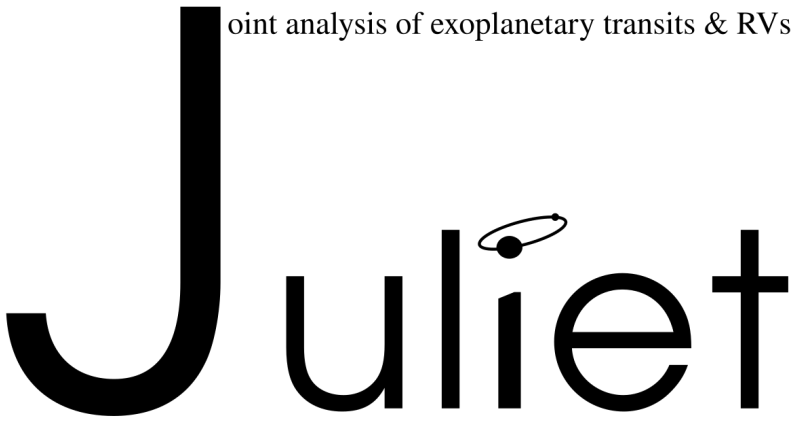
juliet

Mar 20, 2021

User Guide



`juliet` is a versatile modelling tool for transiting and non-transiting exoplanetary systems that allows to perform quick-and-easy fits to data coming from transit photometry, radial velocity or both using bayesian inference and, in particular, using Nested Sampling in order to allow both efficient fitting and proper model comparison.

In this documentation you'll be able to check out the features `juliet` can offer for your research, which range from fitting different datasets simultaneously for both transits and radial-velocities to accounting for systematic trends both using linear models or Gaussian Processes (GP), to even extract information from photometry alone (e.g., stellar rotation periods) with just a few lines of code.

`juliet` builds on the work of "giants" that have made publicly available tools for transit (batman, starry), radial-velocity (radvel), GP modelling (george, celerite) and Nested Samplers (*MultiNest* via pymultinest, dynesty, ultranest) and thus can be seen as a wrapper of all of those in one. Somewhat like an Infinity Gauntlet for exoplanets.

The library is in active development in its public repository on GitHub. If you discover any bugs or have requests for us, please consider sending us an email or opening an issue.

CHAPTER 1

Installation

1.1 Installing via pip

`juliet` can be easily installed using pip:

```
pip install juliet
```

The core of `juliet` is comprised of the transit (batman, starry), radial-velocity (radvel) and Gaussian Process (george, celerite) modelling tools, as well as of the Nested Sampling algorithms (*MultiNest* via pymultinest, dynesty) that it uses. However, **by default the ‘‘juliet‘‘ installation will force ‘dynesty‘ as the main sampler to be installed, and will not install ‘pymultinest‘**. This is because the `pymultinest` installation can involve a couple of extra steps, which we really recommend following, as `pymultinest` might be faster for problems involving less than about 20 free parameters (see below).

1.2 Installing from source

The source code for `juliet` can be downloaded from GitHub by running

```
git clone https://github.com/nespinoza/juliet.git
```

Once cloned, simply enter the `juliet` folder and do

```
python setup.py install
```

To install the latest version of the code.

1.3 Installing pymultinest

As described above, we really recommend installyng `pymultinest`. The full instructions on how to install pymultinest can be found in the project’s documentation. We repeat here the main steps. First, install it via pip:

```
pip install pymultinest
```

Then, you need to build and compile *MultiNest*. For this, do:

```
git clone https://github.com/JohannesBuchner/MultiNest
cd MultiNest/build
cmake ..
make
```

This will create a file `libmultinest.so` or `libmultinest.dylib` under `MultiNest/lib`: that is the one that will allow us to use `pymultinest`. To not move that file around in your system, you can include the `MultiNest/lib` folder in your `LD_LIBRARY_PATH` (e.g., in your `~/.bash_profile` or `~/.bashrc` file). In my case, the library is under `/Users/nespinoza/github/MultiNest/lib`, so I added the following line to my `~/.bash_profile` file:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/Users/nespinoza/github/MultiNest/lib
```

Dependencies

The above installation instructuins for `juliet` assume you have a Python installation. `juliet`, in turn, depends on the following libraries/packages, all of which will be installed automatically if you follow the instructions above:

1. NumPy,
2. SciPy,
3. batman,
4. radvel,
5. george,
6. celerite,
7. dynesty,
8. pymultinest (optional),
9. matplotlib (optional), and
10. seaborn (optional).

The last are only needed for certain plotting functions within `juliet`. The `pymultinest` installation is optional, but highly recommended.

CHAPTER 2

Getting started

2.1 Two ways of using juliet

In the spirit of accomodating the code for everyone to use, `juliet` can be used in two different ways: as an **imported library** and also in **command line mode**. Both give rise to the same results because the command line mode simply calls the `juliet` libraries in a python script.

To use `juliet` as an **imported library**, inside any python script you can simply do:

```
import juliet
dataset = juliet.load(priors = priors, t_lc=times, y_lc=flux, yerr_lc=flux_error)
results = dataset.fit()
```

In this example, `juliet` will perform a fit on a lightcurve dataset defined by a dictionary of times `times`, relative fluxes `flux` and error on those fluxes `flux_error` given some prior information `priors` which, as we will see below, is also defined through a dictionary.

In **command line mode**, `juliet` can be used through a simple call in any terminal. To do this, after installing juliet, you can from anywhere in your system simply do:

```
juliet -flag1 -flag2 --flag3
```

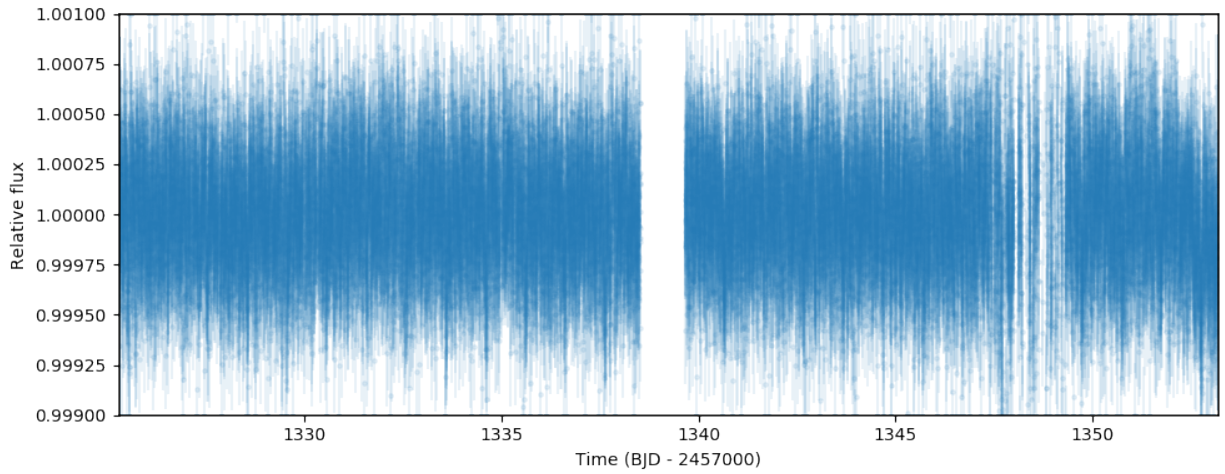
In this example, `juliet` is performing a fit using different inputs defined by `-flag1`, `-flag2` and `--flag3`. There are several flags that can be used to accomodate your `juliet` runs through command-line which we'll explore in the tutorials. There is a third way of using `juliet`, which is by calling the `juliet.py` code and applying these same flags (as it is currently explained in project's wiki page). However, no further updates will be done for that method, and the ones defined above should be the preferred ones to use.

2.2 A first fit to data with juliet

To showcase how `juliet` works, let us first perform an extremely simple fit to data using `juliet` as an *imported library*. We will fit the TESS data of TOI-141 b, which was shown to host a 1-day transiting exoplanet by Espinoza et

al. (2019). Let us first load the data corresponding to this object, which is hosted in MAST. For TESS data, `juliet` has already built-in functions to load the data arrays directly given a web link to the data — let's load it and plot the data to see how it looks:

```
import juliet
import numpy as np
# First, get times, normalized-fluxes and errors for TOI-141 from MAST:
t,f,ferr  = juliet.get_TESS_data('https://archive.stsci.edu/hlsps/tess-data-alerts/'+\
                                 'hlsp_tess-data-alerts_tess_phot_00403224672-'+\
                                 's01_tess_v1_lc.fits')

# Plot the data!
import matplotlib.pyplot as plt
plt.errorbar(t,f,yerr=ferr,fmt='.')

plt.xlim([np.min(t),np.max(t)])
plt.ylim([0.999,1.001])
plt.xlabel('Time (BJD - 2457000)')
plt.ylabel('Relative flux')
```



This will save arrays of times, fluxes (`PDCSAP_FLUX` fluxes, in particular) and errors on the `t`, `f` and `ferr` arrays. Now, in order to load this dataset into a format that `juliet` likes, we need to put these into dictionaries. This, as we will see, will make it extremely easy to add data from more instruments, as these will be simply stored in different keys of the same dictionary. For now, let us just use this TESS data; we put them in dictionaries that `juliet` likes as follows:

```
# Create dictionaries:
times, fluxes, fluxes_error = {},{},{}
# Save data into those dictionaries:
times['TESS'], fluxes['TESS'], fluxes_error['TESS'] = t,f,ferr
# If you had data from other instruments you would simply do, e.g.,
# times['K2'], fluxes['K2'], fluxes_error['K2'] = t_k2,f_k2,ferr_k2
```

The final step to fit the data with `juliet` is to define the priors for the different parameters that we are going to fit. This can be done in two ways. The longest (but more jupyter-notebook-friendly?) is to create a dictionary that, on each key, has the names of the parameter to be fitted. Each of those elements will be dictionaries themselves, containing the `distribution` of the parameter and their corresponding `hyperparameters` (for details on what distributions `juliet` can handle, what are the hyperparameters and what each parameter name mean, see the next section of this document: *Models, priors and outputs*).

Let us give normal priors for the period P_p1, time-of-transit center t0_p1, mean out-of-transit flux mflux_TESS, uniform distributions for the parameters r1_p1 and r2_p1 of the Espinoza (2018) parametrization for the impact parameter and planet-to-star radius ratio, same for the q1_p1 and q2_p1 Kipping (2013) limb-darkening parametrization (juliet assumes a quadratic limb-darkening by default — other laws can be easily defined, as it will be shown in the tutorials), log-uniform distributions for the stellar density rho (in kg/m3) and jitter term sigma_w_TESS (in parts-per-million, ppm), and leave the rest of the parameters (eccentricity ecc_p1, argument of periastron (in degrees) omega_p1 and dilution factor mdilution_TESS) fixed:

```
priors = {}

# Name of the parameters to be fit:
params = ['P_p1','t0_p1','r1_p1','r2_p1','q1_TESS','q2_TESS','ecc_p1','omega_p1',\
             'rho', 'mdilution_TESS', 'mflux_TESS', 'sigma_w_TESS']

# Distribution for each of the parameters:
dists = ['normal','normal','uniform','uniform','uniform','uniform','fixed','fixed',\
                'loguniform', 'fixed', 'normal', 'loguniform']

# Hyperparameters of the distributions (mean and standard-deviation for normal
# distributions, lower and upper limits for uniform and loguniform distributions, and
# fixed values for fixed "distributions", which assume the parameter is fixed)
hyperps = [[1.,0.1], [1325.55,0.1], [0.,1], [0.,1.], [0., 1.], [0., 1.], 0.0, 90.,\
                  [100., 10000.], 1.0, [0.,0.1], [0.1, 1000.]]

# Populate the priors dictionary:
for param, dist, hyperp in zip(params, dists, hyperps):
    priors[param] = {}
    priors[param]['distribution'], priors[param]['hyperparameters'] = dist, hyperp
```

With these definitions, to fit this dataset with juliet one would simply do:

```
# Load dataset into juliet, save results to a temporary folder called toi141_fit:
dataset = juliet.load(priors=priors, t_lc = times, y_lc = fluxes, \
                      yerr_lc = fluxes_error, out_folder = 'toi141_fit')

# Fit and absorb results into a juliet.fit object:
results = dataset.fit(n_live_points = 300)
```

This code will run juliet and save the results both to the results object and to the toi141_fit folder.

The second way to define the priors for juliet (and perhaps the most simple) is to create a text file where in the first column one defines the parameter name, in the second column the name of the distribution and in the third column the hyperparameters. The priors defined above would look like this in a text file:

```
P_p1                normal              1.0,0.1
t0_p1               normal              1325.55,0.1
r1_p1               uniform             0.0,1.0
r2_p1               uniform             0.0,1.0
q1_TESS             uniform             0.0,1.0
q2_TESS             uniform             0.0,1.0
ecc_p1              fixed               0.0
omega_p1            fixed               90.0
rho                 loguniform          100.0,10000.0
mdilution_TESS      fixed               1.0
mflux_TESS          normal              0.0,0.1
sigma_w_TESS        loguniform          0.1,1000.0
```

To run the same fit as above, suppose this prior file is saved under toi141_fit/priors.dat. Then, to load this

dataset into `juliet` and fit it, one would do:

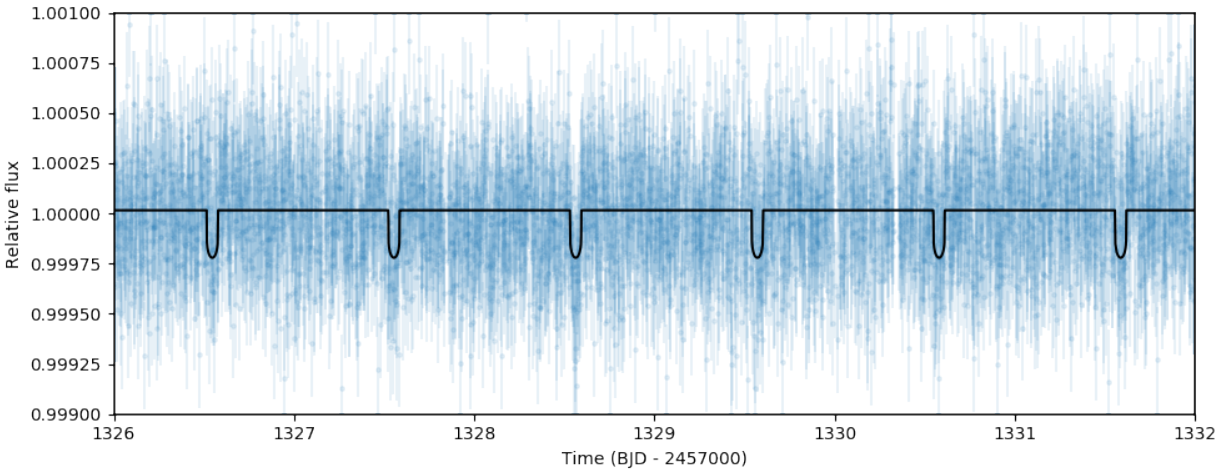
```
# Load dataset into juliet, save results to a temporary folder called toi141_fit:
dataset = juliet.load(priors='toi141_fit/priors.dat', t_lc = times, y_lc = fluxes, \
                      yerr_lc = fluxes_error, out_folder = 'toi141_fit')

# Fit and absorb results into a juliet.fit object:
results = dataset.fit(n_live_points = 300)
```

And that's it! Cool `juliet` fact is that, once you have defined an `out_folder`, **all your data will be saved there — not only the prior file and the results of the fit, but also the photometry or radial-velocity you fed into juliet will be saved**. This makes it easy to come back later to this dataset without having to download the data all over again, or re-run your fits. So, for example, suppose we have already ran the code above, closed our terminals, and wanted to come back at this dataset again with another `python` session and say, plot the data and best-fit model. To do this one can simply do:

```
import juliet

# Load already saved dataset with juliet:
dataset = juliet.load(input_folder = 'toi141_fit', out_folder = 'toi141_fit')

# Load results (the data.fit call will recognize the juliet output files in
# the toi141_fit folder generated when we ran the code for the first time):
results = dataset.fit()

import matplotlib.pyplot as plt

# Plot the data:
plt.errorbar(dataset.times_lc['TESS'], dataset.data_lc['TESS'], \
             yerr = dataset.errors_lc['TESS'], fmt = '.', alpha = 0.1)

# Plot the model:
plt.plot(dataset.times_lc['TESS'], results.lc.evaluate('TESS'))

# Plot portion of the lightcurve, axes, etc.:
plt.xlim([1326,1332])
plt.ylim([0.999,1.001])
plt.xlabel('Time (BJD - 2457000)')
plt.ylabel('Relative flux')
plt.show()
```

Which will give us a nice plot of the data and the `juliet` fit:

Warning: When using MultiNest, make sure that the `out_folder` full path is less than 69 characters long. This is because MultiNest internally has a character limit for the full output path of 100 characters (see this fun discussion). Because the largest MultiNest output `juliet` produces (produced by MultiNest itself) is called `jomnest_post_equal_weights.dat`, which has 30 characters, this leaves the possible total character length of the folder to be 69 characters not counting the backlash at the end. Bottom line: when using MultiNest, stick to small `out_folder` lengths.


1.00100
1.00075
1.00050
1.00025
1.00000
0.99975
0.99950
0.99925
0.99900
Relative flux
1326
1327
1328
1329
1330
1331
1332
Time (BJD - 2457000)

CHAPTER 3

Models, priors and outputs

As it was shown in the *Getting started* section, a typical `juliet` run will make use of two objects which form the core of the library: the `load` object and the `fit` object. The former is used to load a dataset, and the second is used to fit that dataset using the samplers supported within `juliet`, which in turn spits out the result of the fit including posterior distributions, fits, etc. In general, a dataset can be loaded to `juliet` by the simple call

```
import juliet
dataset = juliet.load(priors=priors, t_lc = times, y_lc = fluxes, \
                      yerr_lc = fluxes_error, t_rv = rvtimes, y_rv = rvs, \
                      yerr_rv = rv_errs, out_folder = yourfolder)
```

Here `times`, `fluxes` and `fluxes_error` are dictionaries containing the lightcurve data and `rvtimes`, `rvs` and `rv_errs` contain the radial-velocity data, where each key should have the instrument name and under each of those instruments an array should be given with the corresponding data. Alternatively, you might give paths to files that contain your data given they have times in the first column, data in the second, errors in the data in the third column and instrument names in the fourth via the `lcfilename` and `rvfilename` options (e.g., `juliet.load(..., lcfilename = path_to_lc)`).

The `priors` variable, on the other hand, is either a dictionary or a filename containing the prior distribution information for each parameter in the model (see below) and `yourfolder` is a user-defined folder that is used to save the results (and the data!).

Once a `dataset` is loaded it can be fit by doing `dataset.fit()`. The options of the fit can of course be modified — we refer the users to the API on this documentation for details on this front.

3.1 Exoplanets with juliet, pt. I: planetary parameters

To date, `juliet` is able to model transit and radial-velocities (RVs), each of which have their own set of parameters. We have divided the types of parameters into what we call the **planetary parameters** and the **instrument parameters**. Within `juliet`, the former set of parameters are always of the form `parameter_pN`, where `N` is a number identifier for a given planet (yes, `juliet` handles multiple-planet systems!). The instrument parameters, on the other hand, are always of the form `parameter_instrument`, where `instrument` is an instrument name.

The (basic) **planetary parameters** currently supported by `juliet` are:

Parameter name	Description
P_p1	The planetary period of the planet under study (days).
t0_p1	The time-of-transit center of the planet under study (days).
p_p1	Planet-to-star radius ratio (Rp/Rs).
b_p1	Impact parameter of the orbit.
a_p1	Scaled semi-major axis of the orbit (a/R*).
ecc_p1	Eccentricity of the orbit.
omega_p1	Argument of periastron passage of the orbit (in degrees).
K_p1	RV semi-amplitude of the orbit of the planet (same units as RV data).

Within `juliet`, it is very important that the periods of the planets are in chronological order, i.e., that `P_p1 < P_p2 < ....`. This is to avoid solutions in which the periods of the planets can be exchanged between the variables. When fitting for transit data, all of the above but `K` have to be defined for each planet. When fitting radial-velocities, only `P`, `t0`, `ecc`, `omega` and `K` have to be defined. When fitting both, all of these have to be defined.

Although the above are the basic planetary parameters allowed by `juliet`, the library allows to perform three more advanced and efficient parametrizations for some of its parameters:

- **The first** is the one proposed by Espinoza (2018), in which instead of fitting for `p` and `b`, one fits for the parameters `r1` and `r2` which, if sampled with uniform priors between 0 and 1, are able to allow only physically plausible values for `p` and `b` (i.e., `b < 1 + p`). This parametrization needs one to define the smallest planet-to-star radius ratio to be considered, `pl` and the maximum planet-to-star radius ratio to be considered, `pu`. For a coarse search, one could set `pl` to zero and `pu` to 1 — these are the default values within `juliet`.
- **The second parametrization** allowed by `juliet` is to define a prior for the stellar density, `rho` (in kg/m^3) instead of the scaled semi-major axis of the planets, `a`. This is useful because setting this for a system, using Kepler's third law one can recover `a` for each planet using only the period, `P`, which is a mandatory parameter for any `juliet` run. In this way, instead of fitting for `a` for different planetary systems, a single value of `rho` can be defined for the system.
- **The third parametrization** has to do with the eccentricity and the argument of periastron. `juliet` allows either to (1) fit for them directly (via the `ecc` and `omega` parameters), (2) to fit for the parameters `esinomega = ecc*sin(omega*pi/180)` and `ecosomega = ecc*cos(omega*pi/180)` or (3) to fit for the parameters `sesinomega = sqrt(ecc)*sin(omega*pi/180)` and `secosomega = sqrt(ecc)*cos(omega*pi/180)`. The latter two are typically defined between -1 and 1, and within `juliet` it is always ensured that the eccentricity is smaller than 1.

Finally, for RVs there are three additional "planetary parameters" that can be passed, which are helpful to model long-period planets for which no full cycles have been observed in the data yet. These are the `rv_intercept`, `rv_slope` and `rv_quad`. These fit a long-term trend to the RVs which is added to the Keplerian model and is of the form `rv_intercept + (t-ta)*rv_slope + (t-ta)**2*rv_quad`. `ta` is an arbitrary time, which within `juliet` is defined to be `2458460` — this arbitrary time can of course be changed by the user. To do it, when fitting a `dataset` simply do `dataset.fit(..., ta = yourdate)`.

3.2 Exoplanets with juliet, pt. II: instrumental parameters

The **instrument parameters** currently supported by `juliet` are:

Parameter name	Description
mdilution_instrument	The dilution factor for the photometric *instrument*.
mflux_instrument	The offset relative flux for the photometric *instrument*.
sigma_w_instrument	A jitter (in ppm or RV units) added in quadrature to the errorbars of *instrument*.
q1_instrument	Limb-darkening parametrization for photometric *instrument*.
q2_instrument	Limb-darkening parametrization for photometric *instrument*.
mu_instrument	Systemic radial-velocity for a radial-velocity *instrument* (same units as data).

Here, q1 and q2 are the limb-darkening parametrizations of Kipping (2013) for two-parameter limb-darkening laws for all laws except for the logarithmic, where they correspond to the transformations in Espinoza & Jordan (2016). If the linear law is to be used, the user has to only define q1 which will be interpreted directly as the limb-darkening coefficient of the linear law. For juliet analyses only using photometry, mdilution, mflux, sigma_w, q1 and q2 have to defined. For analyses only using radial-velocity measurements, mu and sigma_w have to be defined. All must be defined in the case of joint fits.

By default, the limb-darkening law assumed in the fits for all instruments is the quadratic law. However, one can define different limb-darkening laws for different instruments passing a string to the ld_laws input of the juliet.load() object, where the string defines the names and laws to be used for each instrument separated by commas (e.g., juliet.load(...,ld_laws = 'TESS-quadratic,K2-logarithmic,LCOGT-linear'). Limb-darkening coefficients and dilution factors can be common within instruments, too. To force this, simply give all the instruments that should be common to different instruments separated by underscores when passing the *priors* (see below) to juliet, e.g., q1_TESS_K2.

Warning: Because in *juliet* the internal parameters include underscores (_), the instrument names **should not** contain underscores. In this way, for example, instead of naming your instrument *My_Instrument* (as in, e.g., *mdilution_My_instrument*), prefer *My-Instrument* or *MyInstrument* instead.

3.3 Exoplanets with juliet, pt. III: linear models & gaussian processes

There are additional instrument parameters that can be given to *juliet* to account for linear models in the data and/or gaussian-processes. For *linear models*, it is assumed each linear regressor X of instrument instrument will be weighted by a parameter thetaX_instrument. There is no limit to the number of linear terms a given instrument can have, and the linear regressors can either be given directly as a dictionary through the juliet.load call (through the linear_regressors_lc input for lightcurve linear regressors and/or the linear_regressors_rv input for linear regressors for the radial-velocities), or as extra columns in any input lightcurve or radial-velocity file the user is giving as input to that same call. For details, check out the *Incorporating linear models* tutorial.

For *Gaussian Processes* (GPs), the regressors can be given in a similar manner as for linear regressors when doing the juliet.load call (i.e., via the analogous GP_regressors_lc and GP_regressors_rv inputs). Alternatively, the name of a file which contains the different regressors on each column with the last column being the instrument name can be given through the same juliet.load call using the GPlceparamfile for the file defining the GP regressors for the lightcurves and GPrveparamfile for the file defining the GP regressors for the radial-velocities.

juliet automatically identifies which kernel the user wants to use for each instrument depending on the name of the GP hyperparameters in the *priors*. For instrument-by-instrument models (i.e., GP regressions which are individual to each instrument) the parameter names follow the pname_instrument form, where pname is any of the parameter names listed below and instrument is a given instrument (e.g., GP_sigma_TESS). For so-called "global" models, which are models that are *not* instrument-specific (for more details on the difference between those types of models, check the juliet paper and/or the *Incorporating Gaussian Processes* tutorial), the parameter names follow the pname_lc form for global lightcurve models, and pname_rv for radial-velocity global models.

Below we list the GP kernels implemented so far within *juliet*. More kernels can be implemented upon request and/or via git push to the *juliet* repository — again, for usage details, please check out the *Incorporating Gaussian Processes* tutorial:

Multi-dimensional squared-exponential kernel

Hyperparameters	Description
GP_sigma	Amplitude of the GP (in ppm for the photometry, units of measurements for RVs)
GP_alpha0	Inverse (squared) length-scale/normalized amplitude of the first external parameter
GP_alpha1	Inverse (squared) length-scale/normalized amplitude of the second external parameter
...	...
GP_alphan	Inverse (squared) length-scale/normalized amplitude of the n+1 external parameter

Exp-sine-squared kernel

Hyperparameters	Description
GP_sigma	Amplitude of the GP (in ppm for the photometry, units of measurements for RVs)
GP_alpha	Inverse (squared) length-scale of the external parameter
GP_Gamma	Amplitude of the sine-part of the kernel
GP_Prot	Period of the quasi-periodic kernel

celerite quasi-periodic kernel

Hyperparameters	Description
GP_B	Amplitude of the GP (in ppm for the photometry, units of measurements for RVs)
GP_C	Additive factor impacting on the amplitude of the GP
GP_L	Length-scale of exponential part of the GP
GP_Prot	Period of the quasi-periodic GP

celerite Simple Harmonic Oscillator (SHO) kernel

Hyperparameters	Description
GP_S0	Characteristic power of the SHO
GP_omega0	Characteristic frequency of the SHO
GP_Q	Quality factor of the SHO

celerite (approximate) Matern kernel

Hyperparameters	Description
GP_sigma	Amplitude of the GP (in ppm for the photometry, units of measurements for RVs)
GP_rho	Time/length-scale of the GP

celerite exponential kernel

Hyperparameters	Description
GP_sigma	Amplitude of the GP (in ppm for the photometry, units of measurements for RVs)
GP_timescale	Time/length-scale of the GP

celerite (approximate) Matern multiplied by exponential kernel

Hyperparameters	Description
GP_sigma	Amplitude of the GP (in ppm for the photometry, units of measurements for RVs)
GP_rho	Time/length-scale of the Matern part of the GP
GP_timescale	Time/length-scale of the exponential part of the GP

3.4 Priors

As introduced at the beggining, a set of priors can be defined for the parameters under consideration via the `priors` variable, which can be either a filename containing a file with the priors as was done in the *Getting started* section, or a dictionary, as was also done in that section. Currently, *juliet* supports the following prior distributions to be defined for the parameters:

Distribution	Description	Hyperparameters
Uniform	A uniform distribution defined between a lower (a) and upper (b) limit.	a,b
Normal	A normal distribution defined by its mean mu and standard-deviation sigma.	mu, sigma
TruncatedNormal	A normal distribution defined by its mean mu and standard-deviation sigma, along with a lower (a) and upper (b) limit defining its support.	mu, sigma, a,b
Jeffreys or Loguniform	A log-uniform distribution defined between a lower (a) and upper (b) limit.	a,b
Beta	A beta distribution having support between 0 and 1 defined by its alpha and beta parameters.	alpha, beta

Note that the hyperparameters have to be passed on the order defined above in the prior file or dictionary. Further distributions can be made available for *juliet* upon request, as they are extremely easy to implement. If a parameter wants to be fixed to a known value, then the prior distribution can be set to *FIXED*.

3.5 Outputs

Once a `juliet` fit is ran (e.g., `results = dataset.fit()`), this will generate a `juliet.fit` object which has several features the user can explore. The most important is the `juliet.fit.posteriors` dictionary, which contains three important keys: `posterior_samples`, which is a dictionary having the posterior samples for all the fitted parameters, `lnZ`, which has the log-evidence for the current fit and `lnZerr` which has the error on the log-evidence. This same dictionary is also automatically saved to the output folder if there was one defined by the user as a .pkl file.

In addition, a file called `posteriors.dat` file is also printed out if an output folder is given, which is of the form

```
# Parameter Name                Median                  Upper 68 CI                 Lower 68 CI
q2_TESS                         0.4072409698            0.3509391055                0.2793487941
P_p1                            1.0079166018            0.0000827690                0.0000545234
a_p1                            4.5224665335            0.5972474545                1.3392152148
q1_TESS                         0.2178116586            0.2583946746                0.1424332922
```

(continues on next page)

(continued from previous page)

```
r2_p1                          0.0146632299          0.0008468341          0.
↪0006147659
p_p1                           0.0146632299          0.0008468341          0.
↪0006147659
b_p1                           0.5122384103          0.2961574900          0.
↪3206523210
inc_p1                         83.5179400288         4.3439922509          8.
↪1734713106
mflux_TESS                     -0.0000154812         0.0000021394          0.
↪0000020902
rho                            1722.5385338667       776.2573107345        1121.
↪9672108451
t0_p1                          1325.5386166342       0.0008056050          0.
↪0012949209
r1_p1                          0.6748256069          0.1974383267          0.
↪2137682140
sigma_w_TESS                   127.3813413245        3.6857084428          3.
↪3647860049
```

This contains on the first column the parameter name, in the second the median, in the third the upper 68% credibility band in the fourth column the 68% lower credibility band of the parameter, as extracted from the posterior distribution. For more output results (e.g., model evaluations, predictions, plots) check out the tutorials!

CHAPTER 4

API

The core classes within `juliet` are the `load` and `fit` classes. When creating a `juliet.load` object, the returned object will be able to call a `fit` function which in turn returns a `juliet.fit` object, which saves all the information about the fit (results statistics, posteriors, model evaluations, etc.) — these classes are explained in detail below:

class juliet.**load**(*priors=None*, *starting_point=None*, *input_folder=None*, *t_lc=None*, *y_lc=None*, *yerr_lc=None*, *t_rv=None*, *y_rv=None*, *yerr_rv=None*, *GP_regressors_lc=None*, *linear_regressors_lc=None*, *GP_regressors_rv=None*, *linear_regressors_rv=None*, *out_folder=None*, *lcfilename=None*, *rvfilename=None*, *GPlceparamfile=None*, *GPrveparamfile=None*, *LMlceparamfile=None*, *LMrveparamfile=None*, *lctimedef='TDB'*, *rvtimedef='UTC'*, *ld_laws='quadratic'*, *priorfile=None*, *lc_n_supersamp=None*, *lc_exptime_supersamp=None*, *lc_instrument_supersamp=None*, *mag_to_flux=True*, *verbose=False*, *matern_eps=0.01*, *pickle_encoding=None*)

Given a dictionary with priors (or a filename pointing to a prior file) and data either given through arrays or through files containing the data, this class loads data into a juliet object which holds all the information about the dataset. Example usage:

```
>>> data = juliet.load(priors=priors,t_lc=times,y_lc=fluxes,yerr_lc=fluxes_errors)
```

Or, also,

```
>>> data = juliet.load(input_folder = folder)
```

Parameters

- **priors** – (optional, dict or string) This can be either a python `string` or a python `dict`. If a `dict`, this has to contain each of the parameters to be fit, along with their respective prior distributions and hyperparameters. Each key of this dictionary has to have a parameter name (e.g., `r1_p1`, `sigma_w_TESS`), and each of those elements are, in turn, dictionaries as well containing two keys: a `distribution` key which defines the prior distribution of the parameter and a `hyperparameters` key, which contains the hyperparameters of that distribution.

  Example setup of the `priors` dictionary:

```
>>> priors = {}
>>> priors['r1_p1'] = {}
>>> priors['r1_p1']['distribution'] = 'Uniform'
>>> priors['r1_p1']['hyperparameters'] = [0.,1.]
```

If a `string`, this has to contain the filename to a proper juliet prior file; the prior `dict` will then be generated from there. A proper prior file has in the first column the name of the parameter, in the second the name of the distribution, and in the third the hyperparameters of that distribution for the parameter.

Note that this along with either lightcurve or RV data or a `input_folder` has to be given in order to properly load a juliet data object.

- **`starting_point`** – (mandatory if using MCMC, useless if using nested samplers, dict) Dictionary indicating the starting value of each of the parameters for the MCMC run (i.e., currently only of use for `emcee`). Keys should be consistent with the `prior` namings above; each key should have an associated float with the starting value. This is of no use if using nested samplers (which sample directly from the prior).

- **`input_folder`** – (optional, string) Python `string` containing the path to a folder containing all the input data — this will thus be load into a juliet data object. This input folder has to contain at least a `priors.dat` file with the priors and either a `lc.dat` file containing lightcurve data or a `rvs.dat` file containing radial-velocity data. If in this folder a `GP_lc_regressors.dat` file or a `GP_rv_regressors.dat` file is found, data will be loaded into the juliet object as well.

  Note that at least this or a `priors` string or dictionary, along with either lightcurve or RV data has to be given in order to properly load a juliet data object.

- **`t_lc`** – (optional, dictionary) Dictionary whose keys are instrument names; each of those keys is expected to have arrays with the times corresponding to those instruments. For example,

```
>>> t_lc = {}
>>> t_lc['TESS'] = np.linspace(0,100,100)
```

  Is a valid input dictionary for `t_lc`.

- **`y_lc`** – (optional, dictionary) Similarly to `t_lc`, dictionary whose keys are instrument names; each of those keys is expected to have arrays with the fluxes corresponding to those instruments. These are expected to be consistent with the `t_lc` dictionaries.

- **`yerr_lc`** – (optional, dictionary) Similarly to `t_lc`, dictionary whose keys are instrument names; each of those keys is expected to have arrays with the errors on the fluxes corresponding to those instruments. These are expected to be consistent with the `t_lc` dictionaries.

- **`GP_regressors_lc`** – (optional, dictionary) Dictionary whose keys are names of instruments where a GP is to be fit. On each name/element, an array of regressors of shape `(m,n)` containing in each column the `n` GP regressors to be used for `m` photometric measurements has to be given. Note that `m` for a given instrument has to be of the same length as the corresponding `t_lc` for that instrument. Also, note the order of each regressor of each instrument has to match the corresponding order in the `t_lc` array. For example,

```
>>> GP_regressors_lc = {}
>>> GP_regressors_lc['TESS'] = np.linspace(-1,1,100)
```

  If a global model wants to be used, then the instrument should be `rv`, and each of the `m` rows should correspond to the `m` times.

- **linear_regressors_lc** – (optional, dictionary) Similarly as for GP_regressors_lc, this is a dictionary whose keys are names of instruments where a linear regression is to be fit. On each name/element, an array of shape (q,p) containing in each column the p linear regressors to be used for the q photometric measurements. Again, note the order of each regressor of each instrument has to match the corresponding order in the t_lc array.
- **GP_regressors_rv** – (optional, dictionary) Same as GP_regressors_lc but for the radial-velocity data. If a global model wants to be used, then the instrument should be lc, and each of the m rows should correspond to the m times.
- **linear_regressors_rv** – (optional, dictionary) Same as linear_regressors_lc, but for the radial-velocities.
- **t_rv** – (optional, dictionary) Same as t_lc, but for the radial-velocities.
- **y_rv** – (optional, dictionary) Same as y_lc, but for the radial-velocities.
- **yerr_rv** – (optional, dictionary) Same as yerr_lc, but for the radial-velocities.
- **out_folder** – (optional, string) If a path is given, results will be saved to that path as a pickle file, along with all inputs in the standard juliet format.
- **lcfilename** – (optional, string) If a path to a lightcurve file is given, t_lc, y_lc, yerr_lc and instruments_lc will be read from there. The basic file format is a pure ascii file where times are in the first column, relative fluxes in the second, errors in the third and instrument names in the fourth. If more columns are given for a given instrument, those will be identified as linear regressors for those instruments.
- **rvfilename** – (optional, string) Same as lcfilename, but for the radial-velocities.
- **GPlceparamfile** – (optional, string) If a path to a file is given, the columns of that file will be used as GP regressors for the lightcurve fit. The file format is a pure ascii file where regressors are given in different columns, and the last column holds the instrument name. The order of this file has to be consistent with t_lc and/or the lcfilename file. If a global model wants to be used, set the instrument names of all regressors to lc.
- **GPrveparamfile** – (optional, string) Same as GPlceparamfile but for the radial-velocities. If a global model wants to be used, set the instrument names of all regressors to rv.
- **LMlceparamfile** – (optional, string) If a path to a file is given, the columns of that file will be used as linear regressors for the lightcurve fit. The file format is a pure ascii file where regressors are given in different columns, and the last column holds the instrument name. The order of this file has to be consistent with t_lc and/or the lcfilename file. If a global model wants to be used, set the instrument names of all regressors to lc.
- **LMrveparamfile** – (optional, string) Same as LMlceparamfile but for the radial-velocities. If a global model wants to be used, set the instrument names of all regressors to rv.
- **lctimedef** – (optional, string) Time definitions for each of the lightcurve instruments. Default is to assume all instruments (in lcs and rvs) have the same time definitions. If more than one instrument is given, this string should have instruments and time-definitions separated by commas, e.g., TESS-TDB, LCOGT-UTC, etc.
- **rvtimedef** – (optional, string) Time definitions for each of the radial-velocity instruments. Default is to assume all instruments (in lcs and rvs) have the same time definitions. If more than one instrument is given, this string should have instruments and time-definitions separated by commas, e.g., FEROS-TDB, HARPS-UTC, etc.

- **ld_laws** – (optional, string) Limb-darkening law to be used for each instrument. Default is `quadratic` for all instruments. If more than one instrument is given, this string should have instruments and limb-darkening laws separated by commas, e.g., `TESS-quadratic, LCOGT-linear`.
- **priorfile** – (optional, string) If a path to a file is given, it will be assumed this is a prior file. The `priors` dictionary will be overwritten by the data in this file. The file structure is a plain ascii file, with the name of the parameters in the first column, name of the prior distribution in the second column and hyperparameters in the third column.
- **lc_instrument_supersamp** – (optional, array of strings) Define for which lightcurve instruments super-sampling will be applied (e.g., in the case of long-cadence integrations). e.g., `lc_instrument_supersamp = ['TESS','K2']`
- **lc_n_supersamp** – (optional, array of ints) Define the number of datapoints to super-sample. Order should be consistent with order in `lc_instrument_supersamp`. e.g., `lc_n_supersamp = [20,30]`.
- **lc_exptime_supersamp** – (optional, array of floats) Define the exposure-time of the observations for the supersampling. Order should be consistent with order in `lc_instrument_supersamp`. e.g., `lc_exptime_supersamp = [0.020434, 0.020434]`
- **verbose** – (optional, boolean) If True, all outputs of the code are printed to terminal. Default is False.
- **matern_eps** – (optional, float) Epsilon parameter for the Matern approximation (see celerite documentation).
- **pickle_encoding** – (optional, string) Define pickle encoding in case fit was done with Python 2.7 and results are read with Python 3.

append_GP (*ndata*, *instrument_indexes*, *GP_arguments*, *inames*)
This function appends all the GP regressors into one — useful for the global models.

convert_input_data (*t*, *y*, *yerr*)
This converts the input dictionaries to arrays (this is easier to handle internally within juliet; input dictionaries are just asked because it is easier for the user to pass them).

convert_to_dictionary (*t*, *y*, *yerr*, *instrument_indexes*)
Convert data given in arrays to dictionaries for easier user usage

data_preparation (*times*, *instruments*, *linear_regressors*)
This function generates f useful internal arrays for this class: inames which saves the instrument names, `global_times` which is a "flattened" array of the `times` dictionary where all the times for all instruments are stacked, instrument_indexes, which is a dictionary that has, for each instrument the indexes of the `global_times` corresponding to each instrument, lm_boolean which saves booleans for each instrument to indicate if there are linear regressors and lm_arguments which are the linear-regressors for each instrument.

fit (***kwargs*)
Perhaps the most important function of the juliet data object. This function fits your data using the nested sampler of choice. This returns a results object which contains all the posteriors information.

generate_datadict (*dictype*)
This generates the options dictionary for lightcurves, RVs, and everything else you want to fit. Useful for the fit, as it separaters options per instrument.

Parameters **dictype** – (string) Defines the type of dictionary type. It can either be 'lc' (for the lightcurve dictionary) or 'rv' (for the radial-velocity one).

save_data (*fname*, *t*, *y*, *yerr*, *instruments*, *lm_boolean*, *lm_arguments*)
This function saves t,y,yerr,instruments,lm_boolean and lm_arguments data to fname.

save_priorfile (*fname*)
This function saves a priorfile file out to fname

save_regressors (*fname*, *GP_arguments*)
This function saves the GP regressors to fname.

class juliet.**fit**(*data*, *sampler='multinest'*, *n_live_points=500*, *nwalkers=100*, *nsteps=300*, *nburnin=500*, *emcee_factor=0.0001*, *ecclim=1.0*, *pl=0.0*, *pu=1.0*, *ta=2458460.0*, *nthreads=None*, *use_ultranest=False*, *use_dynesty=False*, *dynamic=False*, *dynesty_bound='multi'*, *dynesty_sample='rwalk'*, *dynesty_nthreads=None*, *dynesty_n_effective=inf*, *dynesty_use_stop=True*, *dynesty_use_pool=None*, ***kwargs*)
Given a juliet data object, this class performs a fit to the data and returns a results object to explore the results. Example usage:

```
>>> results = juliet.fit(data)
```

Parameters **data** – (juliet object) An object containing all the information regarding the data to be fitted, including options of the fit. Generated via juliet.load().

On top of `data`, a series of extra keywords can be included:

Parameters

- **sampler** – (optional, string) String defining the sampler to be used on the fit. Current possible options include `multinest` to use PyMultiNest (via importance nested sampling), `dynesty` to use Dynesty's importance nested sampling, `dynamic_dynesty` to use Dynesty's dynamic nested sampling algorithm, `ultranest` to use Ultranest, `slicesampler_ultranest` to use Ultranest's slice sampler and `emcee` to use emcee. Default is `multinest` if PyMultiNest is installed; `dynesty` if not.
- **n_live_points** – (optional, int) Number of live-points to use on the nested sampling samplers. Default is 500.
- **nwalkers** – (optional if using emcee, int) Number of walkers to use by emcee. Default is 100.
- **nsteps** – (optional if using MCMC, int) Number of steps/jumps to perform on the MCMC run. Default is 300.
- **nburnin** – (optional if using MCMC, int) Number of burnin steps/jumps when performing the MCMC run. Default is 500.
- **emcee_factor** – (optional, for emcee only, float) Factor multiplying the standard-gaussian ball around which the initial position is perturbed for each walker. Default is 1e-4.
- **ecclim** – (optional, float) Upper limit on the maximum eccentricity to sample. Default is `1`.
- **pl** – (optional, float) If the `(r1,r2)` parametrization for `(b,p)` is used, this defines the lower limit of the planet-to-star radius ratio to be sampled. Default is `0`.
- **pu** – (optional, float) Same as `pl`, but for the upper limit. Default is `1`.
- **ta** – (optional, float) Time to be substracted to the input times in order to generate the linear and/or quadratic trend to be added to the model. Default is 2458460.

- **nthreads** – (optinal, int) Define the number of threads to use within dynesty or emcee. Default is to use just 1. Note this will not impact PyMultiNest or UltraNest runs — these can be parallelized via MPI only.

In addition, any number of extra optional keywords can be given to the call, which will be directly ingested into the sampler of choice. For a full list of optional keywords for...

- ...PyMultiNest, check the docstring of `PyMultiNest`'s `run` function.
- ...any of the nested sampling algorithms in `dynesty`, see the docstring on the `run_nested` function.
- ...the non-dynamic nested sampling algorithm implemented in `dynesty`, see the docstring on `dynesty.dynesty.NestedSampler` in dynesty's documentation.
- ...the dynamic nested sampling in `dynesty` check the docstring for `dynesty.dynesty.DynamicNestedSampler` in dynesty's documentation.
- ...the `ultranest` sampler, see the docstring for *ultranest.integrationr.ReactiveNestedSampler* in ultranest's documentation

Finally, since `juliet` version 2.0.26, the following keywords have been deprecated, and are recommended to be removed from code using `juliet` as they will be removed sometime in the future:

Parameters

- **use_dynesty** – (optional, boolean) If `True`, use dynesty instead of *MultiNest* for posterior sampling and evidence evaluation. Default is `False`, unless *MultiNest* via `pymultinest` is not working on the system.
- **dynamic** – (optional, boolean) If `True`, use dynamic Nested Sampling with dynesty. Default is `False`.
- **dynesty_bound** – (optional, string) Define the dynesty bound method to use (currently either `single` or `multi`, to use either single ellipsoids or multiple ellipsoids). Default is `multi` (for details, see the dynesty API).
- **dynesty_sample** – (optional, string) Define the sampling method for dynesty to use. Default is `rwalk`. Accorfing to the dynesty API, this should be changed depending on the number of parameters being fitted. If smaller than about 20, `rwalk` is optimal. For larger dimensions, `slice` or `rslice` should be used.
- **dynesty_nthreads** – (optional, int) Define the number of threads to use within dynesty. Default is to use just 1.
- **dynesty_n_effective** – (optional, int) Minimum number of effective posterior samples when using `dynesty`. If the estimated "effective sample size" exceeds this number, sampling will terminate. Default is `None`.
- **dynesty_use_stop** – (optional, boolean) Whether to evaluate the `dynesty` stopping function after each batch. Disabling this can improve performance if other stopping criteria such as maxcall are already specified. Default is `True`.
- **dynesty_use_pool** – (optional, dict) A dictionary containing flags indicating where a pool in `dynesty` should be used to execute operations in parallel. These govern whether `prior_transform` is executed in parallel during initialization (`'prior_transform'`), loglikelihood is executed in parallel during initialization (`'loglikelihood'`), live points are proposed in parallel during a run (`'propose_point'`), and bounding distributions are updated in parallel during a run (`'update_bound'`). Default is True for all options.

The returned `fit` object, in turn, also has other objects inherted in it. In particular, if `results` is a `juliet.fit` object, `results.lc` and `results.rv` are `juliet.model` objects that host all the details about the dataset being modelled. This follows the model definition outlined in Section 2 of the juliet paper:

class juliet.**model**(*data*, *modeltype*, *pl=0.0*, *pu=1.0*, *ecclim=1.0*, *ta=2458460.0*, *log_like_calc=False*)

Given a juliet data object, this kernel generates either a lightcurve or a radial-velocity object. Example usage:

```
>>> model = juliet.model(data, modeltype = 'lc')
```

Parameters

- **data** – (juliet.load object) An object containing all the information about the current dataset.
- **modeltype** – (optional, string) String indicating whether the model to generate should be a lightcurve ('lc') or a radial-velocity ('rv') model.
- **pl** – (optional, float) If the `(r1,r2)` parametrization for `(b,p)` is used, this defines the lower limit of the planet-to-star radius ratio to be sampled. Default is `0`.
- **pu** – (optional, float) Same as `pl`, but for the upper limit. Default is `1`.
- **ecclim** – (optional, float) This parameter sets the maximum eccentricity allowed such that a model is actually evaluated. Default is `1`.
- **log_like_calc** – (optional, boolean) If True, it is assumed the model is generated to generate likelihoods values, and thus this skips the saving/calculation of the individual models per planet (i.e., `self.model['p1']`, `self.model['p2']`, etc. will not exist). Default is False.

evaluate_model(*instrument=None*, *parameter_values=None*, *resampling=None*, *nresampling=None*, *etresampling=None*, *all_samples=False*, *nsamples=1000*, *return_samples=False*, *t=None*, *GPregressors=None*, *LMregressors=None*, *return_err=False*, *alpha=0.68*, *return_components=False*, *evaluate_transit=False*)

This function evaluates the current lc or rv model given a set of posterior distribution samples and/or parameter values. Example usage:

```
>>> dataset = juliet.load(priors=priors, t_lc = times, y_lc = fluxes, yerr_lc␣
↪= fluxes_error)
>>> results = dataset.fit()
>>> transit_model, error68_up, error68_down = results.lc.evaluate('TESS',␣
↪return_err=True)
```

Or:

```
>>> dataset = juliet.load(priors=priors, t_rv = times, y_rv = fluxes, yerr_rv␣
↪= fluxes_error)
>>> results = dataset.fit()
>>> rv_model, error68_up, error68_down = results.rv.evaluate('FEROS', return_
↪err=True)
```

Parameters **instrument** – (optional, string)

Instrument the user wants to evaluate the model on. It is expected to be given for non-global models, not necessary for global models.

Parameters **parameter_values** – (optional, dict)

Dictionary containing samples of the posterior distribution or, more generally, parameter valuesin it. Each key is a parameter name (e.g. 'p_p1', 'q1_TESS', etc.), and inside each of those keys an array of N samples is expected (i.e., parameter_values['p_p1'] is an array of length N). The indexes have to be consistent between different parameters.

Parameters **resampling** – (optional, boolean)

Boolean indicating if the model needs to be resampled or not. Only works for lightcurves.

Parameters **nresampling** – (optional, int)

Number of points to resample for a given time-stamp. Only used if resampling = True. Only applicable to lightcurves.

Parameters **etresampling** – (optional, double)

Exposure time of the resampling (same unit as times). Only used if resampling = True. Only applicable to lightcurves.

Parameters **all_samples** – (optional, boolean)

If True, all posterior samples will be used to evaluate the model. Default is False.

Parameters **nsamples** – (optional, int)

Number of posterior samples to be used to evaluate the model. Default is 1000 (note each call to this function will sample *nsamples* different samples from the posterior, so no two calls are exactly the same).

Parameters **return_samples** – (optional, boolean)

Boolean indicating whether the user wants the posterior model samples (i.e., the models evaluated in each of the posterior sample draws) to be returned. Default is False.

Parameters **t** – (optional, numpy array)

Array with the times at which the model wants to be evaluated.

Parameters **GPRegressors** – (optional, numpy array)

Array containing the GP Regressors onto which to evaluate the models. Dimensions must be consistent with input *t*. If model is global, this needs to be a dictionary.

Parameters **LMRegressors** – (optional, numpy array or dictionary)

If the model is not global, this is an array containing the Linear Regressors onto which to evaluate the model for the input instrument. Dimensions must be consistent with input *t*. If model is global, this needs to be a dictionary.

Parameters **return_err** – (optional, boolean)

If True, this returns the credibility interval on the evaluated model. Default credibility interval is 68%.

Parameters **alpha** – (optional, double)

Credibility interval for return_err. Default is 0.68, i.e., the 68% credibility interval.

Parameters **return_components** – (optional, boolean)

If True, each component of the model is returned (i.e., the Gaussian Process component, the Linear Model component, etc.).

Parameters **evaluate_transit** – (optional, boolean)

If True, the function evaluates only the transit model and not the Gaussian Process or Linear Model components.

Returns By default, the function returns the median model as evaluated with the posterior samples. Depending on the options chosen by the user, this can return up to 5 elements (in that order): *model_samples*, *median_model*, *upper_CI*, *lower_CI* and *components*. The first is an array with all the model samples as evaluated from the posterior. The second is the median model. The third and fourth are the uppper and lower Credibility Intervals, and the latter is a dictionary with the model components.

Finally, the `juliet.load` object also contains a dictionary (`juliet.load.lc_options` for lightcurves and `juliet.load.rv_options` for radial-velocities) which holds, if a gaussian-process is being used to model the noise, a `juliet.gaussian_process` object. This class handles everything related to the gaussian-processes, from model and parameter names/values, to log-likelihood evaluations. This class is defined below:

class juliet.**gaussian_process**(*data*, *model_type*, *instrument*, *george_hodlr=True*, *matern_eps=0.01*)

Given a juliet data object (created via juliet.load), a model type (i.e., is this a GP for a RV or lightcurve dataset) and an instrument name, this object generates a Gaussian Process (GP) object to use within the juliet library. Example usage:

```
>>> GPmodel = juliet.gaussian_process(data, model_type = 'lc', instrument = 'TESS
↪')
```

:param data (juliet.load object) Object containing all the information about the current dataset. This will help in determining the type of kernel the input instrument has and also if the instrument has any errors associated with it to initialize the kernel.

Parameters

- **model_type** – (string) A string defining the type of data the GP will be modelling. Can be either `lc` (for photometry) or `rv` (for radial-velocities).
- **instrument** – (string) A string indicating the name of the instrument the GP is being applied to. This string simplifies cross-talk with juliet's `posteriors` dictionary.
- **george_hodlr** – (optional, boolean) If True, this uses George's HODLR solver (faster).

CHAPTER 5

Lightcurve fitting with juliet

We have already exemplified how to fit a basic transit lightcurve in the *Getting started* section with `juliet`. Here, however, we explore some interesting extra features of the lightcurve fitting process, including limb-darkening laws, parameter transformations and fitting of data from multiple-instruments simultaneously, along with useful details on the model evaluations with `juliet`.

Before going into the tutorial, it is useful to first understand the lightcurve model that `juliet` uses. In the absence of extra linear terms (which we will deal with in the *Incorporating linear models* tutorial), a `juliet` lightcurve model for a given instrument i is given by (see Section 2 of the juliet paper)

$\mathcal{M}_i(t) + \epsilon_i(t)$,

where

$\mathcal{M}_i(t) = [\mathcal{T}_i(t)D_i + (1 - D_i)] \left(\frac{1}{1+D_i M_i}\right)$

is the photometric model composed of the dilution factor D_i, the relative out-of-transit target flux M_i, and the transit model for the instrument $\mathcal{T}_i(t)$ (defined by the transit parameters and by the instrument-dependant limb-darkening coefficients — see the *Models, priors and outputs* section for details). Here, $\epsilon_i(t)$ is a stochastic process that defines a "noise model" for the dataset. In this section we will assume that $\epsilon_i(t)$ is white-gaussian noise, i.e., $\epsilon_i(t) \sim \mathcal{N}(0, \sqrt{\sigma_i(t)^2 + \sigma_{w,i}^2})$, where $\mathcal{N}(\mu, \sigma)$ denotes a normal distribution with mean μ and standard-deviation σ, and where $\sigma_i(t)$ are the errors on the datapoint at time t and $\sigma_{w,i}$ is a so-called "jitter" term. We deal with more general noise models in the *Incorporating Gaussian Processes* tutorial.

The `juliet` lightcurve model is a bit different than typical lightcurve models which typically only fit for an out-of-transit flux offset. The first difference is that our model includes a dilution factor D_i which allows the user to account for possible contaminating sources in the aperture that might produce a smaller transit depth than the real one. In fact, if there are n sources with fluxes F_n in the aperture and the target has a flux F_T, then one can show (see Section 2 of the juliet paper) that the dilution factor can be interpreted as

$D_i = \frac{1}{1+\sum_n F_n/F_T}$.

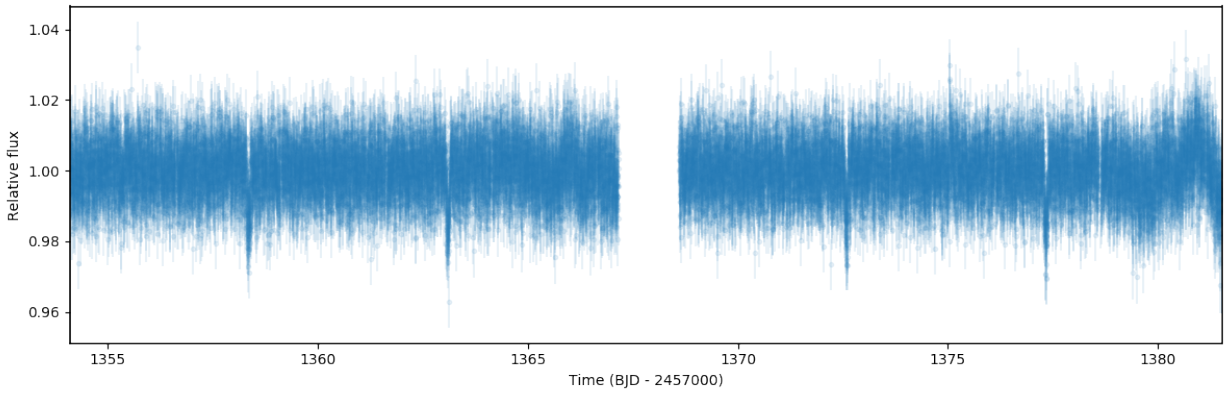
A dilution factor of 1, thus, implies no external contaminating sources. The second difference is that the relative out-of-transit target flux, M_i — which from hereon we refer to as the "mean out-of-transit flux" — is a multiplicative term and not an additive offset. This is because input lightcurves are usually normalized (typically with respect to the mean or the median), and in theory a simple additive offset might thus still not account for this pre-normalization of the

lightcurve. M_i, in turn, has a well defined interpretation: if the real out-of-transit flux was $F_T + \sum_n F_n + E$, where E is an offset flux given by light coming not from the sources F_n or from the target, F_T (e.g., background scattered light, a bias flux, etc.), then this term can be interpreted as E/F_T. As can be seen, then, with D_i and M_i, one can uniquely recover the real (relative to the target) fluxes.

5.1 Transit fits

To showcase the ability of `juliet` to fit transit lightcurves, we will play with the HATS-46b system (Brahm et al., 2017), as the *TESS* data for this system has interesting features that we will be using both in this tutorial and in the *Incorporating Gaussian Processes* tutorial. In this tutorial in particular, we will play with the data obtained in Sector 2, because it seems the level of variability/systematics in this particular dataset is much smaller than the one for Sector 1 (which we tackle in the *Incorporating Gaussian Processes* tutorial). First, let us download and plot the *TESS* data, taking the opportunity to also put the data in dictionaries so we can feed it to `juliet`:

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

# First, get arrays of times, normalized-fluxes and errors for HATS-46
# from Sector 1 from MAST:
t, f, ferr  = juliet.get_TESS_data('https://archive.stsci.edu/hlsps/'+\
                                   'tess-data-alerts/hlsp_tess-data-'+\
                                   'alerts_tess_phot_00281541555-s02_'+\
                                   'tess_v1_lc.fits')

# Put data arrays into dictionaries so we can fit it with juliet:
times, fluxes, fluxes_error = {},{},{}
times['TESS'], fluxes['TESS'], fluxes_error['TESS'] = t,f,ferr

# Plot data:
plt.errorbar(t, f, yerr=ferr, fmt='.')
plt.xlim([np.min(t),np.max(t)])
```



Pretty nice dataset! The transits can be clearly seen by eye. The period seems to be about $P \sim 4.7$ days, in agreement with the Brahm et al., 2017 study, and the time-of-transit center seems to be about $t_0 \sim 1358.4$ days. Let us now fit this lightcurve using these timing constraints as priors. We will use the same "non-informative" priors for the rest of the transit parameters as was already done for TOI-141b in the *Getting started* tutorial:

```
priors = {}

# Name of the parameters to be fit:
params = ['P_p1','t0_p1','r1_p1','r2_p1','q1_TESS','q2_TESS','ecc_p1','omega_p1',\
             'rho', 'mdilution_TESS', 'mflux_TESS', 'sigma_w_TESS']

# Distributions:
dists = ['normal','normal','uniform','uniform','uniform','uniform','fixed','fixed',\
                'loguniform', 'fixed', 'normal', 'loguniform']

# Hyperparameters
hyperps = [[4.7,0.1], [1358.4,0.1], [0.,1], [0.,1.], [0., 1.], [0., 1.], 0.0, 90.,\
                  [100., 10000.], 1.0, [0.,0.1], [0.1, 1000.]]

# Populate the priors dictionary:
for param, dist, hyperp in zip(params, dists, hyperps):
    priors[param] = {}
    priors[param]['distribution'], priors[param]['hyperparameters'] = dist, hyperp
```

Now let's fit the dataset with `juliet`, saving the results to a folder called `hats46`:

```
# Load and fit dataset with juliet:
dataset = juliet.load(priors=priors, t_lc = times, y_lc = fluxes, \
                  yerr_lc = fluxes_error, out_folder = 'hats46')

results = dataset.fit()
```

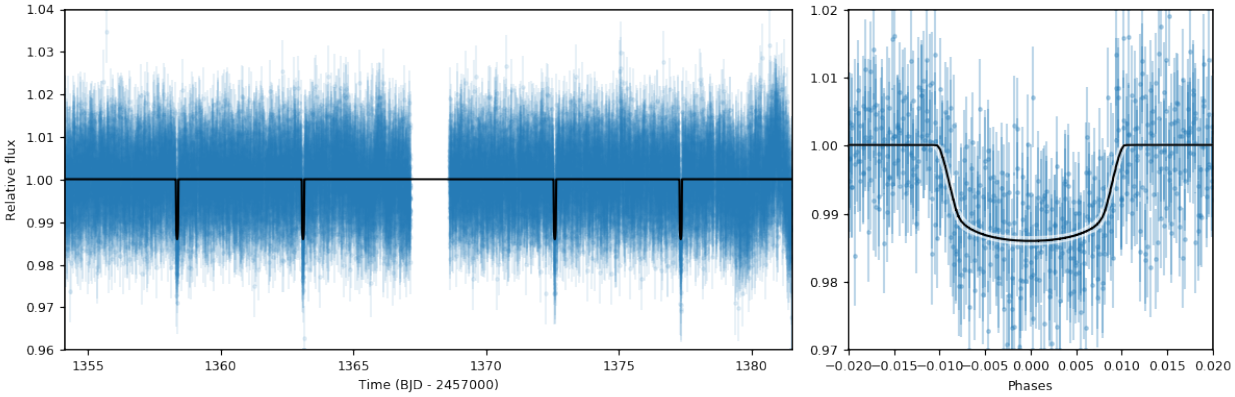
As was already shown in the *Getting started* tutorial, it is easy to plot the `juliet` fit results using the `results.lc.evaluate()` function. In the background, this function extracts by default `nsamples=1000` random samples from the joint posterior distribution of the parameters and evaluates the model using them — by default, a call to this function given an instrument name returns the median of all of those models. However, one can also retrieve the models that are about "1-sigma away" from this median model — i.e., the 68% credibility band of these models — by setting `return_err=True`. One can actually select the percentile credibility band with the `alpha` parameter (by default, `alpha=0.68`). Let us extract and plot the median model and the corresponding 68% credibility band around it using this function. We will create two plots: one of time versus flux, and another one with the phased transit lightcurve:

```
# Extract median model and the ones that cover the 68% credibility band around it:
transit_model, transit_up68, transit_low68  = results.lc.evaluate('TESS', return_
↪err=True)

# To plot the phased lighcurve we need the median period and time-of-transit center:
P, t0 = np.median(results.posteriors['posterior_samples']['P_p1']),\
        np.median(results.posteriors['posterior_samples']['t0_p1'])

# Get phases:
phases = juliet.get_phases(dataset.times_lc['TESS'], P, t0)


import matplotlib.gridspec as gridspec

# Plot the data. First, time versus flux --- plot only the median model here:
fig = plt.figure(figsize=(12,4))
gs = gridspec.GridSpec(1, 2, width_ratios=[2,1])
ax1 = plt.subplot(gs[0])

ax1.errorbar(dataset.times_lc['TESS'], dataset.data_lc['TESS'], \
```

(continues on next page)

(continued from previous page)

```
            yerr = dataset.errors_lc['TESS'], fmt = '.' , alpha = 0.1)

# Plot the median model:
ax1.plot(dataset.times_lc['TESS'], transit_model, color='black',zorder=10)

# Plot portion of the lightcurve, axes, etc.:
ax1.set_xlim([np.min(dataset.times_lc['TESS']),np.max(dataset.times_lc['TESS'])])
ax1.set_ylim([0.96,1.04])
ax1.set_xlabel('Time (BJD - 2457000)')
ax1.set_ylabel('Relative flux')

# Now plot phased model; plot the error band of the best-fit model here:
ax2 = plt.subplot(gs[1])
ax2.errorbar(phases, dataset.data_lc['TESS'], \
             yerr = dataset.errors_lc['TESS'], fmt = '.', alpha = 0.3)
idx = np.argsort(phases)
ax2.plot(phases[idx],transit_model[idx], color='black',zorder=10)
ax2.fill_between(phases[idx],transit_up68[idx],transit_low68[idx],\
                 color='white',alpha=0.5,zorder=5)
ax2.set_xlabel('Phases')
ax2.set_xlim([-0.015,0.015])
ax2.set_ylim([0.98,1.02])
```



As can be seen, the lightcurve model is quite precise! In the code above we also made use of a function and a dictionary which we have not introduced in their entirety yet. The first is the `juliet.get_phases(t, P, t0)` function — this gives you back the phases at the times `t` given a period `P` and a time-of-transit center `t0`. The second is a very important dictionary: it was already briefly introduced in the *Models, priors and outputs* section, but this introduction did not pay justice to its importance. This is the `results.posteriors` dictionary. The `posterior_samples` key of this dictionary stores the posterior distribution of the fitted parameters — we make use of this dictionary in detail in the next part of the tutorial.

5.2 Transit parameter transformations

In the fit done in the previous section we fitted the Sector 2 *TESS* lightcurve of HATS-46b. There, however, we fitted for the transformed parameters `r1_p1` and `r2_p1` which parametrize the planet-to-star radius ratio, $p = R_p/R_*$, and the impact parameter, in our case given by $b = (a/R_*)\cos i$, and the limb-darkening parametrization `q1_TESS` and `q2_TESS`, which in our case parametrize the coefficients u_1 and u_2 of the quadratic limb-darkening law. How do we transform the posterior distributions of those parametrizations, stored in the `results.`

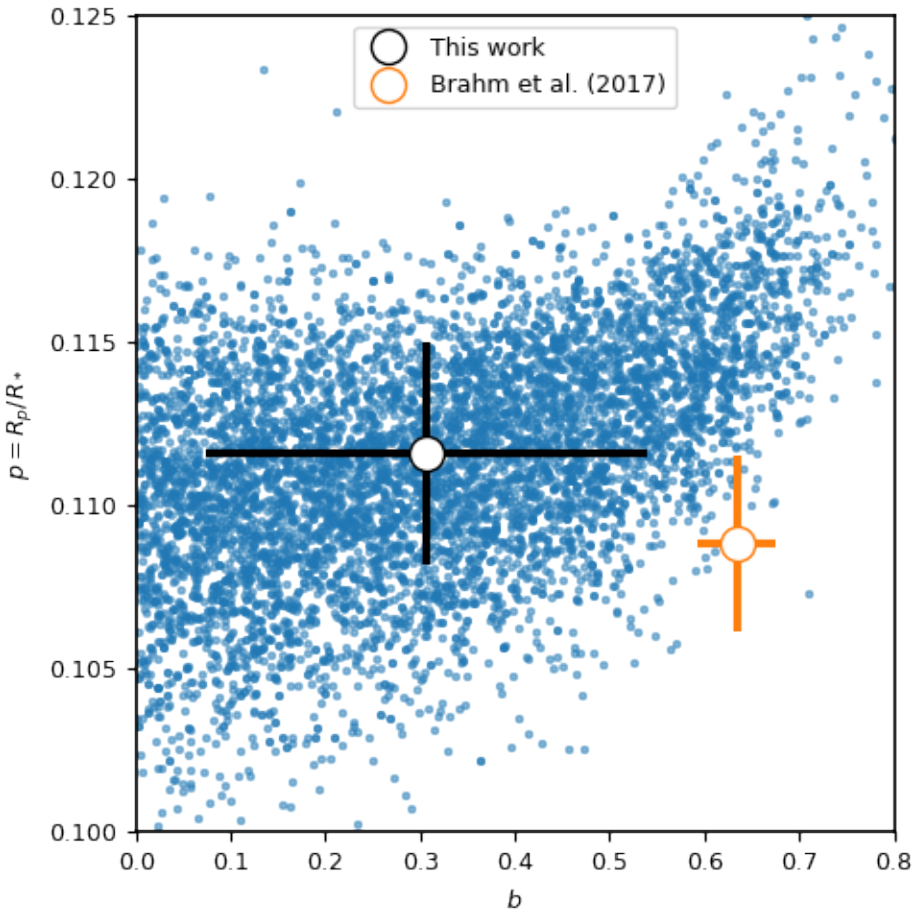
`posteriors['posterior_samples']` dictionary back to their physical parameters? `juliet` has built-in functions to do just this.

To transform from the (r_1, r_2) plane to the (b, p) plane, we have implemented the transformations described in Espinoza (2018). These require one defines the minimum and maximum allowed planet-to-star radius ratio — by default, within `juliet` the parametrization allows to search for all planet-to-star radius ratios from $p_l = 0$ to $p_u = 1$ (and these can be modified in the `fit` object — e.g., `dataset.fit(...,pl= 0.0, pu = 0.2)`). The values used for each fit are always stored in `results.posteriors['pl']` and `results.posteriors['pu']`. In our case, then, to obtain the posterior distribution of b and p, we can use the `juliet.utils.reverse_bp(r1,r2, pl,pu)` function which takes samples from the (r_1, r_2) plane and converts them back to the (b, p) plane. Let us do this transformation for the HATS-46b fit done above and compare with the results obtained in Brahm et al., 2017:

```
fig = plt.figure(figsize=(5,5))
# Store posterior samples for r1 and r2:
r1, r2 = results.posteriors['posterior_samples']['r1_p1'],\
         results.posteriors['posterior_samples']['r2_p1']

# Transform back to (b,p):
b,p = juliet.utils.reverse_bp(r1, r2, 0., 1.)

# Plot posterior distribution:
plt.plot(b,p,'.',alpha=0.5)

# Extract median and 1-sigma errors for b and p from
# the posterior distribution:
bm,bu,bl = juliet.utils.get_quantiles(b)
pm,pu,pl = juliet.utils.get_quantiles(p)

# Plot them:
plt.errorbar(np.array([bm]),np.array([pm]),\
             xerr = np.array([[bu-bm,bm-bl]]),\
             yerr = np.array([[pu-pm,pm-pl]]),\
             fmt = 'o', mfc = 'white', mec = 'black',\
             ecolor='black', ms = 15, elinewidth = 3, \
             zorder = 5, label = 'This work')

# Plot values in Brahm et al. (2017):
plt.errorbar(np.array([0.634]),np.array([0.1088]),\
             xerr = np.array([[0.042,0.034]]), \
             yerr = np.array([0.0027]),zorder = 5,\
             label = 'Brahm et al. (2017)', fmt='o', \
             mfc = 'white', elinewidth = 3, ms = 15)

plt.legend()
plt.xlim([0.,0.8])
plt.ylim([0.1,0.125])
plt.xlabel('$b$')
plt.ylabel('$p = R_p/R_*$')
```

The agreement with Brahm et al., 2017 is pretty good! The planet-to-star radius ratios are consistent within one-sigma, and the (uncertain for *TESS*) impact parameter is consistent at less thant 2-sigma with the one published in that work.

What about the limb-darkening coefficients? `juliet` also has a built-in function to perform the inverse transformation in order to obtain them — this is the `juliet.utils.reverse_ld_coeffs()` function — given a limb-darkening law and the parameters q_1 and q_2, this function gives back the limb-darkening coefficients u_1 and u_2. Let us plot the posterior distribution of the limb-darkening coefficients; let's compare them to theoretical limb-darkening coefficients using limb-darkening (Espinoza & Jordan, 2015):


This work
Brahm et al. (2017)
0.125
0.120
0.115
0.110
0.105
0.100
$p = R_p/R_*$
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
b

```
fig = plt.figure(figsize=(5,5))
# Store posterior samples for q1 and q2:
q1, q2 = results.posteriors['posterior_samples']['q1_TESS'],\
          results.posteriors['posterior_samples']['q2_TESS']

# Transform back to (u1,u2):
u1, u2 = juliet.utils.reverse_ld_coeffs('quadratic', q1, q2)

# Plot posterior distribution:
plt.plot(u1,u2,'.',alpha=0.5)

# Plot medians and errors implied by the posterior:
u1m,u1u,u1l = juliet.utils.get_quantiles(u1)
u2m,u2u,u2l = juliet.utils.get_quantiles(u2)
plt.errorbar(np.array([u1m]),np.array([u2m]),\
             xerr = np.array([[u1u-u1m,u1m-u1l]]),\
             yerr = np.array([[u2u-u2m,u2m-u2l]]),\
             fmt = 'o', mfc = 'white', mec = 'black',\
             ecolor='black', ms = 13, elinewidth = 3, \
             zorder = 5, label = 'This work')

plt.plot(np.array([0.346,0.346]),np.array([-1,1]),'--',color='cornflowerblue')
plt.plot(np.array([-1,1]),np.array([0.251,0.251]),'--',color='cornflowerblue',label=
↪'ATLAS')

plt.plot(np.array([0.377,0.377]),np.array([-1,1]),'--',color='red')
plt.plot(np.array([-1,1]),np.array([0.214,0.214]),'--',color='red',label='PHOENIX')
plt.legend()

plt.xlabel('$u_1$')
plt.ylabel('$u_2$')
plt.xlim([0.0,1.0])
plt.ylim([-0.5,1.0])
```

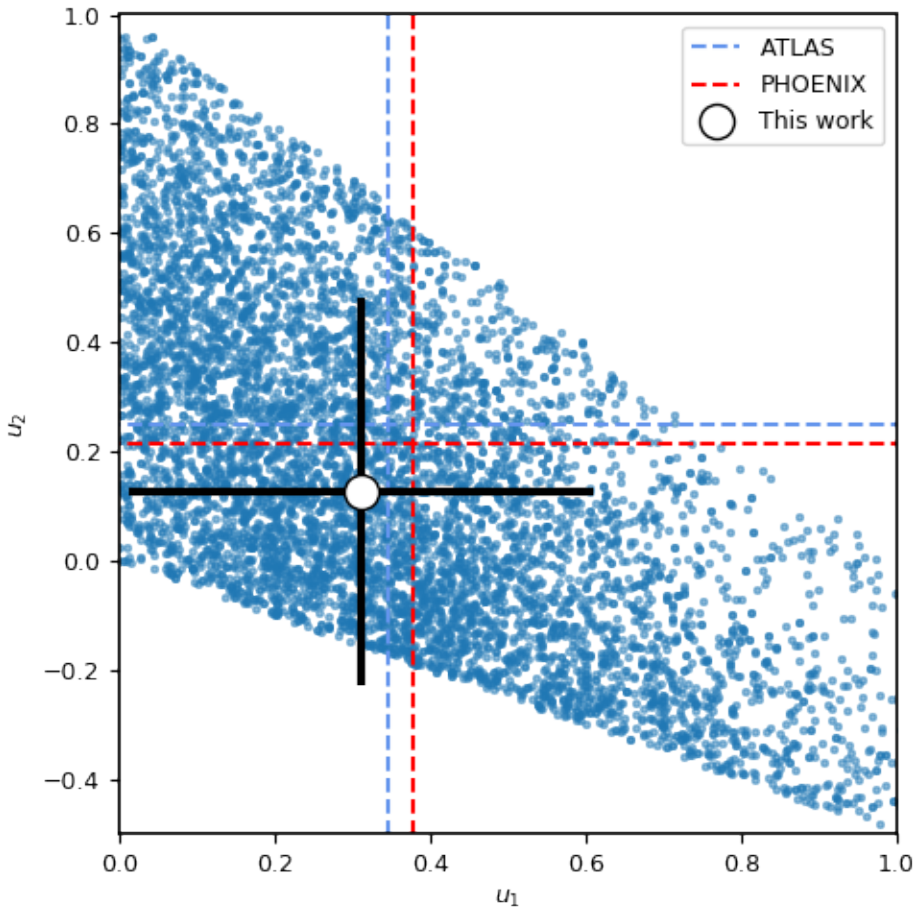
The agreement with the theory is pretty good in this case! It was kind of expected — HATS-46 is a solar-type star after all. Notice the triangular shape of the parameter spaced explored? This is what the (q_1, q_2) sampling is expected to sample — the triangle englobes all the physically plausible parameter space for the limb-darkening coefficients (positive, decreasing-to-the-limb limb-darkening profiles). For details, see Kipping (2013).

5.3 Fitting multiple datasets

In the previous sections we have been fitting the *TESS* data only. What if we want to add extra datasets and fit all of them *jointly* in order to extract the posterior distribution of the transit parameters? As it was already mentioned, this is very easy to do with `juliet`: you simply add new elements/keys to the dictionary one gives as inputs to it. Of course, you also have to add some extra priors for the extra instruments: in particular, one has to define a jitter ($\sigma_{w,i}$), dilution factor (D_i), mean out-of-transit flux (M_i) and limb-darkening parametrization (q_1 if a linear law wants to be assumed, or also give q_2 if a quadratic law wants to be used). Let us fit the *TESS* data together with the follow-up lightcurves obtained by Brahm et al., 2017 from the Las Cumbres Observatory Global Telescope Network (LCOGT) and the 1m Swope Telescope. These can be obtained from CDS following the paper link, but we have uploaded them here and here so it is easier to follow this tutorial. Once that data is downloaded, we can load this data in `juliet` as follows:

```
# Add LCOGT and SWOPE data to the times, fluxes and fluxes_error dictionary.
# Fill also the priors for these instruments:
```

(continues on next page)


ATLAS
PHOENIX
This work
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
u_2
0.0
0.2
0.4
0.6
0.8
1.0
u_1

(continued from previous page)

```
for instrument in ['LCOGT','SWOPE']:
    # Open dataset files, extract times, fluxes and errors to arrays:
    t2,f2,ferr2 = np.loadtxt('hats-46_data_'+instrument+'.txt',\
                             unpack=True,usecols=(0,1,2))
    # Add them to the data dictionaries which already contain the TESS data (see␣
↪above):
    times[instrument], fluxes[instrument], fluxes_error[instrument] = \
                                                t2-2457000, f2, ferr2

    # Add priors to the already defined ones above for TESS, but for the other␣
↪instruments:
    params = ['sigma_w_','mflux_','mdilution_','q1_','q2_']
    dists = ['loguniform', 'normal', 'fixed', 'uniform', 'uniform']
    hyperps = [[0.1,1e5], [0.0,0.1], 1.0, [0.0,1.0], [0.0,1.0]]

    for param, dist, hyperp in zip(params, dists, hyperps):
        priors[param+instrument] = {}
        priors[param+instrument]['distribution'], \
        priors[param+instrument]['hyperparameters'] = dist, hyperp
```

And with this one can simply run a `juliet` fit again:

```
dataset = juliet.load(priors=priors, t_lc = times, y_lc = fluxes, \
                      yerr_lc = fluxes_error, out_folder = 'hats46-extra')

results = dataset.fit(n_live_points=300)
```

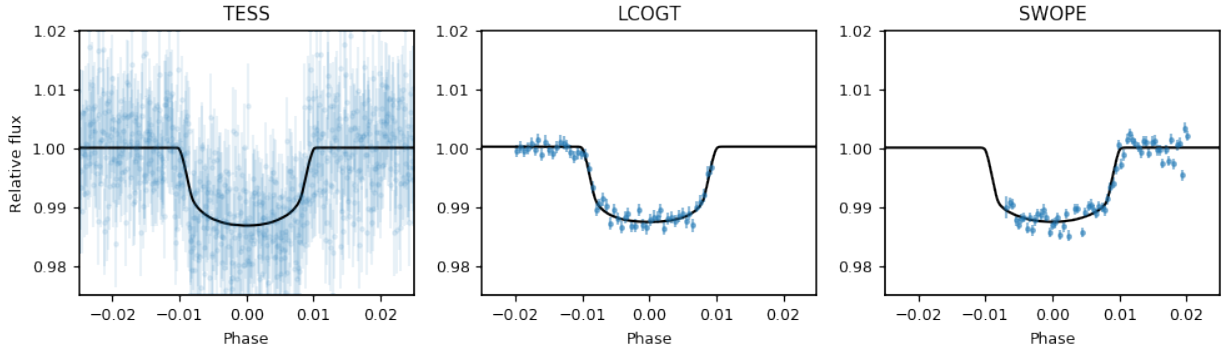
This can actually take a little bit longer than just fitting the *TESS* data (a couple of extra minutes) — it is a 17-dimensional problem after all. Let us plot the results of the joint instrument fit:

```
# Extract new period and time-of-transit center:
P,t0 =  np.median(results.posteriors['posterior_samples']['P_p1']),\
        np.median(results.posteriors['posterior_samples']['t0_p1'])

# Generate arrays to super-sample the models:
model_phases = np.linspace(-0.04,0.04,1000)
model_times = model_phases*P + t0

# Plot figure:
fig = plt.figure(figsize=(10,3))
instruments = ['TESS','LCOGT','SWOPE']
alphas = [0.1, 0.5, 0.5]
for i in range(3):
    instrument = instruments[i]
    plt.subplot('13'+str(i+1))
    # Plot phase-folded data:
    phases = juliet.utils.get_phases(dataset.times_lc[instrument], P, t0)
    plt.errorbar(phases, dataset.data_lc[instrument], \
                 yerr = dataset.errors_lc[instrument], fmt = '.' , alpha = alphas[i])
    # Evaluate model in the supersampled times, plot on top of data:
    model_lc = results.lc.evaluate(instrument, t = model_times)
    plt.plot(model_phases,model_lc,color='black')
    plt.title(instrument)
    plt.xlabel('Phase')
    if i == 0:
        plt.ylabel('Relative flux')
```

(continues on next page)

(continued from previous page)

```
    plt.xlim([-0.025,0.025])
    plt.ylim([0.975,1.02])
```



Pretty nice fit! The Swope data actually shows a little bit more scatter — indeed, the $\sigma_{w,SWOPE} = 1269^{+185}_{-155}$ ppm, which indicates that there seems to be some extra process happening in the lightcurve (e.g., systematics), which are being modelled in our fit with a simple jitter term. So, how does the posteriors of our parameters compare with that of the *TESS*-only fit? We can repeat the plot made above for the planet-to-star radius ratio and impact parameter to check:

Interesting! The transit depth is consistent between fits and with the work of Brahm et al., 2017. Interestingly, the impact parameter is practically the same as the *TESS*-only fit, and just shrinked a little bit. It is still consistent at 2-sigma with the work of Brahm et al., 2017, however.

5.4 A word on limb-darkening and model selection

Throughout the tutorial, we have not explicitly defined what limb-darkening laws we wanted to use for each dataset. By default, `juliet` assumes that if the user defines q_1 and q_2, then a quadratic law wants to be used, whereas if the user only gives q_1, a linear-law is assumed. In general, the limb-darkening law to use depends on the system under study (see, e.g., Espinoza & Jordan, 2016.), and thus the user might want to use laws *other* than the ones that are pre-defined by `juliet`. This can be easily done when loading a dataset via `juliet.load` using the `ld_laws` flag. This flag receives a string with the name of the law to use — currently supported laws are the `linear`, the `quadratic`, the `logarithmic` and the `squareroot` laws. We don't include the exponential law in this list as it has been shown to be a non-physical law in Espinoza & Jordan, 2016.

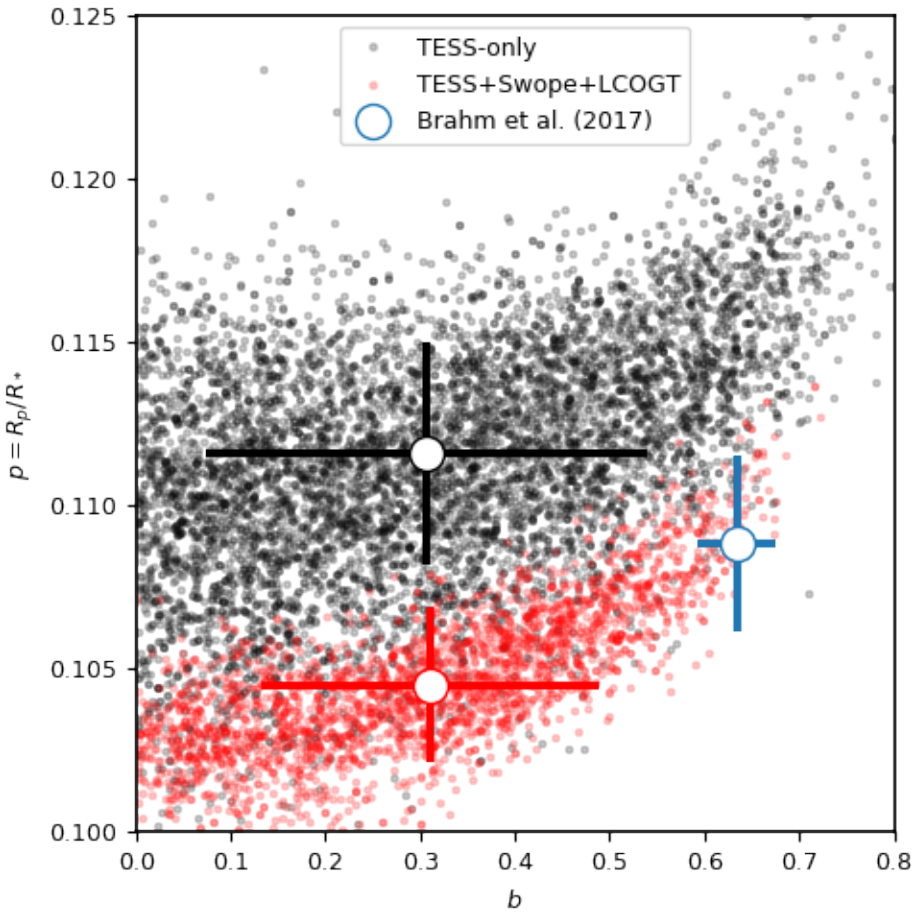
Let us test how the different laws do on the *TESS* dataset of HATS-46b. For this, let us fit the dataset with all the available limb-darkening laws and check the log-evidences, $\ln \mathcal{Z} = \ln \mathcal{P}(D|\text{Model})$ each model gives. Assuming all the models are equally likely, the different log-evidences can be transformed to *odds ratios* (i.e., the ratio of the probabilities of the models given the data, $\mathcal{P}(\text{Model}_i|D)/\mathcal{P}(\text{Model}_j|D)$) by simply substracting the log-evidences of the different models, i.e.,

$$\ln \frac{\mathcal{P}(\text{Model}_i|D)}{\mathcal{P}(\text{Model}_j|D)} = \ln \frac{\mathcal{P}(D|\text{Model}_i)}{\mathcal{P}(D|\text{Model}_j)} = \ln \frac{Z_i}{Z_j},$$

if $P(\text{Model}_i)/P(\text{Model}_j) = 1$. `juliet` also extracts the model evidences in the `results.posteriors` dictionary under the `lnZ` key; errors on this log-evidence calculation are under `lnZerr`. Let us compute the log-evidences for each limb-darkening law and compare them to see which one is the "best" in terms of this model comparison tool:

```
# Load Sector 1 data for HATS-46b again:
t, f, ferr  = juliet.get_TESS_data('https://archive.stsci.edu/hlsps/'+\
                                   'tess-data-alerts/hlsp_tess-data-'+\
                                   'alerts_tess_phot_00281541555-s02_'+\
```

(continues on next page)


TESS-only
TESS+Swope+LCOGT
Brahm et al. (2017)
0.125
0.120
0.115
0.110
0.105
0.100
$p = R_p/R_*$
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
b

(continued from previous page)

```
                                    'tess_v1_lc.fits')

# Put data arrays into dictionaries so we can fit it with juliet:
times, fluxes, fluxes_error = {},{},{}
times['TESS'], fluxes['TESS'], fluxes_error['TESS'] = t,f,ferr

# Define limb-darkening laws to test:
ld_laws = ['linear','quadratic','logarithmic','squareroot']

for ld_law in ld_laws:
    priors = {}
    # If law is not the linear, set priors for q1 and q2. If linear, set only for q1:
    if ld_law != 'linear':
        params = ['P_p1','t0_p1','r1_p1','r2_p1','q1_TESS','q2_TESS','ecc_p1','omega_
↪p1',\
                  'rho', 'mdilution_TESS', 'mflux_TESS', 'sigma_w_TESS']

        dists = ['normal','normal','uniform','uniform','uniform','uniform','fixed',
↪'fixed',\
                     'loguniform', 'fixed', 'normal', 'loguniform']

        hyperps = [[4.7,0.1], [1358.4,0.1], [0.,1], [0.,1.], [0., 1.], [0., 1.], 0.0,
↪ 90.,\
                       [100., 10000.], 1.0, [0.,0.1], [0.1, 1000.]]
    else:
        params = ['P_p1','t0_p1','r1_p1','r2_p1','q1_TESS','ecc_p1','omega_p1',\
                  'rho', 'mdilution_TESS', 'mflux_TESS', 'sigma_w_TESS']

        dists = ['normal','normal','uniform','uniform','uniform','fixed','fixed',\
                     'loguniform', 'fixed', 'normal', 'loguniform']

        hyperps = [[4.7,0.1], [1358.4,0.1], [0.,1], [0.,1.], [0., 1.], 0.0, 90.,\
                       [100., 10000.], 1.0, [0.,0.1], [0.1, 1000.]]

    for param, dist, hyperp in zip(params, dists, hyperps):
        priors[param] = {}
        priors[param]['distribution'], priors[param]['hyperparameters'] = dist,␣
↪hyperp

    dataset = juliet.load(priors=priors, t_lc = times, y_lc = fluxes, \
                          yerr_lc = fluxes_error, out_folder = 'hats46-'+ld_law, \
                          ld_laws = ld_law)

    results = dataset.fit()
    print("lnZ for "+ld_law+" limb-darkening law is: ",results.posteriors['lnZ']\
                                                ,"+-",results.posteriors['lnZerr'])
```

In our runs this gave:

```
lnZ for linear limb-darkening law is:      64202.653 +- 0.040
lnZ for quadratic limb-darkening law is:   64202.182 +- 0.018
lnZ for logarithmic limb-darkening law is: 64202.652 +- 0.077
lnZ for squareroot limb-darkening law is:  64202.786 +- 0.041
```

At face value, the model with the largest log-evidence is the square-root law, whereas the one with the lowest log-evidence is the quadratic law. However, the difference between those two log-evidences is very small: only $\Delta \ln Z = 0.60$ in favor of the square-root law, or an odds ratio between those laws of $\exp(\Delta \ln Z) \approx 2$ — given the data, the

square-root law model is only about two times more likely than the quadratic law. Not much, to be honest — I wouldn't bet my money on the quadratic law being wrong, so our assumption of a quadratic limb-darkening law in our analyses above seems to be very good. It is unlikely more complex limb-darkening laws would have given better results, by the way: note how the simpler linear law is basically equally likely to the square-root law ($\exp(\Delta \ln Z) \approx 1$).

What if more than one instrument is being fit; how do we define limb-darkening laws for each instrument? The `ld_laws` flag can also take as input a comma-separated string where one indicates the law to be used for each instrument in the form `instrument-ldlaw`. For example, if we wanted to fit the TESS, LCOGT and Swope data and define a square-root law for the former and logarithmic law for the other instruments, we would do (assuming we have already loaded the data and priors to the `priors`, `times`, `fluxes` and `fluxes_error` dictionaries):

```
dataset = juliet.load(priors=priors, t_lc = times, y_lc = fluxes, \
                      yerr_lc = fluxes_error,\
                      ld_laws = 'TESS-squareroot,LCOGT-logarithmic,SWOPE-logarithmic')

results = dataset.fit()
```

CHAPTER 6

Fitting radial-velocities

In `juliet`, the radial-velocity model is essentially the same as the one already introduced for the lightcurve in the *Lightcurve fitting with juliet* tutorial, i.e., in the absence of extra linear terms (see *Incorporating linear models*), is of the form (see Section 2 of the juliet paper)

$\mathcal{M}_i(t) + \epsilon_i(t)$,

where $\epsilon_i(t)$ is a noise model for instrument i (which as for the *Lightcurve fitting with juliet* tutorial, here we assume is white-gaussian noise — i.e., we assume $\epsilon_i(t) \sim \mathcal{N}(0, \sqrt{\sigma(t)^2 + \sigma_{w,i}})$, where $\sigma^2_{w,i}$ is a jitter term added to each instrument — we extend this to gaussian processes in the *Incorporating Gaussian Processes* tutorial), and $\mathcal{M}_i(t)$ is the deterministic part of the radial-velocity model for the instrument. The form of this deterministic part of the model is given by

$\mathcal{M}_i(t) = \mathcal{K}(t) + \mu_i + Q(t - t_a)^2 + A(t - t_a) + B.$

Here, $\mathcal{K}(t)$ is a Keplerian model which models the RV perturbations on the star due to the planets orbiting around it, μ_i is the RV of the star as measured by instrument i and the coefficients Q, A and B define an additional long-term trend useful for modelling long-period signals in the RVs that might not be well modelled by an additional Keplerian signal — t_a is just an arbitrary value substracted to the input times for numerical stability of the coefficients (by default $t_a = 2458460$ — but this can be defined by the user). By default, no long-term trend is incorporated in the models (i.e., $Q = A = B = 0$).

6.1 RV fits

To showcase the capabilities `juliet` has for radial-velocity fitting, here we will analyze the radial-velocities of the TOI-141 system (Espinoza et al. (2019)). We already analyzed the transits of this object in the *Getting started* tutorial; here we use the radial-velocities (RVs) of this system as it was shown that not only the signal of the transiting planet was present in the RVs, but there is also evidence for _another_ planet in the system. We have uploaded the dataset in a `juliet`-friendly format [here].

Let us first try to find the RV signature of the transiting planet analyzed in the *Getting started* tutorial in this dataset. From that analysis, the period is $P = 1.007917 \pm 0.000073$ days and the time-of-transit center is $t0 = 2458325.5386 \pm 0.0011$. Let us use these as priors for a first fit to the data — let us in turn assume uniform wide priors for the systemic velocities for each instrument μ_i, jitter terms and RV semi-amplitude; let us also fix the eccentricity to zero for now:

```
import juliet
priors = {}

# Name of the parameters to be fit:
params = ['P_p1','t0_p1','mu_CORALIE14', \
          'mu_CORALIE07','mu_HARPS','mu_FEROS',\
          'K_p1', 'ecc_p1', 'omega_p1', 'sigma_w_CORALIE14','sigma_w_CORALIE07',\
           'sigma_w_HARPS','sigma_w_FEROS']

# Distributions:
dists = ['normal','normal','uniform', \
         'uniform','uniform','uniform',\
         'uniform','fixed', 'fixed', 'loguniform', 'loguniform',\
         'loguniform', 'loguniform']

# Hyperparameters
hyperps = [[1.007917,0.000073], [2458325.5386,0.0011], [-100,100], \
           [-100,100], [-100,100], [-100,100], \
           [0.,100.], 0., 90., [1e-3, 100.], [1e-3, 100.], \
           [1e-3, 100.], [1e-3, 100.]]

# Populate the priors dictionary:
for param, dist, hyperp in zip(params, dists, hyperps):
    priors[param] = {}
    priors[param]['distribution'], priors[param]['hyperparameters'] = dist, hyperp

dataset = juliet.load(priors = priors, rvfilename='rvs_toi141.dat', out_folder =
↪'toi141_rvs')
results = dataset.fit(n_live_points = 300)
```

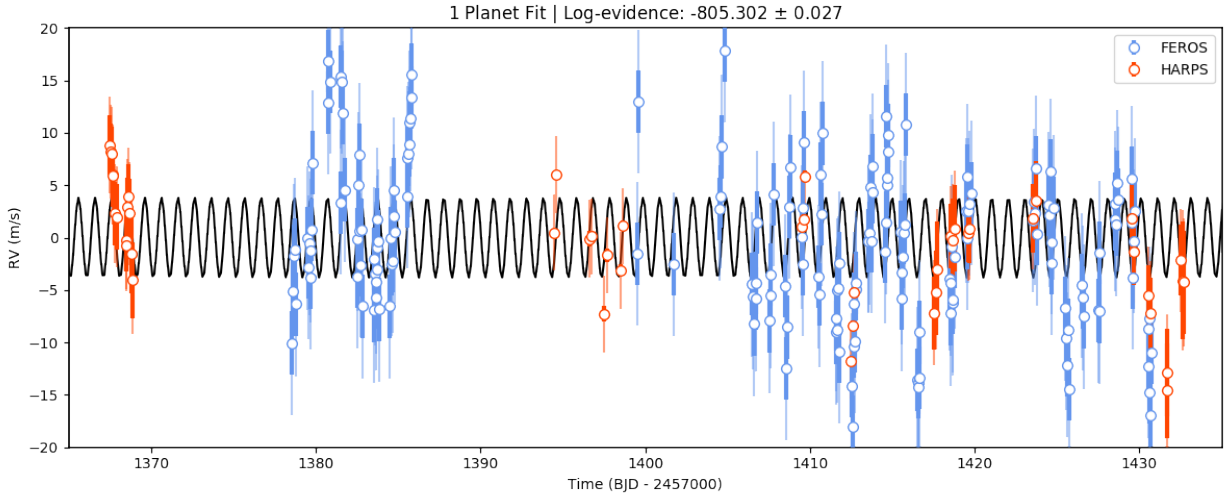
To plot the data, one can extract the models in an analogous fashion as we did for the *Lightcurve fitting with juliet* tutorial: we use the `results.rv.evaluate()` function. As with the `results.lc.evaluate()` function presented in the *Lightcurve fitting with juliet* tutorial, the function receives an instrument name and optionally times in which one wants to evaluate the model. Because each of the RV model parts are additive, it is easy to extract, e.g., the systemic-velocity corrected keplerian signal by simply evaluating the model in an arbitrary instrument and substracting the median of the systemic-velocity for that instrument. Let us do this to plot the above defined fit to see how we did — we'll only plot the HARPS and FEROS data, as the CORALIE data is not very constraining:

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

# Plot HARPS and FEROS datasets in the same panel. For this, first select any
# of the two and substract the systematic velocity to get the Keplerian signal.
# Let's do it with FEROS. First generate times on which to evaluate the model:
min_time, max_time = np.min(dataset.times_rv['FEROS'])-30,\
                     np.max(dataset.times_rv['FEROS'])+30

model_times = np.linspace(min_time,max_time,1000)

# Now evaluate the model in those times, and substract the systemic-velocity to
# get the Keplerian signal:
keplerian = results.rv.evaluate('FEROS', t = model_times) - \
            np.median(results.posteriors['posterior_samples']['mu_FEROS'])

# Now plot the (systematic-velocity corrected) RVs:
fig = plt.figure(figsize=(12,5))
instruments = ['FEROS','HARPS']
```

(continues on next page)

(continued from previous page)

```
colors = ['cornflowerblue','orangered']
for i in range(len(instruments)):
    instrument = instruments[i]
    # Evaluate the median jitter for the instrument:
    jitter = np.median(results.posteriors['posterior_samples']['sigma_w_'+instrument])
    # Evaluate the median systemic-velocity:
    mu = np.median(results.posteriors['posterior_samples']['mu_'+instrument])
    # Plot original data with original errorbars:
    plt.errorbar(dataset.times_rv[instrument]-2457000,dataset.data_rv[instrument]-mu,\
                 yerr = dataset.errors_rv[instrument],fmt='o',\
                 mec=colors[i], ecolor=colors[i], elinewidth=3, mfc = 'white', \
                 ms = 7, label=instrument, zorder=10)

    # Plot original errorbars + jitter (added in quadrature):
    plt.errorbar(dataset.times_rv[instrument]-2457000,dataset.data_rv[instrument]-mu,\
                 yerr = np.sqrt(dataset.errors_rv[instrument]**2+jitter**2),fmt='o',\
                 mec=colors[i], ecolor=colors[i], mfc = 'white', label=instrument,\
                 alpha = 0.5, zorder=5)

# Plot Keplerian model:
plt.plot(model_times-2457000, keplerian,color='black',zorder=1)
plt.ylabel('RV (m/s)')
plt.xlabel('Time (BJD - 2457000)')
plt.title('1 Planet Fit | Log-evidence: {0:.3f} $\pm$ {1:.3f}'.format(results.
↪posteriors['lnZ'],\
       results.posteriors['lnZerr']))
plt.ylim([-20,20])
plt.xlim([1365,1435])
```



Interesting. We have plotted both the original data with the original errorbars, and the errorbars enlarged by the best-fit jitter term. Note how the jitter is large (specially for HARPS)? This is to explain the large variations that appear in this 1-planet-fit result. Could this be due to an additional planet? To test this hypothesis, let's try another fit but now fitting for *two* planets: the 1-day transiting one, and an additional one with an unknown period from, say, 1 to 10 days. To do this, add the extra priors for this model first:

```
# Add second planet to the prior:
params = params + ['P_p2',   't0_p2',  'K_p2',    'ecc_p2','omega_p2']
```

(continues on next page)

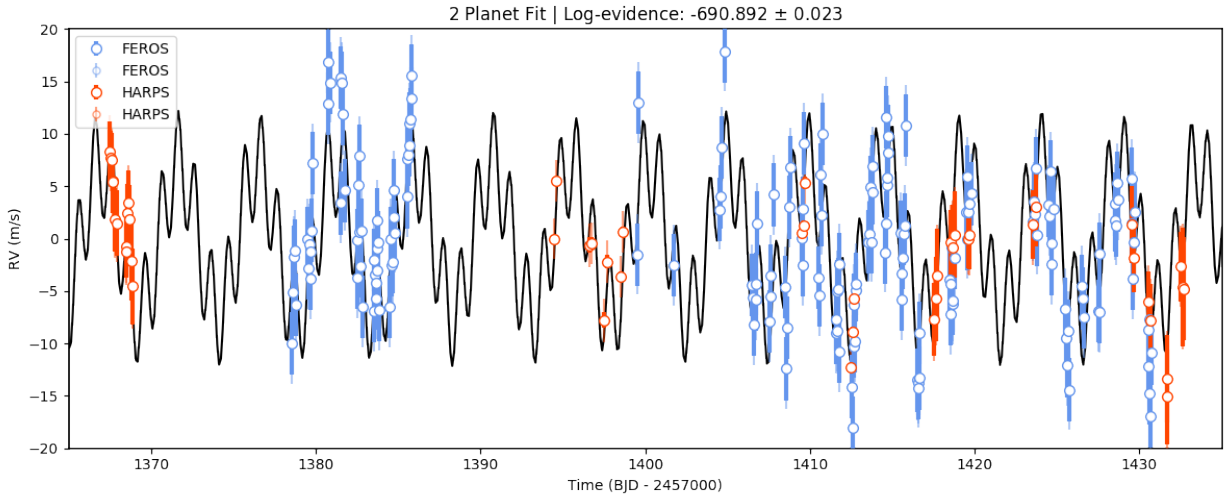
(continued from previous page)

```
dists = dists +   ['uniform','uniform','uniform', 'fixed', 'fixed']
hyperps = hyperps + [[1.,10.],[2458325.,2458330.],[0.,100.], 0., 90.]

# Repopulate priors dictionary:
priors = {}

for param, dist, hyperp in zip(params, dists, hyperps):
    priors[param] = {}
    priors[param]['distribution'], priors[param]['hyperparameters'] = dist, hyperp
```

And let's perform the second `juliet` fit with this two-planet system:

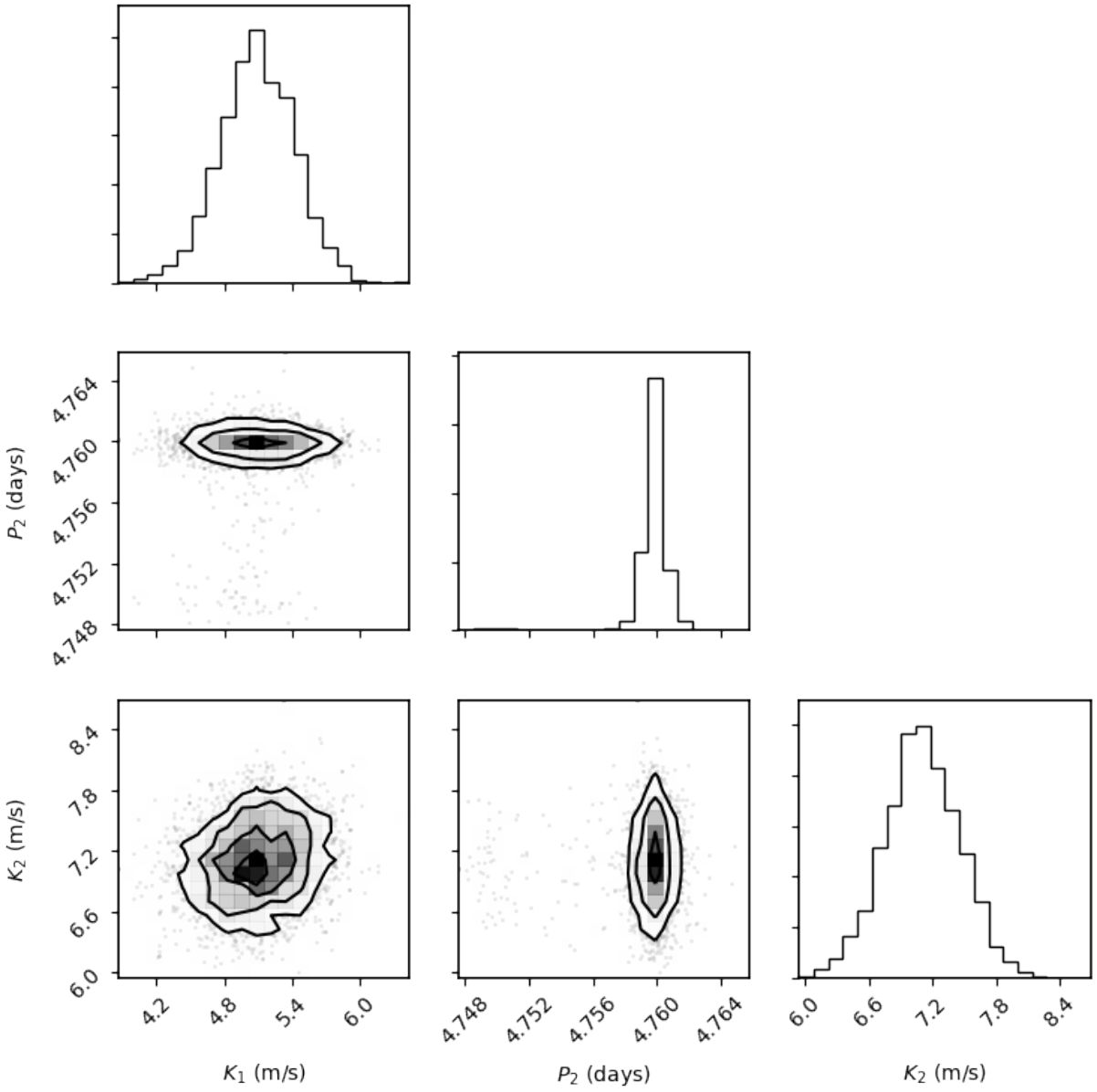
```
dataset = juliet.load(priors = priors, rvfilename='rvs_toi141.dat', out_folder =
↪'toi141_rvs_2planets')
results2 = dataset.fit(n_live_points = 300)
```

Repeating the same plot as above we find:



Woah! Much better fit to the data. Note also that we have plotted the log-evidences that `juliet` gives for these models — and the log-evidence for the 2-planet model is much larger than the one for the 1-planet model, $\Delta \ln Z = 114.4$ which is a *huge* odds ratio in favor of the two-planet model. Let's plot the posterior distributions for the parameters of this fit using Daniel Foreman-Mackey's corner package:

```
import corner

posterior_names = [r"$K_1$ (m/s)", r"$P_2$ (days)", r"$K_2$ (m/s)"]
first_time = True
for i in range(len(params)):
    if dists[i] != 'fixed' and params[i] != 'P_p1' and 't0' not in params[i] and \
    params[i][0:2] != 'mu' and params[i][0:5] != 'sigma':
        if first_time:
            posterior_data = results2.posteriors['posterior_samples'][params[i]]
            first_time = False
        else:
            posterior_data  = np.vstack((posterior_data, results2.posteriors[
↪'posterior_samples'][params[i]]))
posterior_data = posterior_data.T
figure = corner.corner(posterior_data, labels = posterior_names)
```

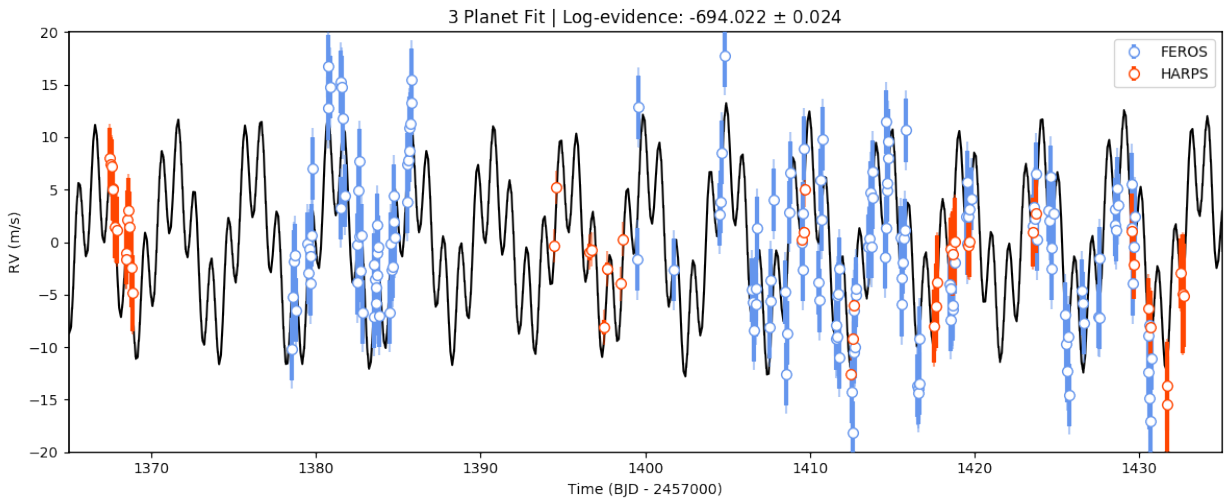

P_2 (days)
4.764
4.760
4.756
4.752
4.748
K_2 (m/s)
8.4
7.8
7.2
6.6
6.0
4.2
4.8
5.4
6.0
K_1 (m/s)
4.748
4.752
4.756
4.760
4.764
P_2 (days)
6.0
6.6
7.2
7.8
8.4
K_2 (m/s)

Best-fit period of this second planet is at 4.76 days — this is slightly off with the value cited in the paper (which is 4.78503 ± 0.0005), we will touch on this "mistery" in the *Joint transit and radial-velocity fits* tutorial. The semi-amplitudes mostly agree with the values in the paper. Judging from the errorbars, it seems there still is *some* unexplained variance in the data. Could it be an additional planet? Let us try fitting an extra planet — this time we will try a larger prior for the period of this third signal, going all the way from 1 to 40 days, which is about half the observing window for the FEROS and HARPS observations, which are the most constraining ones:

```
# Add third planet to the prior:
params3pl = params + ['P_p3',   't0_p3',  'K_p3',    'ecc_p3','omega_p3']
dists3pl = dists +   ['uniform','uniform','uniform', 'fixed', 'fixed']
hyperps3pl = hyperps + [[1.,40.],[2458325.,2458355.],[0.,100.], 0., 90.]

# Repopulate priors dictionary:
priors3pl = {}

for param, dist, hyperp in zip(params3pl, dists3pl, hyperps3pl):
    priors3pl[param] = {}
    priors3pl[param]['distribution'], priors3pl[param]['hyperparameters'] = dist,␣
↪hyperp

# Run juliet:
dataset = juliet.load(priors = priors3pl, rvfilename='rvs_toi141.dat', out_folder =
↪'toi141_rvs_3planets')
results = dataset.fit(n_live_points = 300)
```

The resulting fit doesn't look too different from the 2-planet one:

```
keplerian = results.rv.evaluate('FEROS', t = model_times) - \
            np.median(results.posteriors['posterior_samples']['mu_FEROS'])

# Now plot the (systematic-velocity corrected) RVs:
instruments = ['FEROS','HARPS']
colors = ['cornflowerblue','orangered']
fig = plt.figure(figsize=(12,5))
for i in range(len(instruments)):
    instrument = instruments[i]
    jitter = np.median(results.posteriors['posterior_samples']['sigma_w_'+instrument])
    mu = np.median(results.posteriors['posterior_samples']['mu_'+instrument])
    # Plot original errorbars:
    plt.errorbar(dataset.times_rv[instrument]-2457000,dataset.data_rv[instrument]-mu,\
                 yerr = dataset.errors_rv[instrument],fmt='o',\
                 mec=colors[i], ecolor=colors[i], elinewidth=3, mfc = 'white', \
                 ms = 7, label=instrument, zorder=10)
    # Plot original errorbars + jitter:
    plt.errorbar(dataset.times_rv[instrument]-2457000,dataset.data_rv[instrument]-mu,\
                 yerr = np.sqrt(dataset.errors_rv[instrument]**2+jitter**2),fmt='o',\
                 mec=colors[i], ecolor=colors[i], mfc = 'white', label=None,\
                 alpha = 0.5, zorder=5)

plt.plot(model_times-2457000, keplerian,color='black',zorder=1)
plt.ylabel('RV (m/s)')
plt.xlabel('Time (BJD - 2457000)')
plt.title('3 Planet Fit | Log-evidence: {0:.3f} $\pm$ {1:.3f}'.format(results.
↪posteriors['lnZ'],\
          results.posteriors['lnZerr']))
plt.ylim([-20,20])
plt.xlim([1365,1435])
plt.legend()
```



In fact, the evidence is *worse* in this 3-planet fit ($\ln Z_3 = -694$) than in the 2-planet fit ($\ln Z_2 = -691$). If both models were equiprobable a-priori, these log-evidences mean that, given the data, the 2-planet model is about 20 times more likely than the 3-planet model. So it seems that if there is some extra variance in the dataset, given the data at hand, this cannot be explained by an extra, third planetary signal alone — at least not with periods between 1 and 40 days. But what if there is a *longer* period planet creating a trend in the data? We deal with this possibility next

6.2 Long-term trends in RV data

As mentioned above, within `juliet` it is possible to fit for a long-term trend in the data that is common to all the instruments, parametrized by an intercept B (`rv_intercept` parameter within `juliet`), a slope A (`rv_slope` parameter within `juliet`) and a quadratic coefficient Q (`rv_quad` parameter within `juliet`). This long-term trend is useful to constrain signals whose periods might be longer than the current time baseline, which might *locally* appear as long-term trends. To fit those to the data, we just need to define priors for these parameters — let us do this with the TOI-141 dataset by first trying to fit a simple linear term (i.e., let us define only the parameters `rv_intercept` and `rv_slope`). Let us give wide uniform priors for those, join those priors to the 2-planet-fit priors and perform the fit:

```
# Add linear trend to the prior:
paramsLT = params + ['rv_intercept',   'rv_slope']
distsLT = dists +   ['uniform','uniform']
hyperpsLT = hyperps + [[-100.,100.],[-100., 100.]]

# Repopulate priors dictionary:
priorsLT = {}

for param, dist, hyperp in zip(paramsLT, distsLT, hyperpsLT):
    priorsLT[param] = {}
    priorsLT[param]['distribution'], priorsLT[param]['hyperparameters'] = dist, hyperp

# Run juliet:
dataset = juliet.load(priors = priorsLT, rvfilename='rvs_toi141.dat', out_folder =
↪'toi141_rvs_lineartrend')
results = dataset.fit(n_live_points = 300)
```

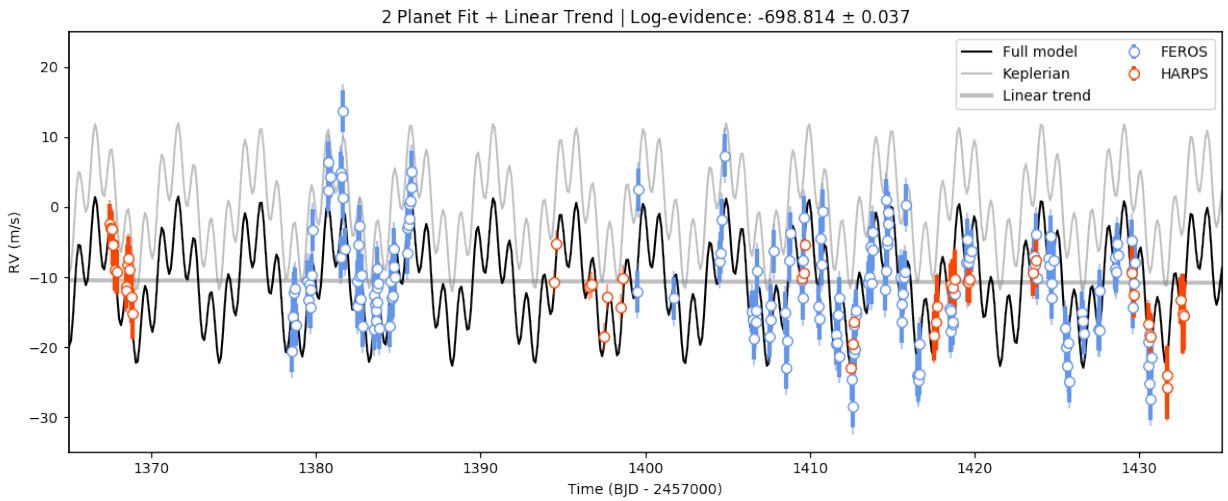
Before plotting the results, note that when we evaluate the model using `results.rv.evaluate` we will get back the *full* model — that is, a Keplerian *plus* the long-term trend model in our case (plus the systemic velocity of the instrument). However, one can pass an extra flag to this function, the `return_components` flag, which in addition

to the full model returns a dictionary that will have all the (deterministic) components of the model. Let us plot all the components of the model on top of each other using this flag:

```
# Return full model and the components of the model:
full_model, components = results.rv.evaluate('FEROS', t = model_times, return_
↪components = True)
# Substract systemic RV from full model (note this is part of the components):
full_model -= components['mu']

# Now plot the (systematic-velocity corrected) RVs (same code as above):
instruments = ['FEROS','HARPS']
colors = ['cornflowerblue','orangered']
fig = plt.figure(figsize=(12,5))
for i in range(len(instruments)):
    instrument = instruments[i]
    jitter = np.median(results.posteriors['posterior_samples']['sigma_w_'+instrument])
    mu = np.median(results.posteriors['posterior_samples']['mu_'+instrument])
    # Plot original errorbars:
    plt.errorbar(dataset.times_rv[instrument]-2457000,dataset.data_rv[instrument]-mu,\
                 yerr = dataset.errors_rv[instrument],fmt='o',\
                 mec=colors[i], ecolor=colors[i], elinewidth=3, mfc = 'white', \
                 ms = 7, label=instrument, zorder=10)
    # Plot original errorbars + jitter:
    plt.errorbar(dataset.times_rv[instrument]-2457000,dataset.data_rv[instrument]-mu,\
                 yerr = np.sqrt(dataset.errors_rv[instrument]**2+jitter**2),fmt='o',\
                 mec=colors[i], ecolor=colors[i], mfc = 'white', label=None,\
                 alpha = 0.5, zorder=5)

# Plot full model:
plt.plot(model_times-2457000, full_model,color='black',zorder=1, label = 'Full model')

# Extract model components and plot them:
plt.plot(model_times-2457000, components['keplerian'],color='grey',zorder=0, alpha=0.
↪5, label = 'Keplerian')
plt.plot(model_times-2457000, components['trend'],color='grey',zorder=0,alpha=0.5, lw␣
↪= 3, label = 'Linear trend')

# Labels:
plt.ylabel('RV (m/s)')
plt.xlabel('Time (BJD - 2457000)')
plt.title('2 Planet Fit + Linear Trend | Log-evidence: {0:.3f} $\pm$ {1:.3f}'.
↪format(results.posteriors['lnZ'],\
          results.posteriors['lnZerr']))
plt.ylim([-35,25])
plt.xlim([1365,1435])
plt.legend(ncol = 2)
```

As can be seen, the `components` dictionary extracted from the `results.rv.evaluate` function contains the Keplerian signal under `components['keplerian']`, and the trend under `components['keplerian']`. In addition, it also stores the Keplerians of each of the individual planets under `components['p1']` and `components['p2']` in our case. Note however, that the linear trend appears to not be significant in our case. So it might be that the unexplained variance could be explained by something else — in the *Incorporating Gaussian Processes* tutorial, we explore adding a Gaussian Process to the dataset in order to explain this.

Note: Note how in our case the `components` dictionary for the FEROS instrument has its systemic RV stored under `components['mu']`, which in general is *different* than taking the median of the `results.posteriors['posterior_samples']['mu_FEROS']` array. This is because, as was already mentioned



in the *Lightcurve fitting with juliet* tutorial, the `results.rv.evaluate` function (and the `results.lc.evaluate` function) evaluate the model by default on `nsamples = 1000` samples of the posterior. Thus, `components['mu']` is the median value of the systemic RV over the same `1000` samples as the other components, whereas `results.posteriors['posterior_samples']['mu_FEROS']` contains *all* the samples and thus, taking the median of that array should be slightly different than `components['mu']`. This difference, of course, is typically much smaller than the errors, so it shouldn't be a problem in general. One can set the `all_samples` flag to `True` in the `results.rv.evaluate` function to use all the samples — in this case, both should give the same results.

CHAPTER 7

Joint transit and radial-velocity fits

We have dealt so far separately between fitting transit lightcurves in the *Lightcurve fitting with juliet* tutorial and with fitting radial-velocity data in the *Fitting radial-velocities* tutorial. Here, we simply join what we have learned in those tutorials in order to showcase the ability of `juliet` to fit both dataset simultaneously.

In the background, `juliet` simply assumes both of these datasets are independant but that they can have common parameters. For example, the period and time-of-transit center are common to both datasets, but the radial-velocity semi-amplitude is only constrained by the radial-velocity dataset. Performing joint fits, thus, one can jointly extract information for common parameters between those datasets simultaneously in order to properly propagate that into the uncertainties and correlations between all the parameters being constrained.

Here, we use the TOI-141 dataset whose transit information was already presented in the quickstart section, and whose radial-velocity data was already presented in the *Fitting radial-velocities* section.

7.1 A joint fit to the TOI-141 system

In the *Fitting radial-velocities* tutorial, we have already seen how the RV data (which you can download from [here]) support the presence of at least two planets in the system, while in the quickstart section we have already seen how to fit a transit lightcurve for this system. Let us then simply join the prior distributions and data from these two sections into one. Let's first define the joint prior distribution:

```
# Define the master prior dictionary. First define the TRANSIT priors:
priors = {}

# Name of the parameters to be fit:
params = ['P_p1','t0_p1','r1_p1','r2_p1','q1_TESS','q2_TESS','ecc_p1','omega_p1',\
              'rho', 'mdilution_TESS', 'mflux_TESS', 'sigma_w_TESS']

# Distribution for each of the parameters:
dists = ['normal','normal','uniform','uniform','uniform','uniform','fixed','fixed',\
                'loguniform', 'fixed', 'normal', 'loguniform']

# Hyperparameters of the distributions (mean and standard-deviation for normal
```

(continues on next page)

(continued from previous page)

```
# distributions, lower and upper limits for uniform and loguniform distributions, and
# fixed values for fixed "distributions", which assume the parameter is fixed). Note␣
↪prior
# on t0 has an added 2457000 to convert from TESS JD to JD:
hyperps = [[1.,0.1], [2457000 + 1325.55,0.1], [0.,1], [0.,1.], [0., 1.], [0., 1.], 0.
↪0, 90.,\
                   [100., 10000.], 1.0, [0.,0.1], [0.1, 1000.]]

# Populate the priors dictionary:
for param, dist, hyperp in zip(params, dists, hyperps):
    priors[param] = {}
    priors[param]['distribution'], priors[param]['hyperparameters'] = dist, hyperp

# Now define the RV priors:
params = ['mu_CORALIE14', 'mu_CORALIE07','mu_HARPS','mu_FEROS', 'K_p1', 'sigma_w_
↪CORALIE14','sigma_w_CORALIE07',\
          'sigma_w_HARPS','sigma_w_FEROS','P_p2',   't0_p2',  'K_p2', 'ecc_p2',
↪'omega_p2']

# Distributions:
dists = ['uniform', 'uniform','uniform','uniform', 'uniform', 'loguniform',
↪'loguniform',\
         'loguniform', 'loguniform', 'uniform','uniform','uniform', 'fixed', 'fixed']

# Hyperparameters
hyperps = [[-100,100], [-100,100], [-100,100], [-100,100], [0.,100.], [1e-3, 100.],␣
↪[1e-3, 100.], \
           [1e-3, 100.], [1e-3, 100.], [1.,10.], [2458325.,2458330.], [0.,100.], 0.,␣
↪90.]

# Populate the priors dictionary:
for param, dist, hyperp in zip(params, dists, hyperps):
    priors[param] = {}
    priors[param]['distribution'], priors[param]['hyperparameters'] = dist, hyperp
```

Now let's get the transit data, load the radial-velocity data and priors into `juliet` and run the fit:

```
import juliet
import numpy as np

# First get TESS photometric data:
t,f,ferr  = juliet.get_TESS_data('https://archive.stsci.edu/hlsps/tess-data-alerts/'+\
                            'hlsp_tess-data-alerts_tess_phot_00403224672-'+\
                            's01_tess_v1_lc.fits')

# Put data in dictionaries, add 2457000 to the times to convert from TESS JD to JD:
times, fluxes, fluxes_error = {},{},{}
times['TESS'], fluxes['TESS'], fluxes_error['TESS'] = t + 2457000,f,ferr

# RV data is given in a file, so let's just pass the filename to juliet and load the␣
↪dataset:
dataset = juliet.load(priors=priors, t_lc = times, y_lc = fluxes, \
                  yerr_lc = fluxes_error, rvfilename='rvs_toi141.dat', \
                  out_folder = 'toi141_jointfit')

# And now let's fit it!
results = dataset.fit(n_live_points = 500)
```

We first should note that this fit has 21 (!) free parameters. Consequently, we have increased the number of live-points (with respect to other tutorials were we defined it to be 300) as there is a larger parameter space the live-points have to explore (for details on this, check Section 2.5 of the juliet paper and references therein). As a rule-of-thumb, live-points $n_{\rm live}$ should scale with about the square of the number of parameters n_p. In our case, $n_p = 21$ so $n_{\rm live} \sim n_p^2 = 440$ — we set it to 500 just to be on the safe side. Given the enlarged parameter space and number of live-points, the run will of course take longer to finish — in my laptop, this fit took about an hour.

Let's plot the *phased* transit lightcurve and radial-velocities of the planets in the same plot, so we can showcase some nice tricks that can be handy for dealing with the results provided by `juliet`. First, let us prepare the plot; we'll generate three panels. On the first we'll plot the phased transit lightcurve, and in the other two we'll plot the phased radial-velocities of the other planets:

```
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
fig = plt.figure(figsize=(14,4))
gs = gridspec.GridSpec(1, 3, width_ratios=[2,2,2])
```

Let's first deal with **the plot for the transiting planet lightcurve** (`p1`). For this one, we will not only plot the phased data and model, but will also bin the data so we can more easily see the transit event — to this end we will use the `juliet.bin_data` function, which given times/phases, data and a number of bins, will bin your data and return binned times/phases, data and errors:

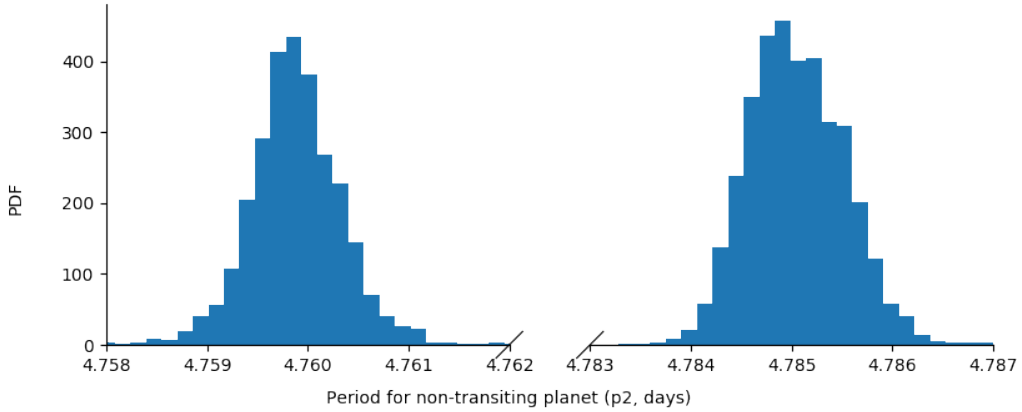
```
# Plot phased transit data and model first. Let's extract the transit
# model --- lightcurve is well sampled, so use the same input times to plot the model:
transit_model = results.lc.evaluate('TESS')

# Extract period and time-of-transit center for the planet:
P, t0 = np.median(results.posteriors['posterior_samples']['P_p1']),\
        np.median(results.posteriors['posterior_samples']['t0_p1'])

# Define plot, get phases, plot data and best-fit model:
ax1 = plt.subplot(gs[0])

phases = juliet.get_phases(dataset.times_lc['TESS'], P, t0)
idx = np.argsort(phases)
ax1.errorbar(phases, dataset.data_lc['TESS'], yerr= dataset.errors_lc['TESS'], fmt =
↪'.', alpha=0.1)
ax1.plot(phases[idx],transit_model[idx], color='black',zorder=10)

# Plot binned data as well, binning 40 datapoints in phase-space:
p_bin, y_bin, yerr_bin = juliet.bin_data(phases[idx], dataset.data_lc['TESS'][idx],␣
↪40)
ax1.errorbar(p_bin, y_bin, yerr = yerr_bin, fmt = 'o', mfc = 'white', mec = 'black',␣
↪ecolor = 'black')

# Labels, limits:
ax1.set_xlabel('Phases')
ax1.set_ylabel('Relative flux')
ax1.set_xlim([-0.06,0.06])
ax1.set_ylim([0.999,1.001])
```

Now, let's plot in the next panel **the radial-velocity data for this planet only**. For this, we will evaluate the radial-velocity model on times that provide a better sampling of the whole Keplerian curve. To "clean" the data from the other planetary and systematic components, we will also evaluate the model at the same times as the data and remove all components *but* the one from the planet. To this end, we will substract the planetary component to the full radial-velocity model, and substract that to the data:

```
# Define times on which we'll evaluate the model to plot:
min_time, max_time = np.min(dataset.times_rv['FEROS'])-30,\
                np.max(dataset.times_rv['FEROS'])+30
model_rv_times = np.linspace(min_time,max_time,1000)

# Evaluate RV model --- use all the posterior samples, also extract model components:
rv_model, components = results.rv.evaluate('FEROS', t = model_rv_times, all_samples =␣
↪True, \
                                            return_components = True)
# Substract FEROS systemic RV from rv_model:
rv_model -= components['mu']

# Define second panel in the plot:
ax2 = plt.subplot(gs[1])

# Iterate through the instruments, evaluate a model at those times, remove the full␣
↪model - planet component, so only the RV
# from the planet under study remain:
instruments = ['FEROS','HARPS']
colors = ['cornflowerblue','orangered']
for i in range(len(instruments)):
    instrument = instruments[i]
    # Evaluate jitter level --- will be added in quadrature to data errors:
    jitter = np.median(results.posteriors['posterior_samples']['sigma_w_'+instrument])
    # Get phases:
    phases = juliet.get_phases(dataset.times_rv[instrument], P, t0)
    # Plot data with the full model *minus* planet 1 substracted, so we see the␣
↪Keplerian of planet
    # 1 imprinted on the data. For this, evaluate model in the data-times first:
    c_model, c_components = results.rv.evaluate(instrument, t = dataset.times_
↪rv[instrument], \
                                                all_samples=True, return_components =␣
↪True)
    # Now plot RV data with (best model - planet component) substracted:
    ax2.errorbar(phases, dataset.data_rv[instrument]- (c_model - c_components['p1']),\
                 yerr = np.sqrt(dataset.errors_rv[instrument]**2+jitter**2),fmt='o',\
                 mec=colors[i], ecolor=colors[i], mfc = 'white', label=None,\
                 alpha = 0.5, zorder=5)

# Now plot the model for planet 1. First get phases of the model:
phases = juliet.get_phases(model_rv_times, P, t0)
# Plot phased model:
idx = np.argsort(phases)
plt.plot(phases[idx], components['p1'][idx], color='black', lw = 3, zorder=6)
# Define limits, labels:
ax2.set_xlim([-0.5,0.5])
ax2.set_ylim([-20,20])
ax2.set_xlabel('Phases')
ax2.set_ylabel('Radial-velocity (m/s)')
```

Now, finally, **we deal with the non-transiting planet** (`p2`). There is an interesting detail about this one, however. We already saw in the *Fitting radial-velocities* tutorial that there we obtained a period slightly different to the one that was published in the paper. Well, if you explore the posterior distribution of the period of this second planet with this joint-fit you will be able to see why: turns out there are actually *two* possible periods (one at 4.785 days and another one at 4.760 days):

I will let the reader find out for her/himself how we cracked this down in the paper, but turns out the real period is the one at 4.785 days (the other one is an alias).



So — how do we use all the posterior samples corresponding to *that* mode in order to plot the radial-velocity curve of this second planet? This is easily done with `juliet`, as one can directly give a posterior distribution dictionary to the `results.rv.evaluate` function using the `parameter_values` flag to evaluate your own custom posterior samples. Let's first find the indexes of all the samples that have periods larger than 4.77 days (so we capture the 4.785-day mode), and save all the posterior samples in a new dictionary, and use that to perform the same model evaluation and plotting as we did above for the transiting planet:

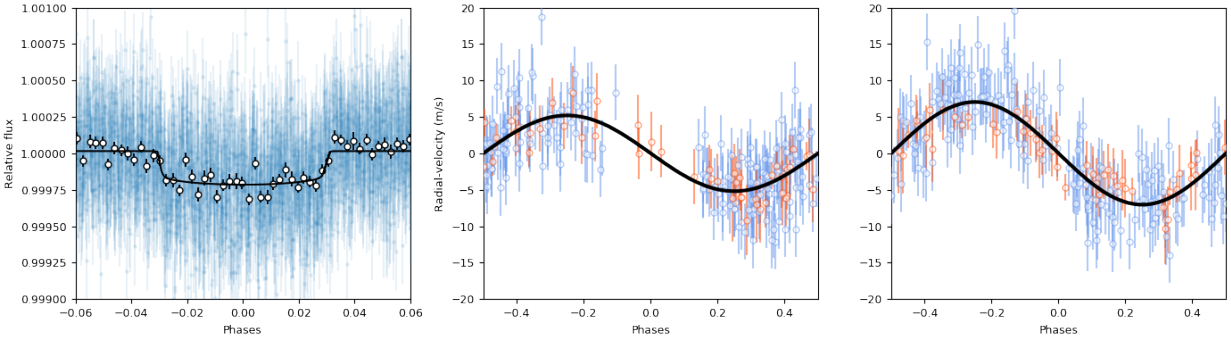
```
# First save all the samples from the mode of interest to a new dictionary:
idx_samples = np.where(results.posteriors['posterior_samples']['P_p2']>4.77)
# Create a "new posteriors" that uses only the samples from that mode:
new_posteriors = {}
for k in results.posteriors['posterior_samples'].keys():
    # We copy all the keys but the "unnamed" one --- we don't need that one.
    if k != 'unnamed':
        new_posteriors[k] = results.posteriors['posterior_samples'][k][idx_samples]

# Now extract the median period and time-of-transit center from this new dictionary:
P, t0 = np.median(new_posteriors['P_p2']),\
    np.median(new_posteriors['t0_p2'])

# And repeat the same as above to plot this second planet RV-curve in the third panel:
ax3 = plt.subplot(gs[2])
rv_model, components = results.rv.evaluate('FEROS', t = model_rv_times, all_samples =␣
↪True, \
                                           return_components = True, parameter_values␣
↪= new_posteriors)
rv_model -= components['mu']

# Loop over instruments, plot (model-planet)-substracted data:
for i in range(len(instruments)):
    instrument = instruments[i]
    # Extract jitters:
    jitter = np.median(new_posteriors['sigma_w_'+instrument])
    # Get phases:
    phases = juliet.get_phases(dataset.times_rv[instrument], P, t0)
    # Plot data with the full model *minus* planet 2 substracted, so we see the␣
↪Keplerian planet
    # 2 imprinted on the data:
    c_model, c_components = results.rv.evaluate(instrument, t = dataset.times_
↪rv[instrument], \
```

(continues on next page)

(continued from previous page)

```
                                                    all_samples=True, return_components =␣
↪True,\
                                                    parameter_values = new_posteriors)
    ax3.errorbar(phases, dataset.data_rv[instrument]-(c_model - c_components['p2']),\
                 yerr = np.sqrt(dataset.errors_rv[instrument]**2+jitter**2),fmt='o',\
                 mec=colors[i], ecolor=colors[i], mfc = 'white', label=None,\
                 alpha = 0.5, zorder=5)

# Plot planet 2 model:
phases = juliet.get_phases(model_rv_times, P, t0)
idx = np.argsort(phases)
ax3.plot(phases[idx], components['p2'][idx], color='black', lw = 3, zorder=6)
ax3.set_xlim([-0.5,0.5])
ax3.set_ylim([-20,20])
ax3.set_xlabel('Phases')
```

All this will give us the following nice plot:



CHAPTER 8

Incorporating linear models

In previous `juliet` tutorials for transits (*Lightcurve fitting with juliet*) and radial-velocities (*Fitting radial-velocities*), we have so far assumed that the only deterministic signals under consideration in the models $\mathcal{M}_i(t)$ for instrument i are composed of underlying physical processes. For transits, we assume the function is a transit model distorted both by a normalization constant and a dilution factor, whereas for the radial-velocities we assume this is an addition between a Keplerian signal, a systemic radial-velocity and a long-term trend. Typically, however, these are not the only components that make up a model. For transits, systematics in the data (e.g., airmass trends, meridian flips, etc.) can distort the signals further — for radial-velocities some linear models might help out constrain activity signals.

Within `juliet` one can model, in addition to the deterministic signal for transits and radial-velocities, $\mathcal{M}_i(t)$, a linear model such that the full data-generating process can be written as

$\mathcal{M}_i(t) + \mathrm{LM}_i(t) + \epsilon_i(t)$,

where the terms $\mathcal{M}_i(t)$ is the transit or radial-velocity model, $\epsilon_i(t)$ is the noise model (for details on those, see previous tutorials on transits and radial-velocities), and where $\mathrm{LM}_i(t)$ is a linear model given by:

$\mathrm{LM}_i(t) = \sum_{n=0}^{p_i} x_{n,i}(t)\theta_{n,i}^{\mathrm{LM}}$.

Here, the $x_{n,i}(t)$ are the $p_i + 1$ linear regressors at time t for instrument i, and the $\theta_{n,i}^{\mathrm{LM}}$ are the coefficients of those regressors (e.g., $x_{n,i}(t) = t^n$ would model a polynomial trend for instrument i).

8.1 Linear models in transit fits

Adding linear terms to a model within `juliet` is very simple, and can be done in two ways. One way is to simply pack the lightcurve and regressors in a text file of the form:

```
2458459.7999999998 1.0126748331 0.0030000000 CHAT 1.2107127967
2458459.8013377925 1.0127453892 0.0030000000 CHAT 1.2107915485
2458459.8026755853 1.0158682599 0.0030000000 CHAT 1.2108919775
2458459.8040133780 1.0117892069 0.0030000000 CHAT 1.2110140837
2458459.8053511707 1.0125201749 0.0030000000 CHAT 1.2111578671
2458459.8066889634 1.0133562197 0.0030000000 CHAT 1.2113233277
.
```

(continues on next page)

(continued from previous page)

```
.
.
```

where, the first column saves the times, second the relative fluxes, third errors on these relative fluxes, fourth the instrument names and the $p_i + 1$ subsequent columns store the $p_i + 1$ linear regressors to be fitted to the data (in the above example, 1). Once this file is created, the filename can be simply given to the `juliet.load` call with the `lcfilename` parameter — this will store the times, lightcurves and linear regressors in a given dataset. The second way is to simply pass all the linear regressors using the `linear_regressors_lc` variable of the `juliet.load` call — the input should be a dictionary, where each key is a different instrument and contains an array of dimensions $(N_i, p_i + 1)$, where N_i is the number of datapoints for instrument i. In this tutorial, we will use the former way of fitting linear models.

In this tutorial we will use the dataset uploaded [here] — this dataset has one linear regressor. For each linear regressor, we must define the prior for the coefficient $\theta_{n,i}$; these are expected to be of the form `thetaN_i`, where `N` is the numbering of the linear regressor (as given in the file or dictionary) and `i` is the instrument name. In our case, we have data from the *CHAT* telescope — let's fit it assuming a linear model:

```
import juliet
import numpy as np

priors = {}

# Name of the parameters to be fit:
params = ['P_p1','t0_p1','r1_p1','r2_p1','q1_CHAT','q2_CHAT','ecc_p1','omega_p1',\
             'rho', 'mdilution_CHAT', 'mflux_CHAT', 'sigma_w_CHAT', 'theta0_CHAT']

# Distributions:
dists = ['fixed','normal','uniform','uniform','uniform','uniform','fixed','fixed',\
                'loguniform', 'fixed', 'normal', 'loguniform', 'uniform']

# Hyperparameters
hyperps = [3.1, [2458460,0.1], [0.,1], [0.,1.], [0., 1.], [0., 1.], 0.0, 90.,\
                  [100., 10000.], 1.0, [0.,0.1], [0.1, 1000.],[-100,100]]

# Populate the priors dictionary:
for param, dist, hyperp in zip(params, dists, hyperps):
    priors[param] = {}
    priors[param]['distribution'], priors[param]['hyperparameters'] = dist, hyperp


# Load dataset:
dataset = juliet.load(priors=priors, lcfilename = 'lc_lm.dat', out_folder = 'lm_
↪transit_fit')
results = dataset.fit(n_live_points = 300)
```

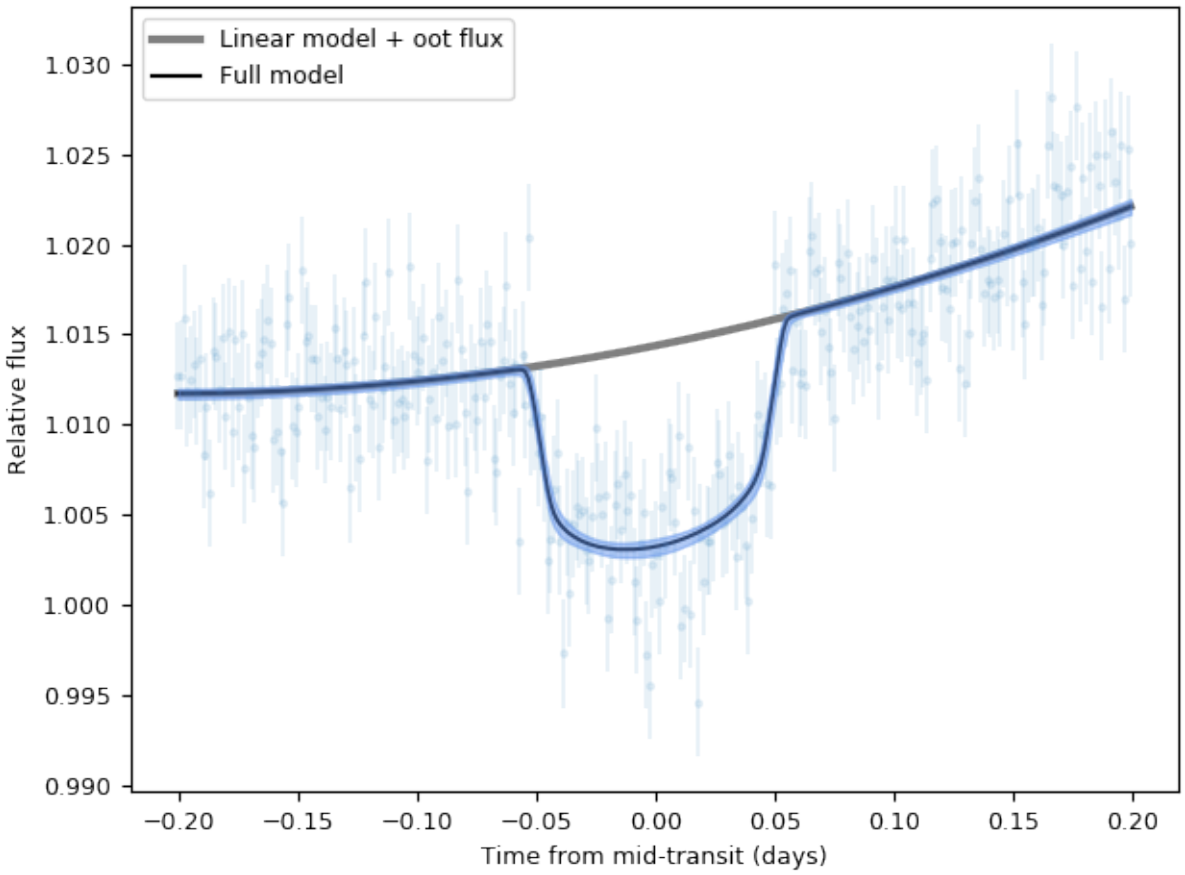
Now let's plot it:

```
t0 = np.median(results.posteriors['posterior_samples']['t0_p1'])

# Plot. First extract model:
transit_model, transit_up68, transit_low68, components  = results.lc.evaluate('CHAT',␣
↪return_err=True, \
                                                                                return_
↪components = True, \
                                                                                all_
↪samples = True)
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
plt.errorbar(dataset.times_lc['CHAT']-t0, dataset.data_lc['CHAT'], \
             yerr = dataset.errors_lc['CHAT'], fmt = '.' , alpha = 0.1)

# Out-of-transit flux:
oot_flux = np.median(1./(1. + results.posteriors['posterior_samples']['mflux_CHAT']))

# Plot non-transit model::
plt.plot(dataset.times_lc['CHAT']-t0, oot_flux + components['lm'], color='grey', lw =␣
↪3, label = 'Linear model + oot flux')
plt.plot(dataset.times_lc['CHAT']-t0, transit_model, color='black', label = 'Full␣
↪model')
plt.fill_between(dataset.times_lc['CHAT']-t0,transit_up68,transit_low68,\
                 color='cornflowerblue',alpha=0.5,zorder=5)

plt.xlabel('Time from mid-transit (days)')
plt.ylabel('Relative flux')
plt.legend()
```



CHAPTER 9

Incorporating Gaussian Processes

So far in the tutorials we have dealt with gaussian white-noise as a good approximation to the underlying signals present behind our transits and radial-velocities. However, this kind of process is very unrealistic for real data. Within `juliet`, we allow to model non-white noise models using Gaussian Proccesses (GPs), which are not only good for underlying stochastic processes that might be present in the data, but are also very good for modelling underlying deterministic processes for which we do not have a good model at hand. GPs attempt to model the likelihood, $\mathcal{L}$, as coming from a multi-variate gaussian distribution, i.e.,

$$\ln \mathcal{L} = -\frac{1}{2}\left[N \ln 2\pi + \ln |\mathbf{\Sigma}| + \vec{r}^{T}\mathbf{\Sigma}^{-1}\vec{r}\right],$$

where $\ln \mathcal{L}$ is the log-likelihood, N is the number of datapoints, $\mathbf{\Sigma}$ is a covariance matrix and $\vec{r}$ is the vector of the residuals (where each elements is simply our model — be it a lightcurve model or radial-velocity model — minus the data). A GP provides a form for the covariance matrix using so-called kernels which define its structure, and allow to efficiently fit for this underlying non-white noise structure. Within `juliet` we provide a wide variety of kernels which are implemented through george and celerite. In this tutorial we test their capabilities using real exoplanetary data!

9.1 Detrending lightcurves with GPs

A very popular use of GPs is to use them for "detrending" lightcurves. This means using the data outside of the feature of interest (e.g., a transit) in order to predict the behaviour of the lightcurve inside the feature and remove it, in order to facilitate or simplify the lightcurve fitting. To highlight the capabilities of `juliet`, here we will play around with *TESS* data obtained in Sector 1 for the HATS-46b system (Brahm et al., 2017). We already analyzed transits in Sector 2 for this system in the *Lightcurve fitting with juliet* tutorial, but here we will tackle Sector 1 data as the systematics in this sector are much stronger than the ones of Sector 2.

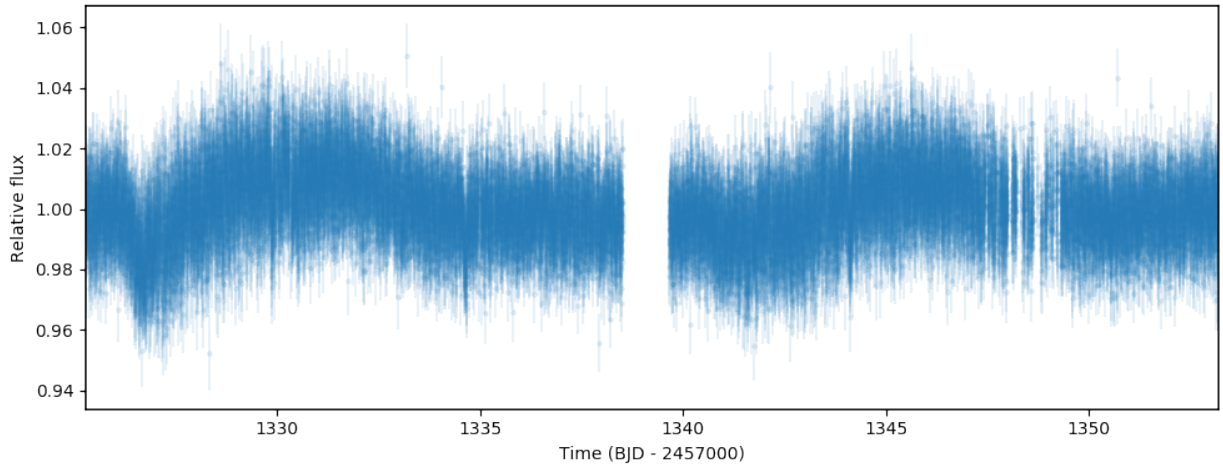
Let's start by downloading and plotting the *TESS* data for HATS-46b in Sector 1 using `juliet`:

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

# First, get arrays of times, normalized-fluxes and errors for HATS-46
```

(continues on next page)

(continued from previous page)

```
#from Sector 1 from MAST:
t, f, ferr  = juliet.get_TESS_data('https://archive.stsci.edu/hlsps/'+\
                                   'tess-data-alerts/hlsp_tess-data-'+\
                                   'alerts_tess_phot_00281541555-s01_'+\
                                   'tess_v1_lc.fits')
# Plot the data:
plt.errorbar(t,f,yerr=ferr,fmt='.')
plt.xlim([np.min(t),np.max(t)])
plt.xlabel('Time (BJD - 2457000)')
plt.ylabel('Relative flux')
```



As can be seen, the data has a fairly strong long-term trend going around. In fact, the trend is so strong that it is quite hard to see the transits by eye! Let us try to get rid of this trend by fitting a GP to the out-of-transit data, and then *predict* the in-transit flux with this model to remove these systematics in the data. Let us first isolate the out-of-transit data from the in-transit data using the ephemerides published in Brahm et al., 2017 — we know where the transits should be, so we will simply phase-fold the data and remove all datapoints out-of-transit (which judging from the plots in that paper, should be all points at absolute phases above 0.02). Let us save this out-of-transit data in dictionaries so we can feed them to `juliet`:

```
# Period and t0 from Anderson et al. (201X):
P,t0 =  4.7423729 ,  2457376.68539 - 2457000
# Get phases --- identify out-of-transit (oot) times by phasing the data
# and selecting all points at absolute phases larger than 0.02:
phases = juliet.utils.get_phases(t, P, t0)
idx_oot = np.where(np.abs(phases)>0.02)[0]
# Save the out-of-transit data into dictionaries so we can feed them to juliet:
times, fluxes, fluxes_error = {},{},{}
times['TESS'], fluxes['TESS'], fluxes_error['TESS'] = t[idx_oot],f[idx_oot],ferr[idx_
↪oot]
```

Now, let us fit a GP to this data. To do this, we will use a simple (approximate) Matern kernel, which was implemented via celerite and which can accomodate itself to both rough and smooth signals. On top of this, the selection was also made because this is implemented in `celerite`, which makes the computation of the log-likelihood blazing fast — this in turn speeds up the posterior sampling within `juliet`. The kernel is given by

$$k(\tau_{i,j}) = \sigma_{GP}^2 \tilde{M}(\tau_{i,j},\rho) + (\sigma_i^2 + \sigma_w^2)\delta_{i,j},$$

where $k(\tau_{i,j})$ gives the element i,j of the covariance matrix $\mathbf{\Sigma}$, $\tau_{i,j} = |t_i - t_j|$ with the t_i and t_j being the i and j GP regressors (typically — as in this case — the times), σ_i the errorbar of the i-th datapoint, σ_{GP} sets the amplitude (in

ppm) of the GP, σ_w (in ppm) is an added (unknown) *jitter* term, $\delta_{i,j}$ a Kronecker's delta (i.e., zero when $i \neq j$, one otherwise) and where

$$\tilde{M}(\tau_{i,j},\rho) = [(1+1/\epsilon)\exp(-[1-\epsilon]\sqrt{3}\tau/\rho) + (1-1/\epsilon)\exp(-[1+\epsilon]\sqrt{3}\tau/\rho)]$$

is the (approximate) Matern part of the kernel, which has a characteristic length-scale ρ.

To use this kernel within `juliet` you just have to give the priors for these parameters in the prior dictionary or file (see below for a full list of all the available kernels). `juliet` will automatically recognize which kernel you want based on the priors selected for each instrument. In this case, if you define a parameter `GP_sigma` (for σ_{GP}) and `rho` (for the Matern time-scale, ρ), `juliet` will automatically recognize you want to use this (approximate) Matern kernel. Let's thus give these priors — for now, let us set the dilution factor `mdilution` to 1, give a normal prior for the mean out-of-transit flux `mflux` and wide log-uniform priors for all the other parameters:

```
# Set the priors:
params =  ['mdilution_TESS', 'mflux_TESS', 'sigma_w_TESS', 'GP_sigma_TESS', \
           'GP_rho_TESS']
dists =   ['fixed',          'normal',     'loguniform',   'loguniform',\
           'loguniform']
hyperps = [1., [0.,0.1], [1e-6, 1e6], [1e-6, 1e6],\
           [1e-3,1e3]]

priors = {}
for param, dist, hyperp in zip(params, dists, hyperps):
    priors[param] = {}
    priors[param]['distribution'], priors[param]['hyperparameters'] = dist, hyperp

# Perform the juliet fit. Load dataset first (note the GP regressor will be the␣
↪times):
dataset = juliet.load(priors=priors, t_lc = times, y_lc = fluxes, \
                      yerr_lc = fluxes_error, GP_regressors_lc = times, \
                      out_folder = 'hats46_detrending')
# Fit:
results = dataset.fit()
```

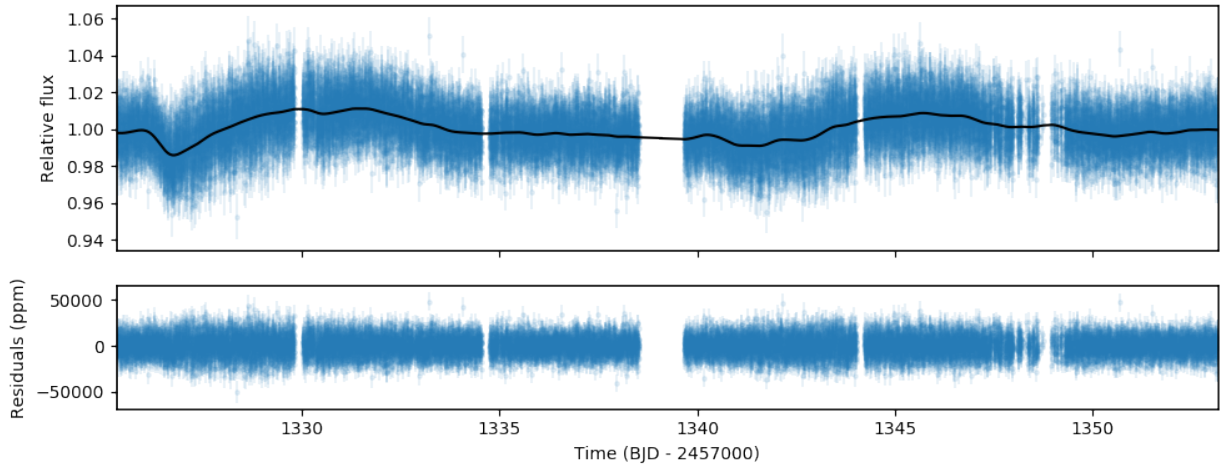
Note that the only new part in terms of loading the dataset is that one has to now add a new piece of data, the `GP_regressors_lc`, in order for the GP to run (emphasized in the code above). This is also a dictionary, which specifies the GP regressors for each instrument. For `celerite` kernels, in theory the regressors have to be one-dimensional and ordered in ascending or descending order — however, internally `juliet` performs this ordering so the user doesn't have to worry about this last part. Let us now plot the GP fit and some residuals below to see how we did:

```
# Import gridspec:
import matplotlib.gridspec as gridspec
# Get juliet model prediction for the full lightcurve:
model_fit = results.lc.evaluate('TESS')

# Plot:
fig = plt.figure(figsize=(10,4))
gs = gridspec.GridSpec(2, 1, height_ratios=[2,1])

# First the data and the model on top:
ax1 = plt.subplot(gs[0])
ax1.errorbar(times['TESS'], fluxes['TESS'], fluxes_error['TESS'],fmt='.',alpha=0.1)
ax1.plot(times['TESS'], model_fit, color='black', zorder=100)
ax1.set_ylabel('Relative flux')
ax1.set_xlim(np.min(times['TESS']),np.max(times['TESS']))
ax1.xaxis.set_major_formatter(plt.NullFormatter())
```

(continues on next page)

(continued from previous page)

```
# Now the residuals:
ax2 = plt.subplot(gs[1])
ax2.errorbar(times['TESS'], (fluxes['TESS']-model_fit)*1e6, \
             fluxes_error['TESS']*1e6,fmt='.',alpha=0.1)
ax2.set_ylabel('Residuals (ppm)')
ax2.set_xlabel('Time (BJD - 2457000)')
ax2.set_xlim(np.min(times['TESS']),np.max(times['TESS']))
```



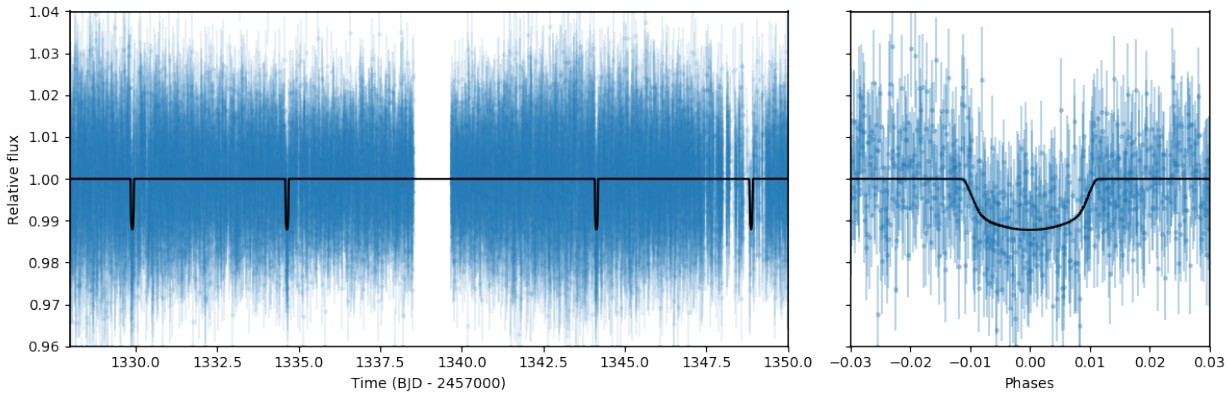
Seems we did pretty good! By default, the `results.lc.evaluate` function evaluates the model on the input dataset (i.e., on the input GP regressors and input times). In our case, this was the out-of-transit data. To detrend the lightcurve, however, we have to *predict* the model on the full time-series. This is easily done using the same function but giving the times and GP regressors we want to predict the data on. So let us detrend the original lightcurve (stored in the arrays `t`, `f` and `ferr` that we extracted at the beggining of this section), and fit a transit to it to see how we do:

```
# Get model prediction from juliet:
model_prediction = results.lc.evaluate('TESS', t = t, GPregressors = t)

# Repopulate dictionaries with new detrended flux:
times['TESS'], fluxes['TESS'], fluxes_error['TESS'] = t, f/model_prediction, \
                                                      ferr/model_prediction

# Set transit fit priors:
priors = {}

params = ['P_p1','t0_p1','r1_p1','r2_p1','q1_TESS','q2_TESS','ecc_p1','omega_p1',\
             'rho', 'mdilution_TESS', 'mflux_TESS', 'sigma_w_TESS']

dists = ['normal','normal','uniform','uniform','uniform','uniform','fixed','fixed',\
                'loguniform', 'fixed', 'normal', 'loguniform']

hyperps = [[4.7,0.1], [1329.9,0.1], [0.,1], [0.,1.], [0., 1.], [0., 1.], 0.0, 90.,\
                  [100., 10000.], 1.0, [0.,0.1], [0.1, 1000.]]

# Populate the priors dictionary:
for param, dist, hyperp in zip(params, dists, hyperps):
    priors[param] = {}
    priors[param]['distribution'], priors[param]['hyperparameters'] = dist, hyperp
```

(continues on next page)

(continued from previous page)

```
# Perform juliet fit:
dataset = juliet.load(priors=priors, t_lc = times, y_lc = fluxes, \
                  yerr_lc = fluxes_error, out_folder = 'hats46_detrended_transitfit')

results = dataset.fit()

# Extract transit model prediction given the data:
transit_model = results.lc.evaluate('TESS')

# Plot results:
fig = plt.figure(figsize=(10,4))
gs = gridspec.GridSpec(1, 2, width_ratios=[2,1])
ax1 = plt.subplot(gs[0])

# Plot time v/s flux plot:
ax1.errorbar(dataset.times_lc['TESS'], dataset.data_lc['TESS'], \
         yerr = dataset.errors_lc['TESS'], fmt = '.', alpha = 0.1)

ax1.plot(dataset.times_lc['TESS'], transit_model,color='black',zorder=10)

ax1.set_xlim([1328,1350])
ax1.set_ylim([0.96,1.04])
ax1.set_xlabel('Time (BJD - 2457000)')
ax1.set_ylabel('Relative flux')

# Now phased transit lightcurve:
ax2 = plt.subplot(gs[1])
ax2.errorbar(phases, dataset.data_lc['TESS'], \
             yerr = dataset.errors_lc['TESS'], fmt = '.', alpha = 0.1)
idx = np.argsort(phases)
ax2.plot(phases[idx],transit_model[idx], color='black',zorder=10)
ax2.yaxis.set_major_formatter(plt.NullFormatter())
ax2.set_xlim([-0.03,0.03])
ax2.set_ylim([0.96,1.04])
ax2.set_xlabel('Phases')
```



Pretty good! In the next section, we explore *joint* fitting for the transit model and the GP process.

9.2 Joint GP and lightcurve fits

One might wonder what the impact of doing the two-stage process mentioned above is when compared with fitting *jointly* the GP process and the transit model. This latter method, in general, seems more appealing because it can take into account in-transit non-white noise features, which in turn might give rise to more realistic errorbars on the retrieved planetary parameters. Within `juliet` performing this kind of model fit is fairly easy to do: one just has to add the priors for the GP process to the transit paramenters, and feed the GP regressors. Let us use the same GP kernel as in the previous section then to model the underlying process for HATS-46b *jointly* with the transit parameters:

```
# First define the priors:
priors = {}

# Same priors as for the transit-only fit, but we now add the GP priors:
params = ['P_p1','t0_p1','r1_p1','r2_p1','q1_TESS','q2_TESS','ecc_p1','omega_p1',\
          'rho', 'mdilution_TESS', 'mflux_TESS', 'sigma_w_TESS', \
          'GP_sigma_TESS', 'GP_rho_TESS']

dists = ['normal','normal','uniform','uniform','uniform','uniform','fixed','fixed',\
         'loguniform', 'fixed', 'normal', 'loguniform', \
         'loguniform', 'loguniform']

hyperps = [[4.7,0.1], [1329.9,0.1], [0.,1], [0.,1.], [0., 1.], [0., 1.], 0.0, 90.,\
           [100., 10000.], 1.0, [0.,0.1], [0.1, 1000.], \
           [1e-6, 1e6], [1e-3, 1e3]]

# Populate the priors dictionary:
for param, dist, hyperp in zip(params, dists, hyperps):
    priors[param] = {}
    priors[param]['distribution'], priors[param]['hyperparameters'] = dist, hyperp

times['TESS'], fluxes['TESS'], fluxes_error['TESS'] = t,f,ferr
dataset = juliet.load(priors=priors, t_lc = times, y_lc = fluxes, \
                      yerr_lc = fluxes_error, GP_regressors_lc = times, out_folder =
↪'hats46_transitGP', verbose = True)

results = dataset.fit()
```

Note that in comparison with the transit-only fit, we have just added the priors for the GP parameters (highlighted lines above). The model being fit in this case by `juliet` is the one given in Section 2 of the juliet paper, i.e., a model of the form

$\mathcal{M}_{\rm TESS}(t) + \epsilon(t)$,

where

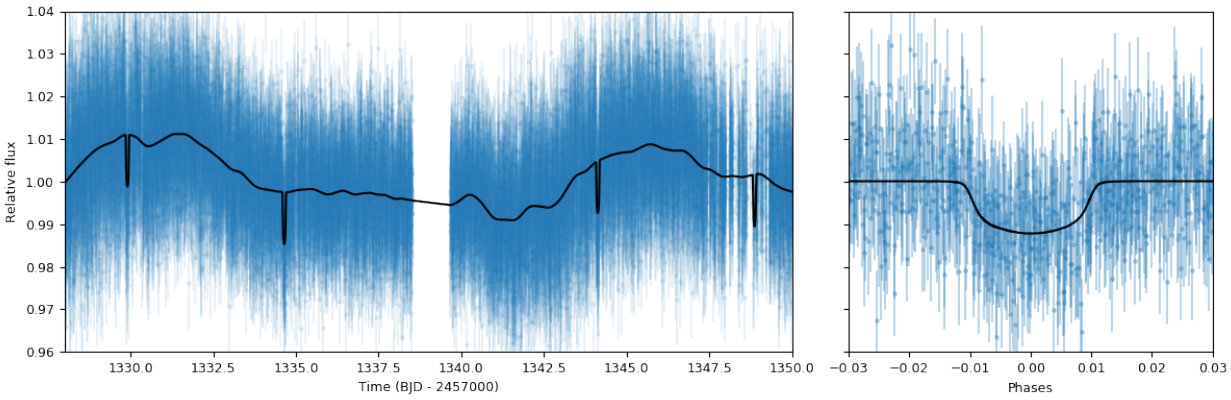
$\mathcal{M}_{\rm TESS}(t) = [\mathcal{T}_{\rm TESS}(t) D_{\rm TESS} + (1 - D_{\rm TESS})] \left(\frac{1}{1+D_{\rm TESS} M_{\rm TESS}} \right)$

is the photometric model composed of the dilution factor $D_{\rm TESS}$ (`mdilution_TESS`), the mean out-of-transit flux $M_{\rm TESS}$ (`mflux_TESS`) and the transit model for the instrument $\mathcal{T}_{\rm TESS}(t)$ (defined by the transit parameters and by the instrument-dependant limb-darkening parametrization given by `q1_TESS` and `q2_TESS`). This is the *deterministic* part of the model, as $\mathcal{M}_{\rm TESS}(t)$ is a process that, given a time and a set of parameters, will always be the same: you can easily evaluate the model from the above definition. $\epsilon(t)$, on the other hand, is the *stochastic* part of our model: a noise model which in our case is being modelled as a GP. Given a set of parameters and times for the GP model, the process *cannot* directly be evaluated because it defines a probability distribution, not a deterministic function like $\mathcal{M}_{\rm TESS}(t)$. This means that every time you sample from this GP, you would get a different curve — ours was just *one realization* of many possible ones. However, we do have a (noisy) realization (our data) and so our process can be constrained by it. This is what we plotted in the previous section of this tutorial (which in strict rigor is a filter). Also note that in this model the GP is an additive process.

Once the fit is done, `juliet` allows to retrieve (1) the full median posterior model (i.e., the deterministic part of the model **plus** the median GP process) via the `results.lc.evaluate()` function already used in the previous section and (2) all parts of the model separately via the `results.lc.model` dictionary, which holds the `deterministic` key which hosts the deterministic part of the model ($\mathcal{M}_{\rm TESS}(t)$) and the `GP` key which holds the stochastic part of the model ($\epsilon(t)$, constrained on the data). To show how this works, let us extract these components below in order to plot the full model, and remove the median GP process from the data in order to plot the ("systematics-corrected") phase-folded lightcurve:

```
# Extract full model:
transit_plus_GP_model = results.lc.evaluate('TESS')

# Deterministic part of the model (in our case transit divided by mflux):
transit_model = results.lc.model['TESS']['deterministic']

# GP part of the model:
gp_model = results.lc.model['TESS']['GP']

# Now plot. First preambles:
fig = plt.figure(figsize=(12,4))
gs = gridspec.GridSpec(1, 2, width_ratios=[2,1])
ax1 = plt.subplot(gs[0])

# Plot data
ax1.errorbar(dataset.times_lc['TESS'], dataset.data_lc['TESS'], \
             yerr = dataset.errors_lc['TESS'], fmt = '.', alpha = 0.1)

# Plot the (full, transit + GP) model:
ax1.plot(dataset.times_lc['TESS'], transit_plus_GP_model, color='black',zorder=10)

ax1.set_xlim([1328,1350])
ax1.set_ylim([0.96,1.04])
ax1.set_xlabel('Time (BJD - 2457000)')
ax1.set_ylabel('Relative flux')

ax2 = plt.subplot(gs[1])

# Now plot phase-folded lightcurve but with the GP part removed:
ax2.errorbar(phases, dataset.data_lc['TESS'] - gp_model, \
             yerr = dataset.errors_lc['TESS'], fmt = '.', alpha = 0.3)

# Plot transit-only (divided by mflux) model:
idx = np.argsort(phases)
ax2.plot(phases[idx],transit_model[idx], color='black',zorder=10)
ax2.yaxis.set_major_formatter(plt.NullFormatter())
ax2.set_xlabel('Phases')
ax2.set_xlim([-0.03,0.03])
ax2.set_ylim([0.96,1.04])
```

Looks pretty good! As can be seen, the `results.lc.model['TESS']['deterministic']` dictionary holds the deterministic part of the model. This includes the transit model which is distorted by the dilution factor (set to 1 in our case) and the mean out-of-transit flux, which we fit together with the other parameters in our joint fit — this deterministic model is the one that is plotted in the right panel in the above presented figure. The `results.lc.model['TESS']['GP']` dictionary, on the other hand, holds the GP part of the model — because this is an additive process in this case, we can just substract it from the data in order to get the "systematic-corrected" data that we plot in the right panel in the figure above.



9.3 Global and instrument-by instrument GP models

In the previous lightcurve analysis we dealt with GP models which are individually defined for each instrument. This means that even if the hyperparameters between the GPs (e.g., timescales) are shared between different instruments because we believe they might arise from the same parent physical process, we are modelling each instrument as if the data we observe in them was produced by a different realization from that GP. In some cases, however, we would want to model a GP which is *common* to all the instruments, i.e., a GP model whose realization gave rise to the data we see in *all* of our instruments simultaneously. Within `juliet`, we refer to those kind of models as *global* GP models. These are most useful in radial-velocity analyses, where an underlying physical signal might be common to all the instruments. For example, we might believe a given signal in our radial-velocity data is produced by stellar activity, and if all the instruments have similar bandpasses, then the amplitude, period and timescales are associated with the process itself and not with each instrument. Of course, one can still define different individual jitter terms for each instrument in this case.

In practice, as explained in detail in the Section 2 of the juliet paper, the difference between a **global** model and an **instrument-by-instrument** model is that for the former a unique covariance matrix (and set of GP hyperparameters) is defined for the problem. This means that the log-likelihood of a **global** model is written as presented at the introduction of this tutorial, i.e.,

$$\mathcal{L} = -\frac{1}{2}\left[N\ln 2\pi + \ln|\mathbf{\Sigma}| + \vec{r}^{T}\mathbf{\Sigma}^{-1}\vec{r}\right].$$

Here, N is the total number of datapoints considering all the instruments in the problem, $\mathbf{\Sigma}$ is the covariance matrix for that same full dataset and $\vec{r}$ is the vector of residuals for the same dataset. In the **instrument-by-instrument** type of models, however, a different covariance matrix (and thus different GP hyperparameters — which might be shared, as we'll see in a moment!) is defined for each instrument. The total log-likelihood of the problem is, thus, given by:

$$\mathcal{L} = \sum_i -\frac{1}{2}\left[N_i\ln 2\pi + \ln|\mathbf{\Sigma}_i| + \vec{r}_i^{T}\mathbf{\Sigma}_i^{-1}\vec{r}_i\right],$$

where N_i is the number of datapoints for instrument i, $\mathbf{\Sigma}_i$ is the covariance matrix for that instrument and $\vec{r}_i$ is the vector of residuals for that same instrument. The lightcurve examples above were instrument-by-instrument models, which makes sense because the instrumental systematics were individual to the TESS lightcurves — if we had to incorporate extra datasets, those would most likely have to have different GP hyperparameters (and, perhaps, kernels). Here, we will exemplify the difference between those two types of models using the radial-velocity dataset for TOI-141 already analyzed in the *Fitting radial-velocities* tutorial which can be downloaded from [here]. We will use the time as the GP regressor in our case; we have uplaoded a file containing those times [here].

Let us start by fitting a *global* GP model to that data. To this end, let's try to fit the same Matern kernel defined in the previous GP examples. To define a global GP model, for radial-velocity fits, one has to simply add `rv` instead of the instrument name to the GP hyperparameters:

```
import numpy as np
import juliet
priors = {}

# Name of the parameters to be fit:
params = ['P_p1','t0_p1','mu_CORALIE14', \
          'mu_CORALIE07','mu_HARPS','mu_FEROS',\
          'K_p1', 'ecc_p1', 'omega_p1', 'sigma_w_CORALIE14','sigma_w_CORALIE07',\
           'sigma_w_HARPS','sigma_w_FEROS','GP_sigma_rv','GP_rho_rv']

# Distributions:
dists = ['normal','normal','uniform', \
         'uniform','uniform','uniform',\
         'uniform','fixed', 'fixed', 'loguniform', 'loguniform',\
         'loguniform', 'loguniform','loguniform','loguniform']

# Hyperparameters
hyperps = [[1.007917,0.000073], [2458325.5386,0.0011], [-100,100], \
           [-100,100], [-100,100], [-100,100], \
           [0.,100.], 0., 90., [1e-3, 100.], [1e-3, 100.], \
           [1e-3, 100.], [1e-3, 100.],[0.01,100.],[0.01,100.]]

# Populate the priors dictionary:
for param, dist, hyperp in zip(params, dists, hyperps):
    priors[param] = {}
    priors[param]['distribution'], priors[param]['hyperparameters'] = dist, hyperp

# Add second planet to the prior:
params = params + ['P_p2',   't0_p2',  'K_p2',    'ecc_p2','omega_p2']
dists = dists +   ['uniform','uniform','uniform', 'fixed', 'fixed']
hyperps = hyperps + [[1.,10.],[2458325.,2458330.],[0.,100.], 0., 90.]

# Repopulate priors dictionary:
priors = {}

for param, dist, hyperp in zip(params, dists, hyperps):
    priors[param] = {}
    priors[param]['distribution'], priors[param]['hyperparameters'] = dist, hyperp

dataset = juliet.load(priors = priors, rvfilename='rvs_toi141.dat', out_folder =
↪'toi141_rvs-global', \
                      GPrveparamfile='GP_regressors_rv.dat')

results = dataset.fit(n_live_points = 300)
```

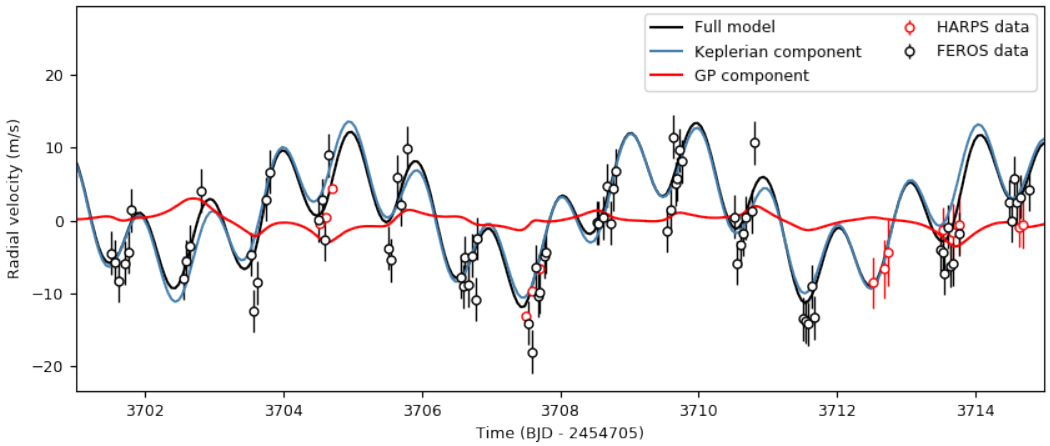
Once done, let's plot the results. We'll plot a portion of the time-series so we can check what the different components of the model are doing, and only plot the HARPS and FEROS data, which are the most constraining for our dataset:

```
# Define minimum and maximum times to evaluate the model on:
min_time, max_time = np.min(dataset.times_rv['FEROS'])-30,\
                 np.max(dataset.times_rv['FEROS'])+30

# Create model times on which we will evaluate the model:
model_times = np.linspace(min_time,max_time,5000)

# Extract full model and components of the RV model:
full_model, components = results.rv.evaluate('FEROS', t = model_times, GPregressors =␣
↪model_times, return_components = True)
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
instruments = ['HARPS','FEROS']
colors = ['red','black']

fig = plt.figure(figsize=(10,4))
for instrument,color in zip (instruments,colors):
    plt.errorbar(dataset.times_rv[instrument]-2454705,dataset.data_rv[instrument] -␣
↪components['mu'][instrument], \
                 yerr = dataset.errors_rv[instrument], fmt = 'o', label = instrument+
↪' data',mfc='white', mec = color, ecolor = color, \
                 elinewidth=1)

plt.plot(model_times-2454705,full_model - components['mu']['FEROS'],label='Full model
↪',color='black')
plt.plot(model_times-2454705,results.rv.model['deterministic'],label = 'Keplerian␣
↪component', color = 'steelblue')
plt.plot(model_times-2454705,results.rv.model['GP'], label = 'GP component',color='red
↪')
plt.xlim([3701,3715])
plt.ylabel('Radial velocity (m/s)')
plt.xlabel('Time (BJD - 2454705)')
plt.legend(ncol=2)
```



Nice! This plot is very similar to the one shown in Figure 8 of the TOI-141b paper in Espinoza et al. (2019) — only that in that paper, the authors used a different kernel. It is reassuring that this simple kernel gives very similar results! As can be seen, the key idea of a *global* model is evident from these results: it is a model that spans different instruments, modelling what could be an underlying physical process that impacts all of them simultaneously.

Now let us model the same data assuming an **instrument-by-instrument** model. For this, let's suppose the time-scale of the process is common to all the instruments, but that the amplitudes of the process are different for each of them. In order to tell to `juliet` that we want an instrument-by-instrument model, we have to first create a file with the GP regressors that identifies the regressors for each instrument — we have uploaded the one used in this example [here]. Then, we simply define the GP hyperparameters for each instrument — common parameters between instruments will have instruments separated by underscores after the GP hyperparameter name, like for `GP_rho` below:

```
priors = {}
```

(continues on next page)

(continued from previous page)

```
# Name of the parameters to be fit:
params = ['P_p1','t0_p1','mu_CORALIE14', \
          'mu_CORALIE07','mu_HARPS','mu_FEROS',\
          'K_p1', 'ecc_p1', 'omega_p1', 'sigma_w_CORALIE14','sigma_w_CORALIE07',\
          'sigma_w_HARPS','sigma_w_FEROS','GP_sigma_HARPS','GP_sigma_FEROS','GP_sigma_
↪CORALIE14', 'GP_sigma_CORALIE07',\
          'GP_rho_HARPS_FEROS_CORALIE14_CORALIE07']

# Distributions:
dists = ['normal','normal','uniform', \
         'uniform','uniform','uniform',\
         'uniform','fixed', 'fixed', 'loguniform', 'loguniform',\
         'loguniform', 'loguniform','loguniform','loguniform','loguniform','loguniform
↪',\
         'loguniform']

# Hyperparameters
hyperps = [[1.007917,0.000073], [2458325.5386,0.0011], [-100,100], \
           [-100,100], [-100,100], [-100,100], \
           [0.,100.], 0., 90., [1e-3, 100.], [1e-3, 100.], \
           [1e-3, 100.], [1e-3, 100.],[0.01,100.],[0.01,100.],[0.01,100.],[0.01,100.],
↪\
           [0.01,100.]]

# Populate the priors dictionary:
for param, dist, hyperp in zip(params, dists, hyperps):
    priors[param] = {}
    priors[param]['distribution'], priors[param]['hyperparameters'] = dist, hyperp

# Add second planet to the prior:
params = params + ['P_p2',   't0_p2',  'K_p2',    'ecc_p2','omega_p2']
dists = dists +   ['uniform','uniform','uniform', 'fixed', 'fixed']
hyperps = hyperps + [[1.,10.],[2458325.,2458330.],[0.,100.], 0., 90.]

# Repopulate priors dictionary:
priors = {}

for param, dist, hyperp in zip(params, dists, hyperps):
    priors[param] = {}
    priors[param]['distribution'], priors[param]['hyperparameters'] = dist, hyperp

dataset = juliet.load(priors = priors, rvfilename='rvs_toi141.dat', out_folder =
↪'toi141_rvs_i-i', \
                      GPrveparamfile='GP_regressors_rv_i-i.dat', verbose = True)



results = dataset.fit(n_live_points = 300)
```

Now let us plot the results of the fit. Because this is an instrument-by-instrument model, we have to plot the fits individually for each instruments. Let's plot the FEROS and HARPS data once again:

```
model_times = np.linspace(np.max(dataset.t_rv)-50,np.max(dataset.t_rv),1000)

import matplotlib.pyplot as plt
instruments = ['HARPS','FEROS']
```

(continues on next page)

(continued from previous page)

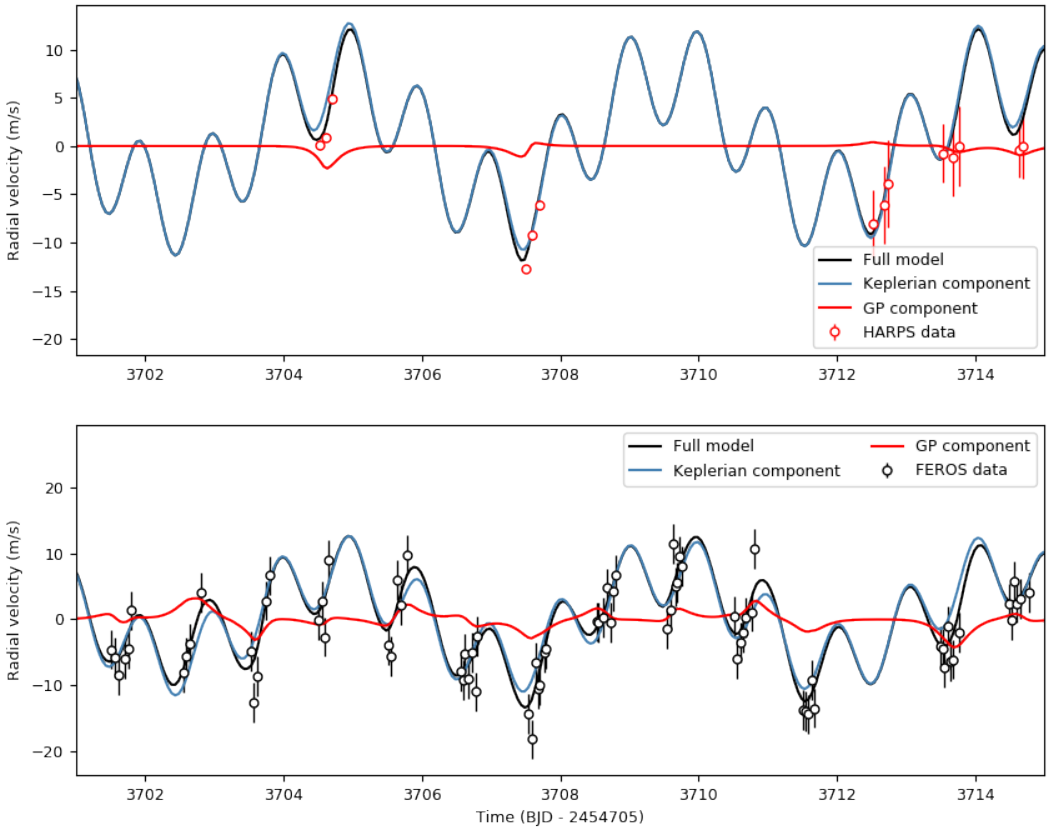
```
colors = ['red','black']

fig = plt.figure(figsize=(10,8))
counter = 0
for instrument,color in zip (instruments,colors):
    plt.subplot('21'+str(counter+1))
    keplerian, components = results.rv.evaluate(instrument,t = model_times,␣
↪GPregressors = model_times, return_components = True)
    plt.errorbar(dataset.times_rv[instrument]-2454705,dataset.data_rv[instrument] -␣
↪components['mu'], \
                 yerr = dataset.errors_rv[instrument], fmt = 'o', label = instrument+
↪' data',mfc='white', mec = color, ecolor = color, \
                 elinewidth=1)
    plt.plot(model_times-2454705,keplerian,label='Full model',color='black')
    plt.plot(model_times-2454705,results.rv.model[instrument]['deterministic'],label␣
↪= 'Keplerian component', color = 'steelblue')
    plt.plot(model_times-2454705,results.rv.model[instrument]['GP'], label = 'GP␣
↪component',color='red')
    counter += 1
    plt.legend()
    plt.xlim([3701,3715])
    plt.ylabel('Radial velocity (m/s)')
plt.xlabel('Time (BJD - 2454705)')
```

Notice how in this instrument-by-instrument GP fit, not only the amplitude but the overall shape of the GP component is different between instruments. This is exactly what we are modelling with an instrument-by-instrument GP fit: a process that might share some hyperparameters, but that has different realizations on each instrument.

So, is the instrument-by-instrument model or the global GP fit the best for the TOI-141 dataset? We can use the log-evidences to find this out! For the global model, we obtain a log-evidence of $\ln Z = -678.76 \pm 0.03$, whereas for the instrument-by-instrument model we obtain a log-evidence of $\ln Z = -679.4 \pm 0.1$. From this, we see that although they are statistically indistinguishable ($\Delta \ln Z < 2$), we will most likely want to favor the global model as it has fewer parameters. One interesting point the reader might make is that, from the plots above, it might *seem* FEROS is dominating the GP component — so it might be that the GP signal is actually arising from the FEROS data, and not from all the other instruments. One way to check if this is the case is to run an instrument-by-instrument GP model where a GP is applied only to the FEROS data; physically, this would be modelling a signal that is only arising in this instrument due to, e.g., unknown instrumental systematics. It is easy to test this out with `juliet`; we just repeat the instrument-by-instrument model above but adding a GP only to the FEROS data:

```
priors = {}

# Name of the parameters to be fit:
params = ['P_p1','t0_p1','mu_CORALIE14', \
          'mu_CORALIE07','mu_HARPS','mu_FEROS',\
          'K_p1', 'ecc_p1', 'omega_p1', 'sigma_w_CORALIE14','sigma_w_CORALIE07',\
          'sigma_w_HARPS','sigma_w_FEROS','GP_sigma_FEROS', 'GP_rho_FEROS']

# Distributions:
dists = ['normal','normal','uniform', \
         'uniform','uniform','uniform',\
         'uniform','fixed', 'fixed', 'loguniform', 'loguniform',\
         'loguniform', 'loguniform','loguniform','loguniform']

# Hyperparameters
hyperps = [[1.007917,0.000073], [2458325.5386,0.0011], [-100,100], \
           [-100,100], [-100,100], [-100,100], \
```

(continues on next page)


Radial velocity (m/s)
Full model
Keplerian component
GP component
HARPS data
Full model
Keplerian component
GP component
FEROS data
Radial velocity (m/s)
Time (BJD - 2454705)

(continued from previous page)

```
            [0.,100.], 0., 90., [1e-3, 100.], [1e-3, 100.], \
            [1e-3, 100.], [1e-3, 100.],[0.01,100.],[0.01,100.]]

# Populate the priors dictionary:
for param, dist, hyperp in zip(params, dists, hyperps):
    priors[param] = {}
    priors[param]['distribution'], priors[param]['hyperparameters'] = dist, hyperp

# Add second planet to the prior:
params = params + ['P_p2',   't0_p2',  'K_p2',    'ecc_p2','omega_p2']
dists = dists +   ['uniform','uniform','uniform', 'fixed', 'fixed']
hyperps = hyperps + [[1.,10.],[2458325.,2458330.],[0.,100.], 0., 90.]

# Repopulate priors dictionary:
priors = {}

for param, dist, hyperp in zip(params, dists, hyperps):
    priors[param] = {}
    priors[param]['distribution'], priors[param]['hyperparameters'] = dist, hyperp

dataset = juliet.load(priors = priors, rvfilename='rvs_toi141.dat', out_folder =
↪'toi141_rvs_i-i-FEROS', \
                      GPrveparamfile='GP_regressors_rv_i-i-FEROS.dat', verbose = True)



results = dataset.fit(n_live_points = 300)
```

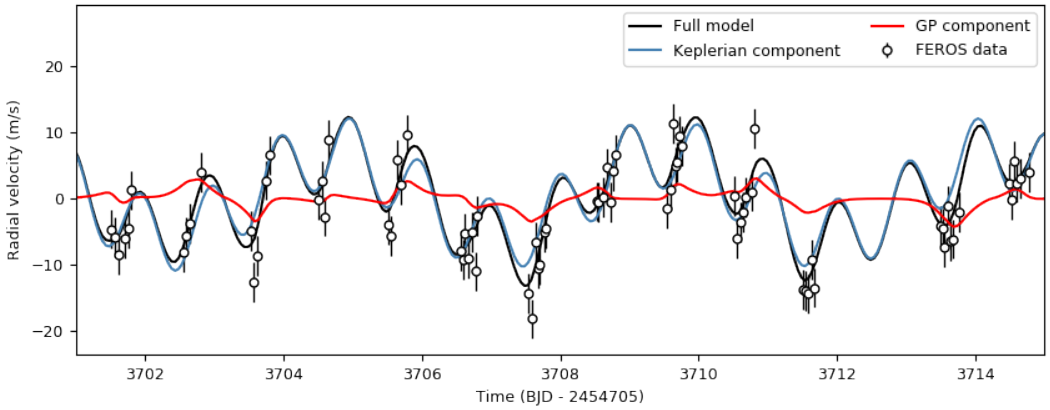
Let us plot the result to see how this looks like:

```
model_times = np.linspace(np.max(dataset.t_rv)-50,np.max(dataset.t_rv),1000)

import matplotlib.pyplot as plt
instruments = ['FEROS']
colors = ['black']

fig = plt.figure(figsize=(10,8))
counter = 0
for instrument,color in zip (instruments,colors):
    plt.subplot('21'+str(counter+1))
    keplerian, components = results.rv.evaluate(instrument,t = model_times,␣
↪GPregressors = model_times, return_components = True)
    plt.errorbar(dataset.times_rv[instrument]-2454705,dataset.data_rv[instrument] -␣
↪components['mu'], \
                 yerr = dataset.errors_rv[instrument], fmt = 'o', label = instrument+
↪' data',mfc='white', mec = color, ecolor = color, \
                 elinewidth=1)
    plt.plot(model_times-2454705,keplerian,label='Full model',color='black')
    plt.plot(model_times-2454705,results.rv.model[instrument]['deterministic'],label␣
↪= 'Keplerian component', color = 'steelblue')
    plt.plot(model_times-2454705,results.rv.model[instrument]['GP'], label = 'GP␣
↪component',color='red')
    counter += 1
    plt.legend()
    plt.xlim([3701,3715])
    plt.ylabel('Radial velocity (m/s)')
plt.xlabel('Time (BJD - 2454705)')
```

(continues on next page)

(continued from previous page)

```
plt.legend(ncol=2)
```



It seems the signal is fairly similar in this narrow time-range to the one we obtained in the global model and the instrument-by-instrument models above! However, `juliet` has one more piece of data that can allow us to discriminate the "best" model: the log-evidence. This model has a log-evidence of $\ln Z = -681.65 \pm 0.07$ — the global model has a log-evidence which is $\Delta \ln Z = 2.9$ *higher* than this model and thus is about 18 times more likely than this FEROS-only instrument-by-instrument model. Given our data, then, it seems the *global* model is the best model at hand, at least compared against the instrument-by-instrument models defined above.

CHAPTER 10

Incorporating transit-timing variations

The transit fits that have been presented so far in the tutorials assume that the transit times, T are exactly periodic, i.e., they can be predicted by the simple relationship

$T(n) = t_0 + nP$,

where t_0 is the time-of-transit center at epoch zero ($n = 0$), P is the period of the orbit and n is the transit epoch. In some particular cases, however, this simple relationship might not be satisfied. Because of gravitational/dynamical interactions with additional bodies in the system, the exoplanet under study might undergo what we usually refer to as *transit timing variations* (TTVs), where the transit times are not exactly periodic and vary due to these (in principle unknown) interactions. If we define those variations as extra perturbations δt_n to the above defined timing equation, we can write the time-of-transit centers as:

$T(n) = t_0 + nP + \delta t_n$.

Within `juliet`, there are two ways to fit for these perturbations. One way is to fit for each of the $T(n)$ directly, while there is also an option to fit for *some* perturbations δt_n. In this tutorial, we explore why those two possible parametrizations are allowed, and what they imply for the fits we perform. We will use the HATS-46 b TESS dataset, which we already analyzed in the *Lightcurve fitting with juliet* section, as a case-study in this tutorial.

10.1 Fitting for the transit times directly

If we choose to fit for the transit times $T(n)$ directly, `juliet` will expect priors for these but it is expected that you will *not* supply priors for t_0 and P (e.g., `t0_p1` and `P_p1`). The reason for this is that these latter parameters will be computed directly from each sample of the $T(n)$ as the intercept (t_0) and slope (P) that best-fits (in a least-squares sense) the sampled $T(n)$. This is, of course, a matter of definition — we are assuming that what we refer to when we speak of P and t_0 in a TTV fit are the slope and intercept, respectively, of a last-squares fit to the transit times.

Within *juliet*, the transit times are defined through the parameter `T_p1_instrument_n` — here, `instrument` defines the instrument where that transit occurs (e.g., `TESS`), `n` the transit epoch and, in this case, we are fitting the transit-times to planet `p1`; `juliet` is able to handle different perturbations for different planets in the system.

Let's try finding how big the perturbations are on the HATS-46 b TESS dataset. For this, we use the same priors used in section *Lightcurve fitting with juliet*, but we remove the priors on t_0 and P (i.e., `t0_p1` and `P_p1`), and add the

priors for each time of transit. We will assume normal, zero-mean gaussian priors with a standard deviation of 0.1 days (i.e., about 2.4 hours) for the planet. We define these along the other priors previously defined for HATS-46 b as follows:

```
import juliet

# First, load original dataset we used in the previous tutorial:
t, f, ferr  = juliet.get_TESS_data('https://archive.stsci.edu/hlsps/'+\
                                   'tess-data-alerts/hlsp_tess-data-'+\
                                   'alerts_tess_phot_00281541555-s02_'+\
                                   'tess_v1_lc.fits')

times, fluxes, fluxes_error = {},{},{}
times['TESS'], fluxes['TESS'], fluxes_error['TESS'] = t,f,ferr

# Define same parameters, distributions and hyperparameters defined in
# that same tutorial:
params = ['r1_p1','r2_p1','q1_TESS','q2_TESS','ecc_p1','omega_p1',\
          'rho', 'mdilution_TESS', 'mflux_TESS', 'sigma_w_TESS']

dists = ['uniform','uniform','uniform','uniform','fixed','fixed',\
                 'loguniform', 'fixed', 'normal', 'loguniform']

hyperps = [[0.,1], [0.,1.], [0., 1.], [0., 1.], 0.0, 90.,\
                   [100., 10000.], 1.0, [0.,0.1], [0.1, 1000.]]

# Add to these the transit times:
params = params + ['T_p1_TESS_0','T_p1_TESS_1', 'T_p1_TESS_3', 'T_p1_TESS_4']
dists = dists + ['normal','normal','normal','normal']
hyperps = hyperps + [[1358.4,0.1],[1363.1,0.1], [1372.5,0.1], [1377.2,0.1]]
```

Note how we have defined transit-times only for $n = 0, 1, 3, 4$. We skipped the transit with $n = 2$ as this one falls just where there is a gap in the data (which happens on every TESS sector to download the data back at Earth). We now put everything together into the `priors` dictionary, and re-fit the data:

```
# Build the prior dictionary with the above information:
priors = juliet.utils.generate_priors(params,dists,hyperps)

# Load and fit dataset with juliet:
dataset = juliet.load(priors=priors, t_lc = times, y_lc = fluxes, \
                      yerr_lc = fluxes_error, out_folder = 'hats46-ttvs')

results = dataset.fit()
```

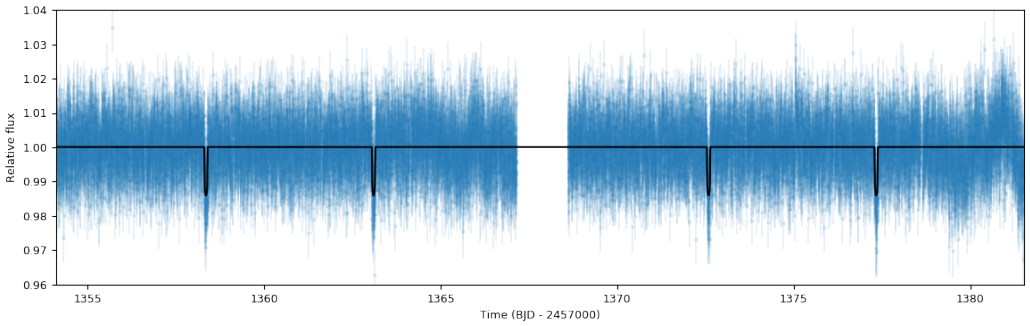
The resulting fit looks as good as the original one shown in the *Lightcurve fitting with juliet* section:

```
import matplotlib.pyplot as plt

# Extract median model and the ones that cover the 68% credibility band around it:
transit_model = results.lc.evaluate('TESS')

# Plot data and best-fit model:
fig = plt.figure(figsize=(12,4))
plt.errorbar(dataset.times_lc['TESS'], dataset.data_lc['TESS'], \
             yerr = dataset.errors_lc['TESS'], fmt = '.' , alpha = 0.1)
plt.plot(dataset.times_lc['TESS'], transit_model, color='black',zorder=10)

# Define labels, limits, etc. of the plot:
```

(continues on next page)

(continued from previous page)

```
plt.xlim([np.min(dataset.times_lc['TESS']),np.max(dataset.times_lc['TESS'])])
plt.ylim([0.96,1.04])
plt.xlabel('Time (BJD - 2457000)')
plt.ylabel('Relative flux')
```



Let us, however, explore the posterior distribution of the parameters, which will enlighten us in understanding the constraints this puts on the HATS-46 b system. First of all, the `posteriors.dat` file for this fit shows the following summary statistics of the posterior distributions of the parameters:

```
# Parameter Name                Median                  Upper 68 CI                 L
↪Lower 68 CI
r1_p1                           0.5416863162            0.1568514219                0.
↪1434447471
r2_p1                           0.1111807484            0.0034296154                0.
↪0035118401
p_p1                            0.1111807484            0.0034296154                0.
↪0035118401
b_p1                            0.3125294743            0.2352771328                0.
↪2151671206
inc_p1                          88.9071308890           0.7710955693                1.
↪0698162411
q1_TESS                         0.2692194780            0.3474123320                0.
↪1815095451
q2_TESS                         0.3763637953            0.3601869056                0.
↪2406970909
rho                             3681.1771806645         728.0596617015              1160.
↪9706095575
mflux_TESS                      -0.0000894483           0.0000568777                0.
↪0000560349
sigma_w_TESS                    4.4343278327            57.2232056206               4.
↪1133207064
T_p1_TESS_0                     1358.3561072664         0.0018110928                0.
↪0021025622
T_p1_TESS_1                     1363.1001349693         0.0020743972                0.
↪0019741023
T_p1_TESS_3                     1372.5833491831         0.0017507552                0.
↪0019396261
T_p1_TESS_4                     1377.3292128814         0.0016890000                0.
↪0014434932
P_p1                            4.7429737505            0.0005494323                0.
↪0005702781
a_p1                            16.3556306970           1.0182669217                1.
↪9356637282
```

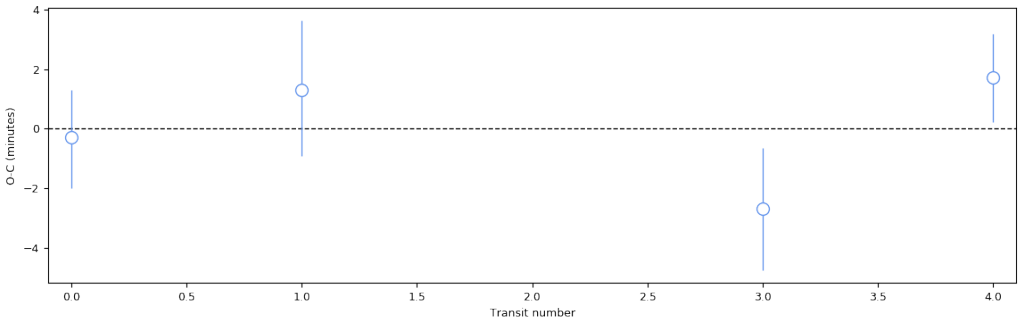
(continues on next page)

(continued from previous page)

```
t0_p1                          1358.3562648736          0.0016147678              0.
↪0016588470
```

First of all, note how `juliet` spits out not only the posterior distributions for the `T` parameters (i.e., the $T(n)$ in our notation above), but also for the corresponding slope (`P_p1`) and intercept (`t0_p1`) that best fits the transit times. These are actually pretty useful to plot the observed (i.e., the $T(n)$) minus the predicted (assuming the transits were exactly periodic, i.e., $t0+nP$) variations from our data, which is actually what allows us to see what level (amplitude) of TTVs our data constrain. We can plot this so-called "O-C" plot as follows:

```
# To extract O-C data from the posterior distributions, first define some variables:
transit_numbers = np.array([0,1,3,4])
OC = np.zeros(len(transit_numbers))
OC_up_err = np.zeros(len(transit_numbers))
OC_down_err = np.zeros(len(transit_numbers))
instrument = 'TESS'
# Now go through known transit-numberings, and generate the O-C distributions. From␣
↪there,
# compute the medians and 68% credibility bands:
for i in range(len(transit_numbers)):
    transit_number = transit_numbers[i]
    # Compute distribution of predicted times:
    computed_time = results.posteriors['posterior_samples']['t0_p1'] + transit_
↪number*results.posteriors['posterior_samples']['P_p1']
    # Extract observed times:
    observed_time = results.posteriors['posterior_samples']['T_p1_'+instrument+'_
↪'+str(transit_number)]
    # Generate O-C (multiply by 24*60 to get it in minutes) posterior distribution,
    # and get quantiles from it:
    val,vup,vdown = juliet.utils.get_quantiles((observed_time - computed_time)*24*60.)
    # Save value and "1-sigma" errors:
    OC[i], OC_up_err[i], OC_down_err[i] = val, vup-val,val-vdown

# Finally, generate plot with the O-C:
fig = plt.figure(figsize=(14,4))
plt.errorbar(transit_numbers,OC,yerr=[OC_down_err,OC_up_err],fmt='o',mfc='white',mec=
↪'cornflowerblue',ecolor='cornflowerblue',ms=10,elinewidth=1,zorder=3)
plt.plot([-0.1,4.1],[0.,0],'--',linewidth=1,color='black',zorder=2)
plt.xlim([-0.1,4.1])
plt.xlabel('Transit number')
plt.ylabel('O-C (minutes)')
plt.savefig('oc.png',transparent=True)
```



Beautiful! From this plot we can see that any possible TTV amplitudes are constrained to be below ~a couple of

minutes if they exist within the observed time-frame of the HATS-46 b observations in this sector.

10.2 Fitting for transit timing perturbations

Suppose a colleague of yours (or a referee) finds that transit number 3 above is "interesting", as it is more than one sigma away from the dashed line (i.e., 1-sigma away from showing "no deviation from a perfectly periodic transit"). You answer back that, assuming the errors are more or less gaussian, having 1 out of 4 datapoints not matching at 1-sigma is expected. However, they are still intrigued: is there evidence in the data for that transit being special in terms of its transit timing? Could it be that a hint from TTVs showed up on that particular transit? Answering questions like this one is when fitting for the TTV perturbations defined above, the δt_n, becomes handy.

Let's assume that all the other transits are periodic except for transit number 3. To fit for an extra perturbation in that transit, within `juliet` we use the `dt_p1_instrument_n` parameters — here, `instrument` defines the instrument where that transit occurs (e.g., `TESS`), `n` the transit epoch and, in this case, we are fitting the transit-time perturbation to planet `p1`. Again, `juliet` is able to handle different perturbations for different planets. In our case, then, we will be adding a parameter `dt_p1_TESS_3`, and will in addition be providing priors for the time-of-transit center (`t0_p1`) and period (`P_p1`) in the system, which will be in turn constrained by the other transits. To do this with `juliet` we would do the following. First, we set the usual priors (the same as the original fit done in the *Lightcurve fitting with juliet* section):

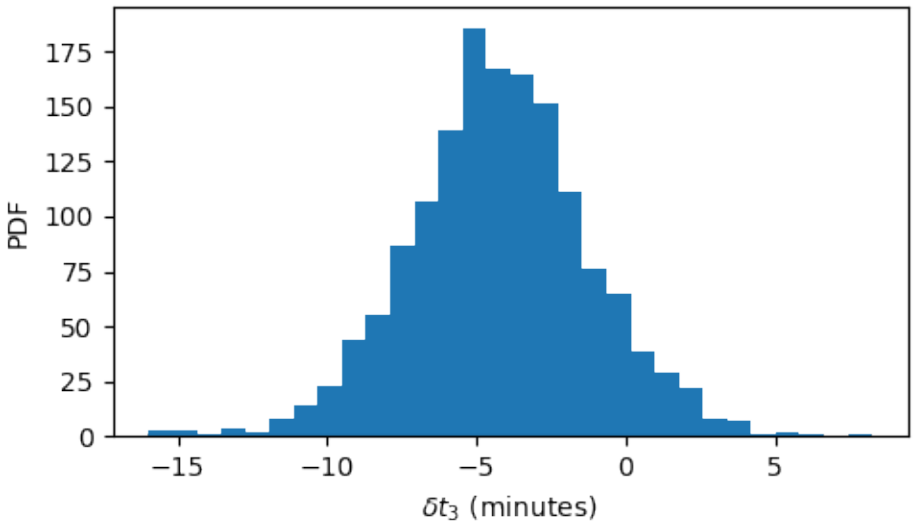
```
# Name of the parameters to be fit:
params = ['P_p1','t0_p1','r1_p1','r2_p1','q1_TESS','q2_TESS','ecc_p1','omega_p1',\
             'rho', 'mdilution_TESS', 'mflux_TESS', 'sigma_w_TESS']

# Distributions:
dists = ['normal','normal','uniform','uniform','uniform','uniform','fixed','fixed',\
                'loguniform', 'fixed', 'normal', 'loguniform']

# Hyperparameters
hyperps = [[4.7,0.1], [1358.4,0.1], [0.,1], [0.,1.], [0., 1.], [0., 1.], 0.0, 90.,\
                  [100., 10000.], 1.0, [0.,0.1], [0.1, 1000.]]

# Populate the priors dictionary:
for param, dist, hyperp in zip(params, dists, hyperps):
    priors[param] = {}
    priors[param]['distribution'], priors[param]['hyperparameters'] = dist, hyperp
```

However, we now add the perturbation to the third transit. We wrap up the `priors` dictionary and perform the fit:

```
params = params + ['dt_p1_TESS_3']
dists = dists + ['normal']
hyperps = hyperps + [[0.0,0.1]]

# Populate the priors dictionary:
priors = juliet.utils.generate_priors(params,dists,hyperps)

# Load and fit dataset with juliet:
dataset = juliet.load(priors=priors, t_lc = times, y_lc = fluxes, \
                  yerr_lc = fluxes_error, out_folder = 'hats46-ttvs-perturbations',␣
↪verbose = True)

results = dataset.fit(n_live_points)
```

The resulting posterior on the timing perturbation looks as follows:

Is this convincing evidence for something special happening in transit 3? Luckily, `juliet` reports the bayesian



evidence of this fit, which is $\ln Z_{per} = 64199$. The corresponding evidence for the fit done in the *Lightcurve fitting with juliet* section (with no perturbation) is $\ln Z_{no-per} = 64202.1$ — so a $\Delta \ln Z = 3$ in favour of **no** perturbation. The model without this timing perturbation is *about 20 times more likely given the data at hand* than the one with the perturbation. A pretty good bet against something special happening on transit number 3 for me (and probably you, your colleague and the referee!).

Note: The implementation discussed here was enormously beneffited by the discussions presented in the literature, both on the EXOFASTv2 paper (Section 18) and the discussion on the `exoplanet` package about their TTV implementation. We refer the users to these sources to learn more about this particular implementation of TTVs, and note that this is an approximation to the real dynamical problem that TTVs impose on transiting exoplanetary systems, as we are not considering changes to the other transit parameters. Photodynamical models are not yet supported within `juliet`.

CHAPTER 11

Multithreading

`juliet` can be used in multiple cores in order to speed up the data fitting processes. If using `MultiNest` this is done via OpenMPI, whereas via `dynesty` this is done using internal `python` multi-threading capabilities. In what follows, we explain how to perform multiple core runs with `juliet`.

11.1 Multithreading with MultiNest

In order to use the multi-threading capabilities with `juliet`, you have to have OpenMPI in your computer. You can check if this is available in your system by opening a terminal and writing `mpirun`. If this command prompts you to something similar to:

```
--------------------------------------------------------------------------
mpirun could not find anything to do.

It is possible that you forgot to specify how many processes to run
via the "-np" argument.
--------------------------------------------------------------------------
```

Then that's it, you have OpenMPI. If not, installing it is simple. You just have to follow the instructions to compile OpenMPI [here]. Once this is done, you have to install `mpi4py`, which is easily done via `pip`:

```
pip install mpi4py
```

Once all this is done you are good to go! To run a juliet run on `X` number of cores, simply do:

```
mpirun -np X python yourscript.py
```

11.2 Multithreading with dynesty

Applying multi-threading capabilities for `dynesty` is much simpler than for MultiNest. This can be automatically activated once a `juliet.load` object is made to fit the data — simply define the number of threads you want to use

and `juliet` will assume you need multi-threading capabilities. So, for example, to use `juliet` with 6 number of cores, in a session you would do:

```
# Load and fit dataset with juliet:
dataset = juliet.load(priors=priors, t_lc = times, y_lc = fluxes, \
                    yerr_lc = fluxes_error, out_folder = 'hats46')

results = dataset.fit(use_dynesty=True, dynesty_nthreads = 6)
```

CHAPTER 12

Contributors

juliet is being developed by Nestor Espinoza (@nespinoza) and Diana Kossakowski (@dianadianadiana).

Contributions have been made by several authors, including Johannes Buchner (@JohannesBuchner), Jonas Kemmer (@JonasKemmer), Martin Schlecker (@matiscke), Jose Vines (@jvines) and Ian Weaver (@icweaver).

Want to contribute? Grab a project, create your own and open a pull request!

CHAPTER 13

License & Attribution

juliet is being developed by Nestor Espinoza and Diana Kossakowski in a public GitHub repository. The source code is made available under the terms of the MIT license.

If you make use of this code, please cite the paper:

```
@ARTICLE{2019MNRAS.490.2262E,
       author = {{Espinoza}, N{\'e}stor and {Kossakowski}, Diana and {Brahm}, Rafael},
        title = "{juliet: a versatile modelling tool for transiting and non-
↪transiting exoplanetary systems}",
      journal = {\mnras},
     keywords = {methods: data analysis, methods: statistical, techniques:␣
↪photometric, techniques: radial velocities, planets and satellites: fundamental␣
↪parameters, planets and satellites: individual: K2-140b, K2-32b, c, d, Astrophysics␣
↪- Earth and Planetary Astrophysics},
         year = "2019",
        month = "Dec",
       volume = {490},
       number = {2},
        pages = {2262-2283},
          doi = {10.1093/mnras/stz2688},
archivePrefix = {arXiv},
       eprint = {1812.08549},
 primaryClass = {astro-ph.EP},
       adsurl = {https://ui.adsabs.harvard.edu/abs/2019MNRAS.490.2262E},
      adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

CHAPTER 14

Additional citations

In addition to the citation above, and depending on the methods and samplers used in your research, please make sure to cite the appropiate sources:

- **If transit fits were performed**, cite `batman`:

```
@ARTICLE{batman,
       author = {{Kreidberg}, Laura},
        title = "{batman: BAsic Transit Model cAlculatioN in Python}",
      journal = {Publications of the Astronomical Society of the Pacific},
     keywords = {Astrophysics - Earth and Planetary Astrophysics},
         year = 2015,
        month = Nov,
       volume = {127},
        pages = {1161},
          doi = {10.1086/683602},
archivePrefix = {arXiv},
       eprint = {1507.08285},
 primaryClass = {astro-ph.EP},
       adsurl = {https://ui.adsabs.harvard.edu/\#abs/2015PASP..127.1161K},
      adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

In addition, `juliet` allows to sample limb-darkening coefficients using the method outlined in Kipping (2013). If using it, please cite:

```
@ARTICLE{2013MNRAS.435.2152K,
       author = {{Kipping}, David M.},
        title = "{Efficient, uninformative sampling of limb darkening coefficients␣
↪for two-parameter laws}",
      journal = {\mnras},
     keywords = {methods: analytical, stars: atmospheres, Astrophysics - Solar and␣
↪Stellar Astrophysics, Astrophysics - Earth and Planetary Astrophysics},
         year = 2013,
        month = nov,
```

(continues on next page)

(continued from previous page)

```
       volume = {435},
       number = {3},
        pages = {2152-2160},
          doi = {10.1093/mnras/stt1435},
archivePrefix = {arXiv},
       eprint = {1308.0009},
 primaryClass = {astro-ph.SR},
       adsurl = {https://ui.adsabs.harvard.edu/abs/2013MNRAS.435.2152K},
      adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

If using the uninformative sample for radius and impact parameters outlined in Espinoza (2018), cite:

```
@ARTICLE{2018RNAAS...2..209E,
       author = {{Espinoza}, N{\'e}stor},
        title = "{Efficient Joint Sampling of Impact Parameters and Transit Depths in␣
↪Transiting Exoplanet Light Curves}",
      journal = {Research Notes of the American Astronomical Society},
     keywords = {Astrophysics - Earth and Planetary Astrophysics},
         year = 2018,
        month = nov,
       volume = {2},
       number = {4},
          eid = {209},
        pages = {209},
          doi = {10.3847/2515-5172/aaef38},
archivePrefix = {arXiv},
       eprint = {1811.04859},
 primaryClass = {astro-ph.EP},
       adsurl = {https://ui.adsabs.harvard.edu/abs/2018RNAAS...2..209E},
      adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

- **If radial-velocity fits were performed**, cite `radvel`:

```
@ARTICLE{radvel,
   author = {{Fulton}, B.~J. and {Petigura}, E.~A. and {Blunt}, S. and {Sinukoff}, E.
       },
    title = "{RadVel: The Radial Velocity Modeling Toolkit}",
  journal = {\pasp},
archivePrefix = "arXiv",
   eprint = {1801.01947},
 primaryClass = "astro-ph.IM",
     year = 2018,
    month = apr,
   volume = 130,
   number = 4,
    pages = {044504},
      doi = {10.1088/1538-3873/aaaaa8},
   adsurl = {http://adsabs.harvard.edu/abs/2018PASP..130d4504F},
  adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

- **If Gaussian Processes were used**, cite either `george` and/or `celerite` depending on the used kernel(s):

```
 @article{george,
    author = {{Ambikasaran}, S. and {Foreman-Mackey}, D. and
```

(continues on next page)

(continued from previous page)

```
                {Greengard}, L. and {Hogg}, D.~W. and {O'Neil}, M.},
     title = "{Fast Direct Methods for Gaussian Processes}",
      year = 2014,
     month = mar,
       url = http://arxiv.org/abs/1403.6015
}
```

```
@article{celerite,
    author = {{Foreman-Mackey}, D. and {Agol}, E. and {Angus}, R. and
              {Ambikasaran}, S.},
     title = {Fast and scalable Gaussian process modeling
              with applications to astronomical time series},
      year = {2017},
   journal = {AJ},
    volume = {154},
     pages = {220},
       doi = {10.3847/1538-3881/aa9332},
       url = {https://arxiv.org/abs/1703.09710}
}
```

- **If MultiNest was used to perform the sampling**, cite `MultiNest` and `PyMultiNest`:

```
@ARTICLE{MultiNest,
   author = {{Feroz}, F. and {Hobson}, M.~P. and {Bridges}, M.},
    title = "{MULTINEST: an efficient and robust Bayesian inference tool for␣
↪cosmology and particle physics}",
  journal = {\mnras},
archivePrefix = "arXiv",
   eprint = {0809.3437},
 keywords = {methods: data analysis , methods: statistical},
     year = 2009,
    month = oct,
   volume = 398,
    pages = {1601-1614},
      doi = {10.1111/j.1365-2966.2009.14548.x},
   adsurl = {http://adsabs.harvard.edu/abs/2009MNRAS.398.1601F},
  adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}

@ARTICLE{PyMultiNest,
   author = {{Buchner}, J. and {Georgakakis}, A. and {Nandra}, K. and {Hsu}, L. and
         {Rangel}, C. and {Brightman}, M. and {Merloni}, A. and {Salvato}, M. and
         {Donley}, J. and {Kocevski}, D.},
    title = "{X-ray spectral modelling of the AGN obscuring region in the CDFS:␣
↪Bayesian model selection and catalogue}",
  journal = {\aap},
archivePrefix = "arXiv",
   eprint = {1402.0004},
 primaryClass = "astro-ph.HE",
 keywords = {accretion, accretion disks, methods: data analysis, methods: statistical,
↪ galaxies: nuclei, X-rays: galaxies, galaxies: high-redshift},
     year = 2014,
    month = apr,
   volume = 564,
      eid = {A125},
    pages = {A125},
```

(continues on next page)

(continued from previous page)

```
      doi = {10.1051/0004-6361/201322971},
   adsurl = {http://adsabs.harvard.edu/abs/2014A%26A...564A.125B},
  adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

- **If dynesty was used to perform the sampling**, cite `dynesty`:

```
@ARTICLE{2020MNRAS.493.3132S,
       author = {{Speagle}, Joshua S.},
        title = "{DYNESTY: a dynamic nested sampling package for estimating Bayesian␣
↪posteriors and evidences}",
      journal = {\mnras},
     keywords = {methods: data analysis, methods: statistical, Astrophysics -␣
↪Instrumentation and Methods for Astrophysics, Statistics - Computation},
         year = 2020,
        month = apr,
       volume = {493},
       number = {3},
        pages = {3132-3158},
          doi = {10.1093/mnras/staa278},
archivePrefix = {arXiv},
       eprint = {1904.02180},
 primaryClass = {astro-ph.IM},
       adsurl = {https://ui.adsabs.harvard.edu/abs/2020MNRAS.493.3132S},
      adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

- **If UltraNest was used to perform the sampling**, follow the instructions in the UltraNest read-the-docs.

Index