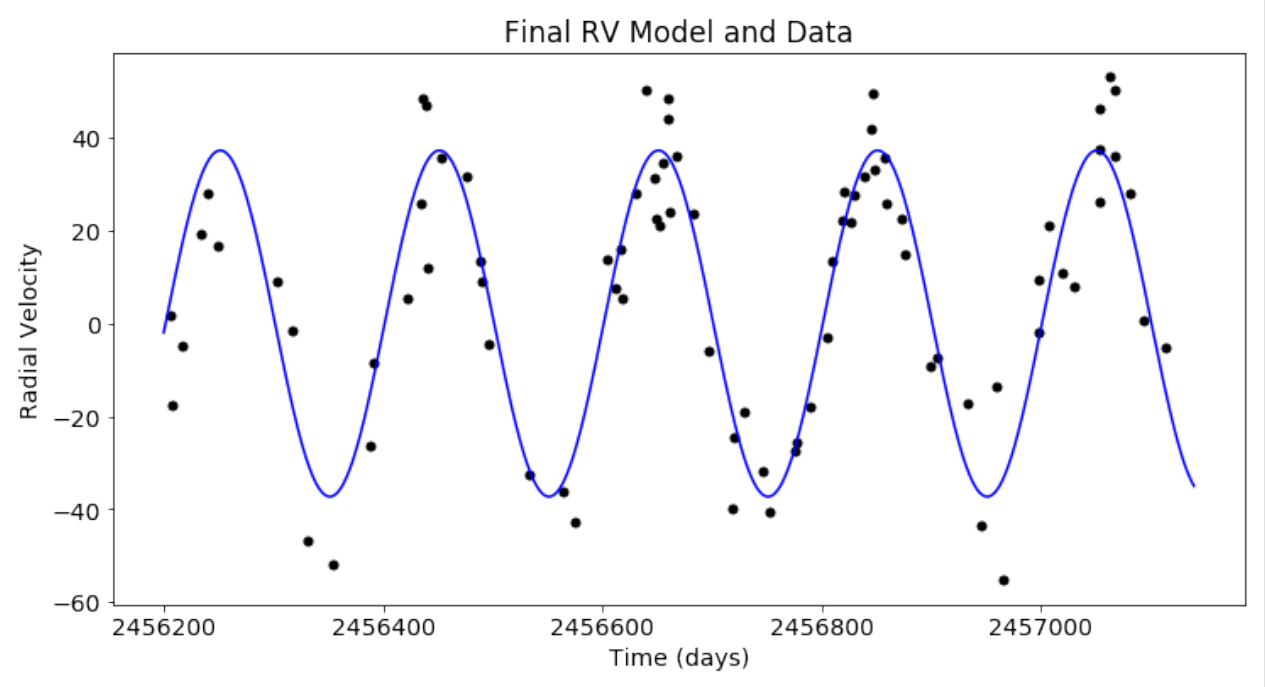
[40]:
```
plt.figure(figsize=(12,6))
plt.scatter(x_trans, y_trans, c='black')
plt.plot(t_trans, post.likelihood.like_list[0].model(t_trans), c='blue')
plt.xlabel('Time (days)')
plt.ylabel('Relative Flux')
plt.title('Final Transit Model and Data')
```

[40]:
```
Text(0.5, 1.0, 'Final Transit Model and Data')
```



[41]:
```
plt.figure(figsize=(12,6))
plt.scatter(x_rv, y_rv, c='black')
plt.plot(t_rv, post.likelihood.like_list[1].model(t_rv), c='blue')
plt.xlabel('Time (days)')
plt.ylabel('Radial Velocity')
plt.title('Final RV Model and Data')
```

[41]:
```
Text(0.5, 1.0, 'Final RV Model and Data')
```



CHAPTER

FOUR

API

4.1 The RV Model

class radvel.model.**GeneralRVModel**(*params*, *forward_model*, *time_base=0*)

A generalized Model

Parameters

- **params** (*radvel.Parameters*) – The parameters upon which the RV model depends.
- **forward_model** (*callable*) – The function that defines the signal as a function of time and parameters. The forward model is called as

  ```
  forward_model(time, params, *args, **kwargs) -> float
  ```

- **time_base** (*float*) – time relative to which 'dvdt' and 'curv' terms are computed.

Examples

```
>>> import radvel
#  In this example, we'll assume a function called 'my_rv_function' that
#  computes RV values has been defined elsewhere. We'll assume that
#  'my_rv_function' depends on planets' usual RV parameters
#  contained in radvel.Parameters as well as some additional
#  parameter, 'my_param'.
>>> params = radvel.Parameters(2)
>>> params['my_param'] = rv.Parameter(my_param_value,vary=True)
>>> rvmodel = radvel.GeneralRVModel(myparams,my_rv_function)
>>> rv = rvmodel(10)
```

__call__(*t*, **args*, ***kwargs*)

Compute the signal

Parameters
t (*array of floats*) – Timestamps to calculate the model

Returns
model at each time in *t*

Return type
vel (array of floats)

__init__(*params*, *forward_model*, *time_base=0*)

__weakref__

list of weak references to the object (if defined)

class radvel.model.**Parameter**(*value=None*, *vary=True*, *mcmcscale=None*, *linear=False*)

Object to store attributes of each orbital parameter .. attribute:: value

value of parameter.

type
float

vary

True if parameter is allowed to vary in MCMC or max likelihood fits, false if fixed

Type
Bool

mcmcscale

step size to be used for MCMC fitting

Type
float

linear

if vary=False and linear=True for gamma parameters then they will be calculated analytically using the trick. derived by Timothy Brandt.

Type
bool

__init__(*value=None*, *vary=True*, *mcmcscale=None*, *linear=False*)

__repr__()

Return repr(self).

__weakref__

list of weak references to the object (if defined)

class radvel.model.**Parameters**(*num_planets*, *basis='per tc secosw sesinw logk'*, *planet_letters=None*)

Object to store the model parameters.

Parameters to describe a radial velocity orbit stored as an OrderedDict.

Parameters

- **num_planets** (*int*) – Number of planets in model
- **basis** (*string*) – parameterization of orbital parameters. See radvel.basis.Basis for a list of valid basis strings.
- **planet_letters** (*dict [optional]*) – custom map to match the planet numbers in the Parameter object to planet letters. Default {1: 'b', 2: 'c', etc.}. The keys of this dictionary must all be integers.

basis

Basis object

Type
radvel.Basis

planet_parameters
orbital parameters contained within the specified basis

Type
list

num_planets
number of planets in the model

Type
int

Examples

```
>>> import radvel
# create a Parameters object for a 2-planet system with
# custom planet number to letter mapping
>>> params = radvel.Parameters(2, planet_letters={1:'d', 2:'e'})
```

__init__(*num_planets*, *basis='per tc secosw sesinw logk'*, *planet_letters=None*)

__reduce__()
Return state information for pickling

tex_labels(*param_list=None*)
Map Parameters keys to pretty TeX code representations.

Parameters
param_list (*list [optional]*) – Manually pass a list of parameter labels

Returns
dictionary mapping Parameters keys to TeX code

Return type
dict

class radvel.model.**RVModel**(*params*, *time_base=0*)
Generic RV Model

This class defines the methods common to all RV modeling classes. The different RV models, with different parameterizations, all inherit from this class.

__init__(*params*, *time_base=0*)

4.2 Keplerian Orbits

radvel.kepler.**kepler**(*Marr*, *eccarr*)
Solve Kepler's Equation :param Marr: input Mean anomaly :type Marr: array :param eccarr: eccentricity :type eccarr: array

Returns
eccentric anomaly

Return type
array

radvel.kepler.**rv_drive**(*t*, *orbel*, *use_c_kepler_solver=True*)

RV Drive :param t: times of observations :type t: array of floats :param orbel: [per, tp, e, om, K]. Omega is expected to be in radians :type orbel: array of floats :param use_c_kepler_solver: (default: True) If True use the Kepler solver written in C, else use the Python/NumPy version. :type use_c_kepler_solver: bool

Returns
(array of floats): radial velocity model

Return type
rv

radvel.orbit.**timeperi_to_timetrans**(*tp*, *per*, *ecc*, *omega*, *secondary=False*)

Convert Time of Periastron to Time of Transit

Parameters

- **tp** (*float*) – time of periastron
- **per** (*float*) – period [days]
- **ecc** (*float*) – eccentricity
- **omega** (*float*) – argument of peri (radians)
- **secondary** (*bool*) – calculate time of secondary eclipse instead

Returns
time of inferior conjunction (time of transit if system is transiting)

Return type
float

radvel.orbit.**timetrans_to_timeperi**(*tc*, *per*, *ecc*, *omega*)

Convert Time of Transit to Time of Periastron Passage

Parameters

- **tc** (*float*) – time of transit
- **per** (*float*) – period [days]
- **ecc** (*float*) – eccentricity
- **omega** (*float*) – longitude of periastron (radians)

Returns
time of periastron passage

Return type
float

radvel.orbit.**true_anomaly**(*t*, *tp*, *per*, *e*)

Calculate the true anomaly for a given time, period, eccentricity.

Parameters

- **t** (*array*) – array of times in JD
- **tp** (*float*) – time of periastron, same units as t
- **per** (*float*) – orbital period in days
- **e** (*float*) – eccentricity

Returns
true anomoly at each time

Return type
array

4.3 Orbital Parameter Basis Sets

class radvel.basis.**Basis**(**args*)

Object that knows how to convert between the various Keplerian bases

Parameters

- **name** (*str*) – basis name
- **num_planets** (*int*) – number of planets

synth_params

name of synth basis

Type
str

Note: Valid basis functions:

'per tp e w k' (The synthesis basis)

'per tc secosw sesinw logk'

'per tc secosw sesinw k'

'per tc ecosw esinw k'

'per tc e w k'

'logper tc secosw sesinw k'

'logper tc secosw sesinw logk'

'per tc se w k'

---

from_synth(*params_in*, *newbasis*, ***kwargs*)

Convert from synth basis into another basis

Convert instance of Parameters with parameters of a given basis into the synth basis

Parameters

- **params_in** (*radvel.Parameters or pandas.DataFrame*) – radvel.Parameters object or pandas.Dataframe containing orbital parameters expressed in current basis
- **newbasis** (*string*) – string corresponding to basis to switch into
- **keep** (*bool [optional]*) – keep the parameters expressed in the old basis, else remove them from the output dictionary/DataFrame

Returns
dict or dataframe with the parameters converted into the new basis

get_circparams()

Return the 3 parameters for a circular orbit of a plent in the object's basis

Returns
the params for a circular orbit

get_eparams()

Return the eccentricity parameters for the object's basis

Returns
the params which have to do with eccentricity

to_any_basis(*params_in*, *newbasis*)

Convenience function for converting Parameters object to an arbitraty basis

Parameters

- **params_in** (*radvel.Parameters*) – radvel.Parameters object expressed in current basis
- **newbasis** (*string*) – string corresponding to basis to switch into

Returns
radvel.Parameters object expressed in the new basis

to_synth(*params_in*, ***kwargs*)

Convert to synth basis Convert Parameters object with parameters of a given basis into the synth basis :param params_in: radvel.Parameters object or pandas.Dataframe containing

orbital parameters expressed in current basis

Parameters
noVary (*bool [optional]*) – if True, set the 'vary' attribute of the returned Parameter objects to '' (used for displaying best fit parameters)

Returns
parameters expressed in the synth basis

Return type
Parameters or DataFrame

4.4 Kernels for GPs

class radvel.gp.**CeleriteKernel**(*hparams*)

Class that computes and stores a matrix approximating the quasi-periodic kernel.

See *radvel/example_planets/k2-131_celerite.py* for an example of a setup file that uses this Kernel object.

See celerite.readthedocs.io and Foreman-Mackey et al. 2017. AJ, 154, 220 (equation 56) for more details.

An arbitrary element, C_{ij}, of the matrix is:

$$C_{ij} = B/(2+C) * exp(-|t_i - t_j|/L) * (\cos(\frac{2\pi|t_i - t_j|}{P_{rot}}) + (1+C))$$

Parameters
hparams (*dict of radvel.Parameter*) – dictionary containing radvel.Parameter objects that are GP hyperparameters of this kernel. Must contain exactly four objects, 'gp_B*', 'gp_C*', 'gp_L*', and 'gp_Prot*', where * is a suffix identifying these hyperparameters with a likelihood object.

compute_covmatrix(*errors*)

Compute the Cholesky decomposition of a celerite kernel

Parameters
errors (*array of float*) – observation errors and jitter added in quadrature

Returns
the celerite solver object, with Cholesky decomposition computed.

Return type
celerite.solver.CholeskySolver

`compute_distances`(*x1*, *x2*)

The celerite.solver.CholeskySolver object does not require distances to be precomputed, so this method has been co-opted to define some unchanging variables.

`class radvel.gp.Kernel`

Abstract base class to store kernel info and compute covariance matrix. All kernel objects inherit from this class.

Note: To implement your own kernel, create a class that inherits from this class. It should have hyperparameters that follow the name scheme 'gp_NAME_SUFFIX'.

`class radvel.gp.PerKernel`(*hparams*)

Class that computes and stores a periodic kernel matrix. An arbitrary element, C_{ij}, of the matrix is:

$$C_{ij} = \eta_1^2 * exp(\frac{-\sin^2(\frac{\pi|t_i - t_j|}{\eta_3^2})}{2\eta_2^2})$$

Parameters
hparams (*dict of radvel.Parameter*) – dictionary containing radvel.Parameter objects that are GP hyperparameters of this kernel. Must contain exactly three objects, 'gp_length*', 'gp_amp*', and 'gp_per*', where * is a suffix identifying these hyperparameters with a likelihood object.

`compute_covmatrix`(*errors*)

Compute the covariance matrix, and optionally add errors along the diagonal.

Parameters
errors (*float or numpy array*) – If covariance matrix is non-square, this arg must be set to 0. If covariance matrix is square, this can be a numpy array of observational errors and jitter added in quadrature.

`class radvel.gp.QuasiPerKernel`(*hparams*)

Class that computes and stores a quasi periodic kernel matrix. An arbitrary element, C_{ij}, of the matrix is:

$$C_{ij} = \eta_1^2 * exp(\frac{-|t_i - t_j|^2}{\eta_2^2} - \frac{\sin^2(\frac{\pi|t_i - t_j|}{\eta_3})}{2\eta_4^2})$$

Parameters
hparams (*dict of radvel.Parameter*) – dictionary containing radvel.Parameter objects that are GP hyperparameters of this kernel. Must contain exactly four objects, 'gp_explength*', 'gp_amp*', 'gp_per*', and 'gp_perlength*', where * is a suffix identifying these hyperparameters with a likelihood object.

`compute_covmatrix`(*errors*)

Compute the covariance matrix, and optionally add errors along the diagonal.

Parameters
errors (*float or numpy array*) – If covariance matrix is non-square, this arg must be set to 0. If covariance matrix is square, this can be a numpy array of observational errors and jitter added in quadrature.

class radvel.gp.**SqExpKernel**(*hparams*)

Class that computes and stores a squared exponential kernel matrix. An arbitrary element, C_{ij}, of the matrix is:

$$C_{ij} = \eta_1^2 * exp(\frac{-|t_i - t_j|^2}{\eta_2^2})$$

Parameters

hparams (*dict of radvel.Parameter*) – dictionary containing radvel.Parameter objects that are GP hyperparameters of this kernel. Must contain exactly two objects, 'gp_length*' and 'gp_amp*', where * is a suffix identifying these hyperparameters with a likelihood object.

compute_covmatrix(*errors*)

Compute the covariance matrix, and optionally add errors along the diagonal.

Parameters

errors (*float or numpy array*) – If covariance matrix is non-square, this arg must be set to 0. If covariance matrix is square, this can be a numpy array of observational errors and jitter added in quadrature.

4.5 Likelihood Functions

class radvel.likelihood.**CeleriteLikelihood**(*model*, *t*, *vel*, *errvel*, *hnames*, *suffix=''*, ***kwargs*)

Celerite GP Likelihood The Likelihood object for a radial velocity dataset modeled with a GP whose kernel is an approximation to the quasi-periodic kernel. See celerite.readthedocs.io and Foreman-Mackey et al. 2017. AJ, 154, 220 (equation 56) for more details. See *radvel/example_planets/k2-131_celerite.py* for an example of a setup file that uses this Likelihood object. :param model: RVModel object :type model: radvel.model.RVModel :param t: time array :type t: array :param vel: array of velocities :type vel: array :param errvel: array of velocity uncertainties :type errvel: array :param hnames: keys corresponding to radvel.Parameter

objects in model.params that are GP hyperparameters

Parameters

suffix (*string*) – suffix to identify this Likelihood object; useful when constructing a *CompositeLikelihood* object

logprob()

Return GP log-likelihood given the data and model. log-likelihood is computed using Cholesky decomposition as: .. math:

```
lnL = -0.5r^TK^{-1}r - 0.5ln[det(K)] - 0.5N*ln(2pi)
```

where r = vector of residuals (GPLikelihood._resids), K = covariance matrix, and N = number of datapoints. Priors are not applied here. Constant has been omitted. :returns: Natural log of likelihood :rtype: float

predict(*xpred*)

Realize the GP using the current values of the hyperparameters at values x=xpred. Used for making GP plots. Wrapper for *celerite.GP.predict()*. :param xpred: numpy array of x values for realizing the GP :type xpred: np.array

Returns

tuple containing:

np.array: numpy array of predictive means

np.array: numpy array of predictive standard deviations

Return type
tuple

class radvel.likelihood.**CompositeLikelihood**(*like_list*)

Composite Likelihood A thin wrapper to combine multiple *Likelihood* objects. One *Likelihood* applies to a dataset from a particular instrument. :param like_list: list of *radvel.likelihood.RVLikelihood* objects :type like_list: list

errorbars()

See *radvel.likelihood.RVLikelihood.errorbars*

logprob()

See *radvel.likelihood.RVLikelihood.logprob*

residuals()

See *radvel.likelihood.RVLikelihood.residuals*

class radvel.likelihood.**GPLikelihood**(*model*, *t*, *vel*, *errvel*, *hnames=['gp_per', 'gp_perlength', 'gp_explength', 'gp_amp']*, *suffix=''*, *kernel_name='QuasiPer'*, ***kwargs*)

GP Likelihood The Likelihood object for a radial velocity dataset modeled with a GP :param model: GP model object :type model: radvel.model.GPModel :param t: time array :type t: array :param vel: array of velocities :type vel: array :param errvel: array of velocity uncertainties :type errvel: array :param hnames: keys corresponding to radvel.Parameter

objects in model.params that are GP hyperparameters

Parameters
suffix (*string*) – suffix to identify this Likelihood object; useful when constructing a *CompositeLikelihood* object

logprob()

Return GP log-likelihood given the data and model. log-likelihood is computed using Cholesky decomposition as: .. math:

```
lnL = -0.5r^TK^{-1}r - 0.5ln[det(K)] - 0.5N*ln(2pi)
```

where r = vector of residuals (GPLikelihood._resids), K = covariance matrix, and N = number of datapoints. Priors are not applied here. Constant has been omitted. :returns: Natural log of likelihood :rtype: float

predict(*xpred*)

Realize the GP using the current values of the hyperparameters at values x=xpred. Used for making GP plots. :param xpred: numpy array of x values for realizing the GP :type xpred: np.array

Returns

tuple containing:
np.array: the numpy array of predictive means

np.array: the numpy array of predictive standard deviations

Return type
tuple

residuals()

Residuals Data minus (orbit model + predicted mean of GP noise model). For making GP plots.

update_kernel_params()

Update the Kernel object with new values of the hyperparameters

class radvel.likelihood.**Likelihood**(*model*, *x*, *y*, *yerr*, *extra_params=[]*, *decorr_params=[]*, *decorr_vectors=[]*)

Generic Likelihood

aic()

Calculate the Aikike information criterion The Small Sample AIC (AICC) is returned because for most RV data sets n < 40 * k (see Burnham & Anderson 2002 S2.4). :returns: AICC :rtype: float

bic()

Calculate the Bayesian information criterion :returns: BIC :rtype: float

class radvel.likelihood.**RVLikelihood**(*model*, *t*, *vel*, *errvel*, *suffix=''*, *decorr_vars=[]*, *decorr_vectors=[]*, ***kwargs*)

RV Likelihood The Likelihood object for a radial velocity dataset :param model: RV model object :type model: radvel.model.RVModel :param t: time array :type t: array :param vel: array of velocities :type vel: array :param errvel: array of velocity uncertainties :type errvel: array :param suffix: suffix to identify this Likelihood object

useful when constructing a *CompositeLikelihood* object.

errorbars()

Return uncertainties with jitter added in quadrature. :returns: uncertainties :rtype: array

logprob()

Return log-likelihood given the data and model. Priors are not applied here. :returns: Natural log of likelihood :rtype: float

residuals()

Residuals Data minus model

radvel.likelihood.**loglike_jitter**(*residuals*, *sigma*, *sigma_jit*)

Log-likelihood incorporating jitter See equation (1) in Howard et al. 2014. Returns loglikelihood, where sigma**2 is replaced by sigma**2 + sigma_jit**2. It penalizes excessively large values of jitter :param residuals: array of residuals :type residuals: array :param sigma: array of measurement errors :type sigma: array :param sigma_jit: jitter :type sigma_jit: float

Returns
log-likelihood

Return type
float

4.6 The Posterior Object

class radvel.posterior.**Posterior**(*likelihood*)

Posterior object Posterior object to be sent to the fitting routines. It is essentially the same as the Likelihood object, but priors are applied here. :param likelihood: Likelihood object :type likelihood: radvel.likelihood.Likelihood :param params: parameters object :type params: radvel.model.Parameters

Note: Append *radvel.prior.Prior* objects to the Posterior.priors list to apply priors in the likelihood calculations.

aic()

Moved to Likelihood.aic

bic()

Moved to Likelihood.bic

logprob()

Log probability Log-probability for the likelihood given the list of priors in *Posterior.priors*. :returns: log probability of the likelihood + priors :rtype: float

logprob_array(*param_values_array*)

Log probability for parameter vector Same as *self.logprob*, but will take a vector of parameter values. Useful as the objective function for routines that optimize a vector of parameter values instead of the dictionary-like format of the *radvel.model.Parameters* object. :returns: log probability of the likelihood + priors :rtype: float

residuals()

Overwrite inherited residuals method that does not work

writeto(*filename*)

Save posterior object to pickle file. :param filename: full path to outputfile :type filename: string

radvel.posterior.**load**(*filename*)

Load posterior object from pickle file. :param filename: full path to pickle file :type filename: string

4.7 Priors

class radvel.prior.**EccentricityPrior**(*num_planets*, *upperlims=0.99*)

Physical eccentricities

Prior to keep eccentricity between 0 and a specified upper limit.

Parameters

- **num_planets** (*int or list*) – Planets to apply the eccentricity prior. If an integer is given then all planets with indexes up to and including the specified integer will be included in the prior. If a list is given then the prior will only be applied to the specified planets.
- **upperlims** (*float or list of floats*) – List of eccentricity upper limits to assign to each of the planets. If a float is given then all planets must have eccentricities less then this value. If a list of floats is given then each planet can have a different eccentricity upper limit.

class radvel.prior.**Gaussian**(*param*, *mu*, *sigma*)

Gaussian prior

Guassian prior on a given parameter.

Parameters

- **param** (*string*) – parameter label
- **mu** (*float*) – center of Gaussian prior
- **sigma** (*float*) – width of Gaussian prior

class radvel.prior.**HardBounds**(*param*, *minval*, *maxval*)

Prior for hard boundaries

This prior allows for hard boundaries to be established for a given parameter.

Parameters

- **param** (*string*) – parameter label
- **minval** (*float*) – minimum allowed value
- **maxval** (*float*) – maximum allowed value

class radvel.prior.**InformativeBaselinePrior**(*param*, *baseline*, *duration=0.0*)

Informative baseline prior suggested by A. Vanderburg (see Blunt et al. 2019).

This prior follows the distribution:

$$p(x) \propto 1 \text{ if } x - t_d B$$
$$\propto (B + t_d)/x \text{ else}$$

with upper bound.

Parameters

- **param** (*string*) – parameter label
- **baseline** (*float*) – B in eq above
- **duration** (*float*) – t_d in eq above (default: 0.0)

class radvel.prior.**Jeffreys**(*param*, *minval*, *maxval*)

Jeffrey's prior

This prior follows the distribution:

$$p(x) \propto \frac{1}{x}$$

with upper and lower bounds to prevent singularity at $x = 0$.

Parameters

- **param** (*string*) – parameter label
- **minval** (*float*) – minimum allowed value
- **maxval** (*float*) – maximum allowed value

class radvel.prior.**ModifiedJeffreys**(*param*, *minval*, *maxval*, *kneeval*)

Modified Jeffry's prior

This prior follows the distribution:

$$p(x) \propto \frac{1}{x - x_0}$$

with upper bound.

Parameters

- **param** (*string*) – parameter label
- **kneeval** (*float*) – "knee" of Jeffrey's prior (x_0 in eq above)
- **minval** (*float*) – minimum allowed value. *minval* must be larger than *kneeval*
- **maxval** (*float*) – maximum allowed value

class radvel.prior.**NumericalPrior**(*param_list*, *values*, *bw_method=None*)

Prior defined by an input array of values

Wrapper for scipy.stats.gaussian_kde.

This prior uses Gaussian Kernel Density Estimation to estimate the probability density function from which a set of values are randomly drawn.

Useful for defining a prior given a posterior obtained from a complementary fitting process. For example, you might use transit data to obtain constraints on secosw and sesinw, then use the posterior on secosw as a prior for a RadVel fit.

Parameters

- **param_list** (*list of str*) – list of parameter label(s).
- **values** (*numpy array of float*) – values of `param` you wish to use to define this prior. For example, this might be a posterior array of values of secosw derived from transit data. In case of univariate data this is a 1-D array, otherwise a 2-D array with shape (# of elements in param_list, # of data points).
- **bw_method** (*str, scalar, or callable [optional]*) – see scipy.stats.gaussian_kde

Note: the larger the input array of values, the longer it will take for calls to this prior to be evaluated. Consider thinning large input arrays to speed up performance.

class radvel.prior.**PositiveKPrior**(*num_planets*)

K must be positive

A prior to prevent K going negative. Be careful with this as it can introduce a bias to larger K values.

Parameters
num_planets (*int*) – Number of planets. Used to ensure K for each planet is positive

class radvel.prior.**SecondaryEclipsePrior**(*planet_num*, *ts*, *ts_err*)

Secondary eclipse prior

Implied prior on eccentricity and omega by specifying measured secondary eclipse time

Parameters

- **planet_num** (*int*) – Number of planet with measured secondary eclipse
- **ts** (*float*) – Secondary eclipse midpoint time. Should be in the same units as the timestamps of your data.
- **ts_err** (*float*) – Uncertainty on secondary eclipse time

class radvel.prior.**UserDefinedPrior**(*param_list*, *func*, *tex_rep*)

Interface for user to define a prior
with an arbitrary functional form.

Parameters

- **param_list** (*list of str*) – list of parameter label(s).
- **func** (*function*) – a Python function that takes in a list of values (ordered as in `param_list`), and returns the corresponding log-value of a pdf.
- **tex_rep** (*str*) – TeX-readable string representation of this prior, to be passed into radvel report and plotting code.

Example

```
>>> def myPriorFunc(inp_list):
...     if inp_list[0] > 0. and inp_list[0] < 1.:
...         return 0.
...     else:
...         return -np.inf
>>> myTexString = 'Uniform Prior on $\sqrt{e}$'
>>> myPrior = radvel.prior.UserDefinedPrior(['se'], myPriorFunc, myTexString)
```

Note: `func` must be properly normalized; i.e. integrating over the entire parameter space must give a probability of 1.

4.8 Maximum A Posteriori Fitting

`radvel.fitting.`**`maxlike_fitting`**(*post*, *verbose=True*, *method='Powell'*)

Maximum A Posteriori Fitting

Perform a maximum a posteriori fit.

Parameters

- **post** (*radvel.Posterior*) – Posterior object with initial guesses
- **verbose** (*bool [optional]*) – Print messages and fitted values?
- **method** (*string [optional]*) – Minimization method. See documentation for *scipy.optimize.minimize* for available options.

Returns

Posterior object with parameters updated to their maximum a posteriori values

Return type

radvel.Posterior

`radvel.fitting.`**`model_comp`**(*post*, *params=[]*, *mc_list=[]*, *verbose=False*)

Model Comparison

Vary the presence of additional parameters and check how the improve the model fit Save results as list of dictionaries of posterior statistics.

Parameters

- **post** (*radvel.Posterior*) – posterior object for final best-fit solution with all planets
- **params** (*list of strings*) – (optional) type of comparison to make via bic/aic
- **mc_list** (*list of OrderedDicts*) – (optional) list of dictionaries from different model comparisons. Each value in the dictionary is a tuple with a statistic as the first element and a description as the second element.
- **verbose** (*bool*) – (optional) print out statistics

Returns

List of dictionaries with fit statistics. Each value in the dictionary is a tuple with the statistic value as the first element and a description of that statistic in the second element.

Return type
list of OrderedDicts

4.9 MCMC Fitting

radvel.mcmc.**convergence_calculate**(*chains*, *oldautocorrelation*, *minAfactor*, *maxArchange*, *minTz*, *maxGR*)

Calculate Convergence Criterion

Calculates the Gelman-Rubin statistic, autocorrelation time factor, relative change in autocorrellation time, and the number of independent draws for each parameter, as defined by Ford et al. (2006) (http://adsabs.harvard.edu/abs/2006ApJ…642..505F). The chain is considered well-mixed if all parameters have a Gelman-Rubin statistic of <= 1.03, the min autocorrelation time factor >= 75, a max relative change in autocorrelation time <= .01, and >= 1000 independent draws.

Parameters

- **chains** (*array*) – A 3 dimensional array of parameter values
- **oldautocorrelation** (*float*) – previously calculated autocorrelation time
- **minAfactor** (*float*) – minimum autocorrelation time factor to consider well-mixed
- **maxArchange** (*float*) – maximum relative change in autocorrelation time to consider well-mixed
- **minTz** (*int*) – minimum Tz to consider well-mixed
- **maxGR** (*float*) – maximum Gelman-Rubin statistic to consider well-mixed

Returns

tuple containing:

ismixed (bool):
Are the chains well-mixed?

afactor (array):
A matrix containing the autocorrelation time factor for each parameter and ensemble combination

archange (matrix):
A matrix containing the relative change in the autocorrelation time factor for each parameter and ensemble combination

autocorrelation (matrix):
A matrix containing the autocorrelation time for each parameter and ensemble combination

gelmanrubin (array):
An NPARS element array containing the Gelman-Rubin statistic for each parameter (equation 25)

Tz (array):
An NPARS element array containing the number of independent draws for each parameter (equation 26)

Return type
tuple

History:

2010/03/01:
Written: Jason Eastman - The Ohio State University

2012/10/08:
Ported to Python by BJ Fulton - University of Hawaii, Institute for Astronomy

2016/04/20:
Adapted for use in RadVel. Removed "angular" parameter.

2019/10/24:
Adapted to calculate and consider autocorrelation times

radvel.mcmc.**convergence_check**(*minAfactor*, *maxArchange*, *maxGR*, *minTz*, *minsteps*, *minpercent*, *headless*)

Check for convergence

Check for convergence for a list of emcee samplers

Parameters

- **minAfactor** (*float*) – Minimum autocorrelation time factor for chains to be deemed well-mixed and halt the MCMC run
- **maxArchange** (*float*) – Maximum relative change in the autocorrelative time to be deemed well-mixed and halt the MCMC run
- **maxGR** (*float*) – Maximum G-R statistic for chains to be deemed well-mixed and halt the MCMC run
- **minTz** (*int*) – Minimum Tz to consider well-mixed
- **minsteps** (*int*) – Minimum number of steps per walker before convergence tests are performed. Convergence checks will start after the minsteps threshold or the minpercent threshold has been hit.
- **minpercent** (*float*) – Minimum percentage of total steps before convergence tests are performed. Convergence checks will start after the minsteps threshold or the minpercent threshold has been hit.
- **headless** (*bool*) – if set to true, the convergence statistics will not be displayed in real time.

radvel.mcmc.**mcmc**(*post*, *nwalkers=50*, *nrun=10000*, *ensembles=8*, *checkinterval=50*, *minAfactor=40*, *maxArchange=0.03*, *burnAfactor=25*, *burnGR=1.03*, *maxGR=1.01*, *minTz=1000*, *minsteps=1000*, *minpercent=5*, *thin=1*, *serial=False*, *save=False*, *savename=None*, *proceed=False*, *proceedname=None*, *headless=False*)

Run MCMC Run MCMC chains using the emcee EnsambleSampler :param post: radvel posterior object :type post: radvel.posterior :param nwalkers: (optional) number of MCMC walkers :type nwalkers: int :param nrun: (optional) number of steps to take :type nrun: int :param ensembles: (optional) number of ensembles to run. Will be run

in parallel on separate CPUs

Parameters

- **checkinterval** (*int*) – (optional) check MCMC convergence statistics every *checkinterval* steps
- **minAfactor** (*float*) – Minimum autocorrelation time factor to deem chains as well-mixed and halt the MCMC run
- **maxArchange** (*float*) – Maximum relative change in autocorrelation time to deem chains and well-mixed

- **burnAfactor** (*float*) – Minimum autocorrelation time factor to stop burn-in period. Burn-in ends once burnGr or burnAfactor are reached.
- **burnGR** (*float*) – (optional) Maximum G-R statistic to stop burn-in period. Burn-in ends once burnGr or burnAfactor are reached.
- **maxGR** (*float*) – (optional) Maximum G-R statistic for chains to be deemed well-mixed and halt the MCMC run
- **minTz** (*int*) – (optional) Minimum Tz to consider well-mixed
- **minsteps** (*int*) – Minimum number of steps per walker before convergence tests are performed. Convergence checks will start after the minsteps threshold or the minpercent threshold has been hit.
- **minpercent** (*float*) – Minimum percentage of total steps before convergence tests are performed. Convergence checks will start after the minsteps threshold or the minpercent threshold has been hit.
- **thin** (*int*) – (optional) save one sample every N steps (default=1, save every sample)
- **serial** (*bool*) – set to true if MCMC should be run in serial
- **save** (*bool*) – set to true to save MCMC chains that can be continued in a future run
- **savename** (*string*) – location of h5py file where MCMC chains will be saved for future use
- **proceed** (*bool*) – set to true to continue a previously saved run
- **proceedname** (*string*) – location of h5py file with previously MCMC run chains
- **headless** (*bool*) – if set to true, the convergence statistics will not display in real time

Returns
DataFrame containing the MCMC samples

Return type
DataFrame

4.10 Pipeline Driver Functions

Driver functions for the radvel pipeline.These functions are meant to be used only withthe *cli.py* command line interface.

radvel.driver.**derive**(*args*)

Derive physical parameters from posterior samples

Parameters
args (*ArgumentParser*) – command line arguments

radvel.driver.**fit**(*args*)

Perform maximum a posteriori fit

Parameters
args (*ArgumentParser*) – command line arguments

radvel.driver.**ic_compare**(*args*)

Compare different models and comparative statistics including
AIC and BIC statistics.

Parameters
args (*ArgumentParser*) – command line arguments

radvel.driver.**load_status**(*statfile*)

Load pipeline status

Parameters
statfile (*string*) – name of configparser file

Returns
configparser.RawConfigParser

radvel.driver.**mcmc**(*args*)

Perform MCMC error analysis

Parameters
args (*ArgumentParser*) – command line arguments

radvel.driver.**plots**(*args*)

Generate plots

Parameters
args (*ArgumentParser*) – command line arguments

radvel.driver.**report**(*args*)

Generate summary report

Parameters
args (*ArgumentParser*) – command line arguments

radvel.driver.**save_status**(*statfile*, *section*, *statevars*)

Save pipeline status

Parameters

- **statfile** (*string*) – name of output file
- **section** (*string*) – name of section to write
- **statevars** (*dict*) – dictionary of all options to populate the specified section

radvel.driver.**tables**(*args*)

Generate TeX code for tables in summary report

Parameters
args (*ArgumentParser*) – command line arguments

4.11 Utility Functions

radvel.utils.**Msini**(*K*, *P*, *Mstar*, *e*, *Msini_units='earth'*)

Calculate Msini

Calculate Msini for a given K, P, stellar mass, and e

Parameters

- **array** (*K (float or*) – Doppler semi-amplitude [m/s]
- **P** (*float or array*) – Orbital period [days]

- **Mstar** (*float or array*) – Mass of star [Msun]
- **e** (*float or array*) – eccentricity
- **Msini_units** (*Optional[str]*) – Units of Msini {'earth','jupiter'} default: 'earth'

Returns
Msini [units = Msini_units]

Return type
float or array

radvel.utils.**bintels**(*t*, *vel*, *err*, *telvec*, *binsize=0.5*)

Bin velocities by instrument

Bin RV data with bins of with binsize in the units of t. Will not bin data from different telescopes together since there may be offsets between them.

Parameters

- **t** (*array*) – array of timestamps
- **vel** (*array*) – array of velocities
- **err** (*array*) – array of velocity uncertainties
- **telvec** (*array*) – array of strings corresponding to the instrument name for each velocity
- **binsize** (*float*) – (optional) width of bin in units of t (default=1/2.)

Returns
(bin centers, binned measurements, binned uncertainties, binned instrument codes)

Return type
tuple

radvel.utils.**date2jd**(*date*)

Convert datetime object to JD"

Parameters
date (*datetime.datetime*) – date to convert

Returns
Julian date

Return type
float

radvel.utils.**density**(*mass*, *radius*, *MR_units='earth'*)

Compute density from mass and radius

Parameters

- **mass** (*float*) – mass [MR_units]
- **radius** (*float*) – radius [MR_units]
- **MR_units** (*string*) – (optional) units of mass and radius. Must be 'earth', or 'jupiter' (default 'earth').

Returns
density in g/cc

Return type
float

radvel.utils.**draw_models_from_chain**(*mod*, *chain*, *t*, *nsamples=50*)

Draw Models from Chain

Given an MCMC chain of parameters, draw representative parameters and synthesize models.

Parameters

- **mod** (*radvel.RVmodel*) – RV model
- **chain** (*DataFrame*) – pandas DataFrame with different values from MCMC chain
- **t** (*array*) – time range over which to synthesize models
- **nsamples** (*int*) – number of draws

Returns
2D array with the different models as different rows

Return type
array

radvel.utils.**fastbin**(*x*, *y*, *nbins=30*)

Fast binning

Fast binning function for equally spaced data

Parameters

- **x** (*array*) – independent variable
- **y** (*array*) – dependent variable
- **nbins** (*int*) – number of bins

Returns
(bin centers, binned measurements, binned uncertainties)

Return type
tuple

radvel.utils.**geterr**(*vec*, *angular=False*)

Calculate median, 15.9, and 84.1 percentile values for a given vector.

Parameters

- **vec** (*array*) – vector, usually an MCMC chain for one parameter
- **angular** (*bool [optioanl]*) – Is this an angular parameter? if True vec should be in radians. This will perform some checks to ensure proper boundary wrapping.

Returns
50, 15.9 and 84.1 percentiles

Return type
tuple

radvel.utils.**initialize_posterior**(*config_file*, *decorr=False*)

Initialize Posterior object

Parse a setup file and initialize the RVModel, Likelihood, Posterior and priors.

Parameters

- **config_file** (*string*) – path to config file
- **decorr** (*bool*) – (optional) decorrelate RVs against columns defined in the decorr_vars list

Returns
(object representation of config file, radvel.Posterior object)

Return type
tuple

radvel.utils.**jd2date**(*jd*)

Convert JD to datetime.datetime object

Parameters
jd (*float*) – Julian date

Returns
calendar date

Return type
datetime.datetime

radvel.utils.**load_module_from_file**(*module_name*, *module_path*)

Loads a python module from the path of the corresponding file.

Parameters

- **module_name** (*str*) – namespace where the python module will be loaded, e.g. `foo.bar`
- **module_path** (*str*) – path of the python file containing the module

Returns
A valid module object

Raises

- **ImportError** – when the module can't be loaded
- **FileNotFoundError** – when module_path doesn't exist

radvel.utils.**round_sig**(*x*, *sig=2*)

Round by significant figures :param x: number to be rounded :type x: float :param sig: (optional) number of significant figures to retain :type sig: int

Returns
x rounded to sig significant figures

Return type
float

radvel.utils.**semi_amplitude**(*Msini*, *P*, *Mtotal*, *e*, *Msini_units='jupiter'*)

Compute Doppler semi-amplitude

Parameters

- **Msini** (*float*) – mass of planet [Mjup]
- **P** (*float*) – Orbital period [days]
- **Mtotal** (*float*) – Mass of star + mass of planet [Msun]
- **e** (*float*) – eccentricity
- **Msini_units** (*Optional[str]*) – Units of Msini {'earth','jupiter'} default: 'jupiter'

Returns
Doppler semi-amplitude [m/s]

radvel.utils.**semi_major_axis**(*P*, *Mtotal*)

Semi-major axis

Kepler's third law

Parameters

- **P** (*float*) – Orbital period [days]
- **Mtotal** (*float*) – Mass [Msun]

Returns
semi-major axis in AU

Return type
float or array

radvel.utils.**sigfig**(*med*, *errlow*, *errhigh=None*)

Format values with errors into an equal number of signficant figures.

Parameters

- **med** (*float*) – median value
- **errlow** (*float*) – lower errorbar
- **errhigh** (*float*) – upper errorbar

Returns
(med,errlow,errhigh) rounded to the lowest number of significant figures

Return type
tuple

radvel.utils.**t_to_phase**(*params*, *t*, *num_planet*, *cat=False*)

Time to phase

Convert JD to orbital phase

Parameters

- **params** (*radvel.params.RVParameters*) – RV parameters object
- **t** (*array*) – JD timestamps
- **num_planet** (*int*) – Which planet's ephemeris to phase fold on
- **cat** (*bool*) – Concatenate/double the output phase array to extend from 0 to 2

Returns
orbital phase at each timestamp

Return type
array

radvel.utils.**time_print**(*tdiff*)

Print time

Helper function to print time remaining in sensible units.

Parameters
tdiff (*float*) – time in seconds

Returns
(float time, string units)

Return type
tuple

radvel.utils.**timebin**(*time*, *meas*, *meas_err*, *binsize*)

Bin in equal sized time bins

This routine bins a set of times, measurements, and measurement errors into time bins. All inputs and outputs should be floats or double. binsize should have the same units as the time array. (from Andrew Howard, ported to Python by BJ Fulton)

Parameters

- **time** (*array*) – array of times
- **meas** (*array*) – array of measurements to be comined
- **meas_err** (*array*) – array of measurement uncertainties
- **binsize** (*float*) – width of bins in same units as time array

Returns
(bin centers, binned measurements, binned uncertainties)

Return type
tuple

radvel.utils.**working_directory**(*dir*)

Do something in a directory

Function to use with *with* statements.

Parameters
dir (*string*) – name of directory to work in

Example

```
>>> with workdir('/temp'):
    # do something within the /temp directory
```

4.12 LaTeX Report

class radvel.report.**RadvelReport**(*planet*, *post*, *chains*, *minafactor*, *maxarchange*, *maxgr*, *mintz*, *compstats=None*, *derived=False*)

Radvel report

Class to handle the creation of the radvel summary PDF

Parameters

- **planet** (*planet object*) – planet configuration object loaded in *kepfit.py* using *imp.load_source*
- **post** (*radvel.posterior*) –

  radvel.posterior object containing the best-fit parameters in
  post.params
- **compstats** (*dict*) – dictionary of model comparison results from *radvel ic*

- **derived** (*bool*) – included table of derived parameters
- **chains** (*DataFrame*) – output DataFrame from a *radvel.mcmc* run
- **criterion** (*DataFrame*) – output DataFrame from a 'radvel.mcmc' run

compile(*pdfname*, *latex_compiler='pdflatex'*, *depfiles=[]*)

Compile radvel report

Compile the radvel report from a string containing TeX code and save the resulting PDF to a file.

Parameters

- **pdfname** (*str*) – name of the output PDF file
- **latex_compiler** (*str*) – path to latex compiler
- **depfiles** (*list*) – list of file names of dependencies needed for LaTex compilation (e.g. figure files)

texdoc()

TeX for entire document

Returns
TeX code for report

Return type
str

class radvel.report.**TexTable**(*report*)

LaTeX table

Class to handle generation of the LaTeX tables within the summary PDF.

Parameters

- **report** (radvel.report.RadvelReport) – radvel report object
- **full** (*bool*) – get full-length RV table [default: True]

tab_comparison()

Model comparisons

tab_crit(*name_in_title=False*)

Table of final convergence criterion values :param name_in_title: if True, include

the name of the star in the table title

tab_derived(*name_in_title=False*)

Table of derived parameter values :param name_in_title: if True, include

the name of the star in the table title

tab_params(*name_in_title=False*)

Table of final parameter values :param name_in_title: if True, include

the name of the star in the table title

tab_prior_summary(*name_in_title=False*)

Summary of priors

Parameters
name_in_title (*Bool [optional]*) – if True, include the name of the star in the table title

tab_rv(*name_in_title=False*, *max_lines=50*)

Table of input velocities

Parameters
name_in_title (*Bool [optional]*) – if True, include the name of the star in the table title

4.13 Plotting

4.13.1 MCMC Plots

class radvel.plot.mcmc_plots.**AutoPlot**(*auto*, *saveplot=None*)

Class to handle the creation of an autocorrelation time plot from output autocorrelation times.

Parameters

- **auto** (*DataFrame*) – Autocorrelation times output by radvel.mcmc
- **saveplot** (*str, optional*) – Name of output file, will show as interactive matplotlib window if not defined.

plot()

Make and either save or display the autocorrelation plot

class radvel.plot.mcmc_plots.**CornerPlot**(*post*, *chains*, *saveplot=None*)

Class to handle the creation of a corner plot from output MCMC chains and a posterior object.

Parameters

- **post** (*radvel.Posterior*) – radvel posterior object
- **chains** (*DataFrame*) – MCMC chains output by radvel.mcmc
- **saveplot** (*str, optional*) – Name of output file, will show as interactive matplotlib window if not defined.

plot()

Make and either save or display the corner plot

class radvel.plot.mcmc_plots.**DerivedPlot**(*chains*, *P*, *saveplot=None*)

Class to handle the creation of a corner plot of derived parameters from output MCMC chains and a posterior object.

Parameters

- **chains** (*DataFrame*) – MCMC chains output by radvel.mcmc
- **P** – object representation of config file
- **(Optional[string]** (*saveplot*) – Name of output file, will show as interactive matplotlib window if not defined.

plot()

Make and either save or display the corner plot

class radvel.plot.mcmc_plots.**TrendPlot**(*post*, *chains*, *nwalkers*, *nensembles*, *outfile=None*)

Class to handle the creation of a trend plot to show the evolution of the MCMC as a function of step number.

Parameters

- **post** (*radvel.Posterior*) – Radvel Posterior object
- **chains** (*DataFrame*) – MCMC chains output by radvel.mcmc
- **nwalkers** (*int*) – number of walkers used in this particular MCMC run
- **outfile** (*string [optional]*) – name of output multi-page PDF file

plot()

Make and save the trend plot as PDF

radvel.plot.mcmc_plots.**texlabel**(*key*, *letter*)

Parameters

- **key** (*list of string*) – list of parameter strings
- **letter** (*string*) – planet letter

Returns
LaTeX label for parameter string

Return type
string

4.13.2 Orbit Plots

class radvel.plot.orbit_plots.**GPMultipanelPlot**(*post*, *saveplot=None*, *epoch=2450000*, *yscale_auto=False*, *yscale_sigma=3.0*, *phase_nrows=None*, *phase_ncols=None*, *uparams=None*, *rv_phase_space=0.08*, *telfmts={}*, *legend=True*, *phase_limits=[]*, *nobin=False*, *phasetext_size='large'*, *figwidth=7.5*, *fit_linewidth=2.0*, *set_xlim=None*, *text_size=9*, *legend_kwargs={'loc': 'best'}*, *subtract_gp_mean_model=False*, *plot_likelihoods_separately=False*, *subtract_orbit_model=False*, *status=None*, *separate_orbit_gp=False*)

Class to handle the creation of RV multipanel plots for posteriors fitted using Gaussian Processes.

Takes the same args as MultipanelPlot, with a few additional bells and whistles…

Parameters

- **subtract_gp_mean_model** (*bool, optional*) – if True, subtract the Gaussian process mean max likelihood model from the data and the model when plotting the results. Default: False.
- **plot_likelihoods_separately** (*bool, optional*) – if True, plot a separate panel for each Likelihood object. Default: False
- **subtract_orbit_model** (*bool, optional*) – if True, subtract the best-fit orbit model from the data and the model when plotting the results. Useful for seeing the structure of correlated noise in the data. Default: False.

- **status** (*ConfigParser*) – (optional) result of radvel.driver.load_status on the .stat status file

plot_gp_like(*like*, *orbit_model4data*, *ci*)

Plot a single Gaussian Process Likleihood object in the current Axes, including Gaussian Process uncertainty bands.

Parameters

- **like** (*radvel.GPLikelihood*) – radvel.GPLikelihood object. The model plotted will be generated from *like.params*.
- **orbit_model4data** (*numpy array*) –
- **ci** (*int*) – index to use when choosing a color to plot from radvel.plot.default_colors. This is only used if the Likelihood object being plotted is not in the list of defaults. Increments by 1 if it is used.

Returns: current (possibly changed) value of the input *ci*

plot_multipanel(*nophase=False*)

Provision and plot an RV multipanel plot for a Posterior object containing one or more Gaussian Process Likelihood objects.

Parameters
nophase (*bool, optional*) – if True, don't include phase plots. Default: False.

Returns

- current matplotlib Figure object
- list of Axes objects

Return type
tuple containing

plot_timeseries()

Make a plot of the RV data and Gaussian Process + orbit model in the current Axes.

class radvel.plot.orbit_plots.**MultipanelPlot**(*post*, *saveplot=None*, *epoch=2450000*, *yscale_auto=False*, *yscale_sigma=3.0*, *phase_nrows=None*, *phase_ncols=None*, *uparams=None*, *telfmts={}*, *legend=True*, *phase_limits=[]*, *nobin=False*, *phasetext_size='large'*, *rv_phase_space=0.08*, *figwidth=7.5*, *fit_linewidth=2.0*, *set_xlim=None*, *text_size=9*, *highlight_last=False*, *show_rms=False*, *legend_kwargs={'loc': 'best'}*, *status=None*)

Class to handle the creation of RV multipanel plots.

Parameters

- **post** (*radvel.Posterior*) – radvel.Posterior object. The model plotted will be generated from *post.params*
- **epoch** (*int, optional*) – epoch to subtract off of all time measurements
- **yscale_auto** (*bool, optional*) – Use matplotlib auto y-axis scaling (default: False)
- **yscale_sigma** (*float, optional*) – Scale y-axis limits for all panels to be +/- yscale_sigma*(RMS of data plotted) if yscale_auto==False
- **phase_nrows** (*int, optional*) – number of columns in the phase folded plots. Default is nplanets.

- **phase_ncols** (*int, optional*) – number of columns in the phase folded plots. Default is 1.
- **uparams** (*dict, optional*) – parameter uncertainties, must contain 'per', 'k', and 'e' keys.
- **telfmts** (*dict, optional*) – dictionary of dictionaries mapping instrument suffix to plotting format code. Example:

  telfmts = {
  'hires': dict(fmt='o',label='HIRES'), 'harps-n' dict(fmt='s')

  }
- **legend** (*bool, optional*) – include legend on plot? Default: True.
- **phase_limits** (*list, optional*) – two element list specifying pyplot.xlim bounds for phase-folded plots. Useful for partial orbits.
- **nobin** (*bool, optional*) – If True do not show binned data on phase plots. Will default to True if total number of measurements is less then 20.
- **phasetext_size** (*string, optional*) – fontsize for text in phase plots. Choice of {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}. Default: 'x-small'.
- **rv_phase_space** (*float, optional*) – amount of space to leave between orbit/residual plot and phase plots.
- **figwidth** (*float, optional*) – width of the figures to be produced. Default: 7.5 (spans a page with 0.5 in margins)
- **fit_linewidth** (*float, optional*) – linewidth to use for orbit model lines in phase-folded plots and residuals plots.
- **set_xlim** (*list of float*) – limits to use for x-axes of the timeseries and residuals plots, in JD - *epoch*. Ex: [7000., 70005.]
- **text_size** (*int*) – set matplotlib.rcParams['font.size'] (default: 9)
- **highlight_last** (*bool*) – make the most recent measurement much larger in all panels
- **show_rms** (*bool*) – show RMS of the residuals by instrument in the legend
- **legend_kwargs** (*dict*) – dict of options to pass to legend (plotted in top panel)
- **status** (*ConfigParser*) – (optional) result of radvel.driver.load_status on the .stat status file

plot_multipanel(*nophase=False*, *letter_labels=True*)

Provision and plot an RV multipanel plot

Parameters

- **nophase** (*bool, optional*) – if True, don't include phase plots. Default: False.
- **letter_labels** (*bool, optional*) – if True, include letter labels on orbit and residual plots. Default: True.

Returns

- current matplotlib Figure object
- list of Axes objects

Return type
tuple containing

plot_phasefold(*pltletter*, *pnum*)

Plot phased orbit plots for each planet in the fit.

Parameters

- **pltletter** (*int*) – integer representation of letter to be printed in the corner of the first phase plot. Ex: ord("a") gives 97, so the input should be 97.
- **pnum** (*int*) – the number of the planet to be plotted. Must be the same as the number used to define a planet's Parameter objects (e.g. 'per1' is for planet #1)

plot_residuals()

Make a plot of residuals in the current Axes.

plot_timeseries()

Make a plot of the RV data and model in the current Axes.

CHAPTER

FIVE

INDICES AND TABLES FOR PYTHON CODE

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

P

Q

R

S

T

U

V

W