exoplanet

Release 0.1.5

Mar 08, 2019

Contents

Scalable Gaussian Processes using celerite.

Fast and accurate limb darkened light curves using starry.

Common reparameterizations for limb darkening parameters, and planet radius and impact parameter.

And many others!

All of these functions and distributions include methods for efficiently calculating their *gradients* so that they can be used with gradient-based inference methods like Hamiltonian Monte Carlo, No U-Turns Sampling, and variational inference. These methods tend to be more robust than the methods more commonly used in astronomy (like ensemble samplers and nested sampling) especially when the model has more than a few parameters. For many exoplanet applications, *exoplanet* (the code) can improve the typical performance by orders of magnitude.

exoplanet is being actively developed in a public repository on GitHub so if you have any trouble, open an issue there.

CHAPTER 1

User guide

1.1 Installation

exoplanet doesn't have a compiled components so it can be easily installed from source or by using pip.

1.1.1 Dependencies

The only required dependencies for *exoplanet* are NumPy, PyMC3 and AstroPy. These can be installed using conda or pip:

```
conda install numpy pymc3 astropy
```

or

```
pip install numpy pymc3 astropy
```

1.1.2 Using pip

exoplanet can also be installed using pip:

```
pip install exoplanet
```

1.1.3 From Source

The source code for *exoplanet* can be downloaded and installed from GitHub by running

```
git clone --recursive https://github.com/dfm/exoplanet.git
cd exoplanet
python setup.py install
```

1.1.4 Testing

To run the unit tests, install the following dependencies using pip or conda (you'll need to use the `conda-forge` channel to get starry):

```
conda install -c conda-forge numpy scipy astropy pymc3 pytest starry pip
pip install batman-package parameterized nose
```

and then execute:

```
py.test -v
```

All of the tests should (of course) pass. If any of the tests don't pass and if you can't sort out why, open an issue on GitHub.

Note: This tutorial was generated from an IPython notebook that can be downloaded here.

```
theano version: 1.0.4
pymc3 version: 3.6
exoplanet version: 0.1.5
```

1.2 Citing exoplanet & its dependencies

The *exoplanet* package is mostly just glue that connects many other ideas and software. In a situation like this, it can be easy to forget about the important infrastructure upon which our science is built. In order to make sure that you can easily give credit where credit is due, we have tried to make it as painless as possible to work out which citations are expected for a model fit using *exoplanet* by including a *`exoplanet.citations.get_citations_for_model()`* function that introspects the current PyMC3 model and constructs a list of citations for the functions used in that model.

For example, you might compute a quadratically limb darkened light curve using `starry` (via the `exoplanet.light_curve.StarryLightCurve` class):

```
import pymc3 as pm
import exoplanet as xo

with pm.Model() as model:
    u = xo.distributions.QuadLimbDark("u")
    orbit = xo.orbits.KeplerianOrbit(period=10.0)
    light_curve = xo.StarryLightCurve(u)
    transit = light_curve.get_light_curve(r=0.1, orbit=orbit, t=[0.0, 0.1])

    txt, bib = xo.citations.get_citations_for_model()
```

The *`exoplanet.citations.get_citations_for_model()`* function would generate an acknowledgement that cites:

- PyMC3: for the inference engine and modeling framework,
- Theano: for the numerical infrastructure,
- AstroPy: for units and constants,
- Kipping (2013): for the reparameterization of the limb darkening parameters for a quadratic law, and

- Luger, et al. (2018): for the light curve calculation.

The first output from *exoplanet.citations.get_citations_for_model()* gives the acknowledgement text:

```
print(txt)
```

```
This research made use of textsf{exoplanet} citep{exoplanet} and its
dependencies citep{exoplanet:astropy13, exoplanet:astropy18,
exoplanet:exoplanet, exoplanet:kipping13, exoplanet:luger18, exoplanet:pymc3,
exoplanet:theano}.
```

And the second output is a string with BibTeX entries for each of the citations in the acknowledgement text:

```
print(bib)
```

```
@misc{exoplanet:exoplanet,
  author = {Dan Foreman-Mackey and
            Geert Barentsen and
            Tom Barclay},
   title = {dfm/exoplanet: exoplanet v0.1.4},
   month = feb,
    year = 2019,
     doi = {10.5281/zenodo.2561395},
     url = {https://doi.org/10.5281/zenodo.2561395}
}


@article{exoplanet:pymc3,
    title={Probabilistic programming in Python using PyMC3},
   author={Salvatier, John and Wiecki, Thomas V and Fonnesbeck, Christopher},
  journal={PeerJ Computer Science},
   volume={2},
    pages={e55},
     year={2016},
publisher={PeerJ Inc.}
}


@article{exoplanet:theano,
    title="{Theano: A {Python} framework for fast computation of mathematical
            expressions}",
   author={{Theano Development Team}},
  journal={arXiv e-prints},
   volume={abs/1605.02688},
     year=2016,
    month=may,
      url={http://arxiv.org/abs/1605.02688}
}


@ARTICLE{exoplanet:kipping13,
   author = {{Kipping}, D.~M.},
    title = "{Efficient, uninformative sampling of limb darkening coefficients
             for two-parameter laws}",
  journal = {mnras},
```

```
     year = 2013,
    month = nov,
   volume = 435,
    pages = {2152-2160},
      doi = {10.1093/mnras/stt1435},
   adsurl = {http://adsabs.harvard.edu/abs/2013MNRAS.435.2152K},
  adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}


@article{exoplanet:astropy13,
   author = {{Astropy Collaboration} and {Robitaille}, T.~P. and {Tollerud},
             E.~J. and {Greenfield}, P. and {Droettboom}, M. and {Bray}, E.␣
↪and
             {Aldcroft}, T. and {Davis}, M. and {Ginsburg}, A. and
             {Price-Whelan}, A.~M. and {Kerzendorf}, W.~E. and {Conley}, A.␣
↪and
             {Crighton}, N. and {Barbary}, K. and {Muna}, D. and {Ferguson},␣
↪H.
             and {Grollier}, F. and {Parikh}, M.~M. and {Nair}, P.~H. and
             {Unther}, H.~M. and {Deil}, C. and {Woillez}, J. and {Conseil},␣
↪S.
             and {Kramer}, R. and {Turner}, J.~E.~H. and {Singer}, L. and
             {Fox}, R. and {Weaver}, B.~A. and {Zabalza}, V. and {Edwards},
             Z.~I. and {Azalee Bostroem}, K. and {Burke}, D.~J. and {Casey},
             A.~R. and {Crawford}, S.~M. and {Dencheva}, N. and {Ely}, J. and
             {Jenness}, T. and {Labrie}, K. and {Lim}, P.~L. and
             {Pierfederici}, F. and {Pontzen}, A. and {Ptak}, A. and {Refsdal}
↪,
             B. and {Servillat}, M. and {Streicher}, O.},
    title = "{Astropy: A community Python package for astronomy}",
  journal = {aap},
     year = 2013,
    month = oct,
   volume = 558,
    pages = {A33},
      doi = {10.1051/0004-6361/201322068},
   adsurl = {http://adsabs.harvard.edu/abs/2013A%26A...558A..33A},
  adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}

@article{exoplanet:astropy18,
   author = {{Astropy Collaboration} and {Price-Whelan}, A.~M. and
             {Sip{H o}cz}, B.~M. and {G{"u}nther}, H.~M. and {Lim}, P.~L. and
             {Crawford}, S.~M. and {Conseil}, S. and {Shupe}, D.~L. and
             {Craig}, M.~W. and {Dencheva}, N. and {Ginsburg}, A. and
             {VanderPlas}, J.~T. and {Bradley}, L.~D. and
             {P{'e}rez-Su{'a}rez}, D. and {de Val-Borro}, M.
             and {Aldcroft}, T.~L. and {Cruz}, K.~L. and {Robitaille}, T.~P.
             and {Tollerud}, E.~J. and {Ardelean}, C. and {Babej}, T. and
             {Bach}, Y.~P. and {Bachetti}, M. and {Bakanov}, A.~V. and
             {Bamford}, S.~P. and {Barentsen}, G. and {Barmby}, P. and
             {Baumbach}, A. and {Berry}, K.~L.  and {Biscani}, F. and
             {Boquien}, M. and {Bostroem}, K.~A. and {Bouma}, L.~G. and
```

```
              {Brammer}, G.~B. and {Bray}, E.~M. and {Breytenbach}, H. and
              {Buddelmeijer}, H. and {Burke}, D.~J. and {Calderone}, G. and
              {Cano Rodr{'{i}}guez}, J.~L. and {Cara}, M. and {Cardoso},
              J.~V.~M. and {Cheedella}, S. and {Copin}, Y. and {Corrales}, L.
              and {Crichton}, D. and {D'Avella}, D. and {Deil}, C. and
              {Depagne}, {'E}. and {Dietrich}, J.~P. and {Donath}, A. and
              {Droettboom}, M. and {Earl}, N. and {Erben}, T. and {Fabbro}, S.
              and {Ferreira}, L.~A. and {Finethy}, T. and {Fox}, R.~T. and
              {Garrison}, L.~H. and {Gibbons}, S.~L.~J. and {Goldstein}, D.~A.
              and {Gommers}, R. and {Greco}, J.~P. and {Greenfield}, P. and
              {Groener}, A.~M. and {Grollier}, F. and {Hagen}, A. and {Hirst},
              P. and {Homeier}, D. and {Horton}, A.~J. and {Hosseinzadeh}, G.
              and {Hu}, L. and {Hunkeler}, J.~S. and {Ivezi{'c}}, {v Z}. and
              {Jain}, A. and {Jenness}, T. and {Kanarek}, G. and {Kendrew}, S.
              and {Kern}, N.~S. and {Kerzendorf}, W.~E. and {Khvalko}, A. and
              {King}, J. and {Kirkby}, D. and {Kulkarni}, A.~M. and {Kumar}, A.
              and {Lee}, A.  and {Lenz}, D.  and {Littlefair}, S.~P. and {Ma},
              Z. and {Macleod}, D.~M. and {Mastropietro}, M. and {McCully}, C.
              and {Montagnac}, S. and {Morris}, B.~M. and {Mueller}, M. and
              {Mumford}, S.~J. and {Muna}, D. and {Murphy}, N.~A. and {Nelson},
              S. and {Nguyen}, G.~H. and {Ninan}, J.~P. and {N{"o}the}, M. and
              {Ogaz}, S. and {Oh}, S. and {Parejko}, J.~K.  and {Parley}, N.␣
↪and
              {Pascual}, S. and {Patil}, R. and {Patil}, A.~A.  and {Plunkett},
              A.~L. and {Prochaska}, J.~X. and {Rastogi}, T. and {Reddy Janga},
              V. and {Sabater}, J.  and {Sakurikar}, P. and {Seifert}, M. and
              {Sherbert}, L.~E. and {Sherwood-Taylor}, H. and {Shih}, A.~Y. and
              {Sick}, J. and {Silbiger}, M.~T. and {Singanamalla}, S. and
              {Singer}, L.~P. and {Sladen}, P.~H. and {Sooley}, K.~A. and
              {Sornarajah}, S. and {Streicher}, O. and {Teuben}, P. and
              {Thomas}, S.~W. and {Tremblay}, G.~R. and {Turner}, J.~E.~H. and
              {Terr{'o}n}, V.  and {van Kerkwijk}, M.~H. and {de la Vega}, A.
              and {Watkins}, L.~L. and {Weaver}, B.~A. and {Whitmore}, J.~B.␣
↪and
              {Woillez}, J.  and {Zabalza}, V. and {Astropy Contributors}},
    title = "{The Astropy Project: Building an Open-science Project and Status
              of the v2.0 Core Package}",
  journal = {aj},
     year = 2018,
    month = sep,
   volume = 156,
    pages = {123},
      doi = {10.3847/1538-3881/aabc4f},
   adsurl = {http://adsabs.harvard.edu/abs/2018AJ....156..123A},
  adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}


@article{exoplanet:luger18,
   author = {{Luger}, R. and {Agol}, E. and {Foreman-Mackey}, D. and {Fleming}
↪,
              D.~P. and {Lustig-Yaeger}, J. and {Deitrick}, R.},
    title = "{STARRY: Analytic Occultation Light Curves}",
  journal = {ArXiv e-prints},
```

```
   eprint = {1810.06559},
     year = 2018,
    month = oct,
   adsurl = {http://adsabs.harvard.edu/abs/2018arXiv181006559L},
  adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

1.3 API documentation

1.3.1 Orbits

class exoplanet.orbits.**KeplerianOrbit**(*period=None*, *a=None*, *t0=0.0*, *incl=None*, *b=None*, *duration=None*, *ecc=None*, *omega=None*, *m_planet=0.0*, *m_star=None*, *r_star=None*, *rho_star=None*, *m_planet_units=None*, *rho_star_units=None*, *model=None*, *contact_points_kwargs=None*, ***kwargs*)

A system of bodies on Keplerian orbits around a common central

Given the input parameters, the values of all other parameters will be computed so a `KeplerianOrbit` instance will always have attributes for each argument. Note that the units of the computed attributes will all be in the standard units of this class (`R_sun`, `M_sun`, and `days`) except for `rho_star` which will be in `g / cm^3`.

There are only specific combinations of input parameters that can be used:

1. First, either `period` or `a` must be given. If values are given for both parameters, then neither `m_star` or `rho_star` can be defined because the stellar density implied by each planet will be computed in `rho_star`.
2. Only one of `incl` and `b` can be given.
3. If a value is given for `ecc` then `omega` must also be given.
4. If no stellar parameters are given, the central body is assumed to be the sun. If only `rho_star` is defined, the radius of the central is assumed to be `1 * R_sun`. Otherwise, at most two of `m_star`, `r_star`, and `rho_star` can be defined.

Parameters

- **period** – The orbital periods of the bodies in days.
- **a** – The semimajor axes of the orbits in `R_sun`.
- **t0** – The time of a reference transit for each orbits in days.
- **incl** – The inclinations of the orbits in radians.
- **b** – The impact parameters of the orbits.
- **ecc** – The eccentricities of the orbits. Must be `0 <= ecc < 1`.
- **omega** – The arguments of periastron for the orbits in radians.
- **m_planet** – The masses of the planets in units of `m_planet_units`.
- **m_star** – The mass of the star in `M_sun`.
- **r_star** – The radius of the star in `R_sun`.
- **rho_star** – The density of the star in units of `rho_star_units`.

- **m_planet_units** – An `astropy.units` compatible unit object giving the units of the planet masses. If not given, the default is `M_sun`.
- **rho_star_units** – An `astropy.units` compatible unit object giving the units of the stellar density. If not given, the default is `g / cm^3`.

get_planet_position(*t*)
The planets' positions in the barycentric frame

Parameters **t** – The times where the position should be evaluated.

Returns The components of the position vector at `t` in units of `R_sun`.

get_planet_velocity(*t*)
Get the planets' velocity vector

Parameters **t** – The times where the velocity should be evaluated.

Returns The components of the velocity vector at `t` in units of `M_sun/day`.

get_radial_velocity(*t*, *K=None*, *output_units=None*)
Get the radial velocity of the star

Parameters

- **t** – The times where the radial velocity should be evaluated.
- **K** (*Optional*) – The semi-amplitudes of the orbits. If provided, the `m_planet` and `incl` parameters will be ignored and this amplitude will be used instead.
- **output_units** (*Optional*) – An AstroPy velocity unit. If not given, the output will be evaluated in `m/s`. This is ignored if a value is given for `K`.

Returns The reflex radial velocity evaluated at `t` in units of `output_units`. For multiple planets, this will have one row for each planet.

get_relative_position(*t*)
The planets' positions relative to the star

Note: This treats each planet independently and does not take the other planets into account when computing the position of the star. This is fine as long as the planet masses are small.

Parameters **t** – The times where the position should be evaluated.

Returns The components of the position vector at `t` in units of `R_sun`.

get_star_position(*t*)
The star's position in the barycentric frame

Note: If there are multiple planets in the system, this will return one column per planet with each planet's contribution to the motion. The star's full position can be computed by summing over the last axis.

Parameters **t** – The times where the position should be evaluated.

Returns The components of the position vector at `t` in units of `R_sun`.

get_star_velocity(*t*)
Get the star's velocity vector

Note: For a system with multiple planets, this will return one column per planet with the contributions from each planet. The total velocity can be found by summing along the last axis.

Parameters **t** – The times where the velocity should be evaluated.

Returns The components of the velocity vector at `t` in units of `M_sun/day`.

in_transit(*t*, *r=0.0*, *texp=None*)
Get a list of timestamps that are in transit

Parameters

- **t** (*vector*) – A vector of timestamps to be evaluated.
- **r** (*Optional*) – The radii of the planets.
- **texp** (*Optional[float]*) – The exposure time.

Returns The indices of the timestamps that are in transit.

class exoplanet.orbits.**TTVOrbit**(**args*, ***kwargs*)
A generalization of a Keplerian orbit with transit timing variations

Only one of the arguments `ttvs` or `transit_times` can be given and the other will be computed from the one that was provided.

Parameters

- **ttvs** – A list (with on entry for each planet) of "O-C" vectors for each transit of each planet in units of days. "O-C" means the difference between the observed transit time and the transit time expected for a regular periodic orbit.
- **transit_times** – A list (with on entry for each planet) of transit times for each transit of each planet in units of days. These times will be used to compute the implied (least squares) `period` and `t0` so these parameters cannot also be given.

get_planet_position(*t*)
The planets' positions in the barycentric frame

Parameters **t** – The times where the position should be evaluated.

Returns The components of the position vector at `t` in units of `R_sun`.

get_planet_velocity(*t*)
Get the planets' velocity vector

Parameters **t** – The times where the velocity should be evaluated.

Returns The components of the velocity vector at `t` in units of `M_sun/day`.

get_radial_velocity(*t*, *K=None*, *output_units=None*)
Get the radial velocity of the star

Parameters

- **t** – The times where the radial velocity should be evaluated.
- **K** (*Optional*) – The semi-amplitudes of the orbits. If provided, the `m_planet` and `incl` parameters will be ignored and this amplitude will be used instead.

- **output_units** (*Optional*) – An AstroPy velocity unit. If not given, the output will be evaluated in `m/s`. This is ignored if a value is given for `K`.

Returns The reflex radial velocity evaluated at `t` in units of `output_units`. For multiple planets, this will have one row for each planet.

get_relative_position(*t*)
The planets' positions relative to the star

Note: This treats each planet independently and does not take the other planets into account when computing the position of the star. This is fine as long as the planet masses are small.

Parameters **t** – The times where the position should be evaluated.

Returns The components of the position vector at `t` in units of `R_sun`.

get_star_position(*t*)
The star's position in the barycentric frame

Note: If there are multiple planets in the system, this will return one column per planet with each planet's contribution to the motion. The star's full position can be computed by summing over the last axis.

Parameters **t** – The times where the position should be evaluated.

Returns The components of the position vector at `t` in units of `R_sun`.

get_star_velocity(*t*)
Get the star's velocity vector

Note: For a system with multiple planets, this will return one column per planet with the contributions from each planet. The total velocity can be found by summing along the last axis.

Parameters **t** – The times where the velocity should be evaluated.

Returns The components of the velocity vector at `t` in units of `M_sun/day`.

in_transit(*t*, *r=0.0*, *texp=None*)
Get a list of timestamps that are in transit

Parameters

- **t** (*vector*) – A vector of timestamps to be evaluated.
- **r** (*Optional*) – The radii of the planets.
- **texp** (*Optional[float]*) – The exposure time.

Returns The indices of the timestamps that are in transit.

class exoplanet.orbits.**SimpleTransitOrbit**(*period=None*, *t0=0.0*, *b=0.0*, *duration=None*, *r_star=1.0*)
An orbit representing a set of planets transiting a common central

This orbit is parameterized by the observables of a transiting system, period, phase, duration, and impact parameter.

Parameters

- **period** – The orbital period of the planets in days.
- **t0** – The midpoint time of a reference transit for each planet in days.
- **b** – The impact parameters of the orbits.
- **duration** – The durations of the transits in days.
- **r_star** – The radius of the star in `R_sun`.

get_relative_position(*t*)

The planets' positions relative to the star

Parameters **t** – The times where the position should be evaluated.

Returns The components of the position vector at `t` in units of `R_sun`.

in_transit(*t*, *r=None*, *texp=None*)

Get a list of timestamps that are in transit

Parameters

- **t** (*vector*) – A vector of timestamps to be evaluated.
- **r** (*Optional*) – The radii of the planets.
- **texp** (*Optional[float]*) – The exposure time.

Returns The indices of the timestamps that are in transit.

1.3.2 Light curve models

class exoplanet.**StarryLightCurve**(*u*, *model=None*)

A limb darkened light curve computed using starry

Parameters **u** (*vector*) – A vector of limb darkening coefficients.

get_light_curve(*orbit=None*, *r=None*, *t=None*, *texp=None*, *oversample=7*, *order=0*, *use_in_transit=True*)

Get the light curve for an orbit at a set of times

Parameters

- **orbit** – An object with a `get_relative_position` method that takes a tensor of times and returns a list of Cartesian coordinates of a set of bodies relative to the central source. This method should return three tensors (one for each coordinate dimension) and each tensor should have the shape `append(t.shape, r.shape)` or `append(t.shape, oversample, r.shape)` when `texp` is given. The first two coordinate dimensions are treated as being in the plane of the sky and the third coordinate is the line of sight with positive values pointing *away* from the observer. For an example, take a look at *orbits.KeplerianOrbit*.
- **r** (*tensor*) – The radius of the transiting body in the same units as `r_star`. This should have a shape that is consistent with the coordinates returned by `orbit`. In general, this means that it should probably be a scalar or a vector with one entry for each body in `orbit`.
- **t** (*tensor*) – The times where the light curve should be evaluated.
- **texp** (*Optional[tensor]*) – The exposure time of each observation. This can be a scalar or a tensor with the same shape as `t`. If `texp` is provided, `t` is assumed to indicate the timestamp at the *middle* of an exposure of length `texp`.

- **oversample** (*Optional[int]*) – The number of function evaluations to use when numerically integrating the exposure time.
- **order** (*Optional[int]*) – The order of the numerical integration scheme. This must be one of the following: `0` for a centered Riemann sum (equivalent to the "resampling" procedure suggested by Kipping 2010), `1` for the trapezoid rule, or `2` for Simpson's rule.
- **use_in_transit** (*Optional[bool]*) – If `True`, the model will only be evaluated for the data points expected to be in transit as computed using the `in_transit` method on `orbit`.

1.3.3 Scalable Gaussian processes

class exoplanet.gp.**GP**(*kernel*, *x*, *diag*, *J=-1*, *model=None*)

class exoplanet.gp.terms.**Term**(***kwargs*)
The abstract base "term" that is the superclass of all other terms

Subclasses should overload the `terms.Term.get_real_coefficients()` and `terms.Term.get_complex_coefficients()` methods.

class exoplanet.gp.terms.**RealTerm**(***kwargs*)
The simplest celerite term

This term has the form

$$k(\tau) = a_j\, e^{-c_j\,\tau}$$

with the parameters `a` and `c`.

Strictly speaking, for a sum of terms, the parameter `a` could be allowed to go negative but since it is somewhat subtle to ensure positive definiteness, we recommend keeping both parameters strictly positive. Advanced users can build a custom term that has negative coefficients but care should be taken to ensure positivity.

Parameters

- **a or log_a** (*tensor*) – The amplitude of the term.
- **c or log_c** (*tensor*) – The exponent of the term.

class exoplanet.gp.terms.**ComplexTerm**(***kwargs*)
A general celerite term

This term has the form

$$k(\tau) = \frac{1}{2}\left[(a_j + b_j)\, e^{-(c_j+d_j)\,\tau} + (a_j - b_j)\, e^{-(c_j-d_j)\,\tau}\right]$$

with the parameters `a`, `b`, `c`, and `d`.

This term will only correspond to a positive definite kernel (on its own) if $a_j\, c_j \geq b_j\, d_j$.

Parameters

- **a or log_a** (*tensor*) – The real part of amplitude.
- **b or log_b** (*tensor*) – The imaginary part of amplitude.
- **c or log_c** (*tensor*) – The real part of the exponent.
- **d or log_d** (*tensor*) – The imaginary part of exponent.

class exoplanet.gp.terms.**SHOTerm**(**args*, ***kwargs*)
A term representing a stochastically-driven, damped harmonic oscillator

The PSD of this term is

$$S(\omega) = \sqrt{\frac{2}{\pi}} \frac{S_0\,\omega_0^4}{(\omega^2 - {\omega_0}^2)^2 + {\omega_0}^2\,\omega^2/Q^2}$$

with the parameters `S0`, `Q`, and `w0`.

Parameters

- **S0 or log_S0** (*tensor*) – The parameter S_0.
- **Q or log_Q** (*tensor*) – The parameter Q.
- **w0 or log_w0** (*tensor*) – The parameter ω_0.

class exoplanet.gp.terms.**Matern32Term**(***kwargs*)
A term that approximates a Matern-3/2 function

This term is defined as

$$k(\tau) = \sigma^2 \left[(1 + 1/\epsilon)\, e^{-(1-\epsilon)\sqrt{3}\,\tau/\rho} (1 - 1/\epsilon)\, e^{-(1+\epsilon)\sqrt{3}\,\tau/\rho}\right]$$

with the parameters `sigma` and `rho`. The parameter `eps` controls the quality of the approximation since, in the limit $\epsilon \to 0$ this becomes the Matern-3/2 function

$$\lim_{\epsilon\to 0} k(\tau) = \sigma^2 \left(1 + \frac{\sqrt{3}\,\tau}{\rho}\right) \exp\left(-\frac{\sqrt{3}\,\tau}{\rho}\right)$$

Parameters

- **sigma or log_sigma** (*tensor*) – The parameter σ.
- **rho or log_rho** (*tensor*) – The parameter ρ.
- **eps** (*Optional[float]*) – The value of the parameter ϵ. (default: *0.01*)

class exoplanet.gp.terms.**RotationTerm**(***kwargs*)
A mixture of two SHO terms that can be used to model stellar rotation

This term has two modes in Fourier space: one at `period` and one at `0.5 * period`. This can be a good descriptive model for a wide range of stochastic variability in stellar time series from rotation to pulsations.

Parameters

- **amp or log_amp** (*tensor*) – The amplitude of the variability.
- **period or log_period** (*tensor*) – The primary period of variability.
- **Q0 or log_Q0** (*tensor*) – The quality factor (or really the quality factor minus one half) for the secondary oscillation.
- **deltaQ or log_deltaQ** (*tensor*) – The difference between the quality factors of the first and the second modes. This parameterization (if `deltaQ > 0`) ensures that the primary mode alway has higher quality.
- **mix** – The fractional amplitude of the secondary mode compared to the primary. This should probably always be `0 < mix < 1`.

1.3.4 Estimators

exoplanet.**estimate_semi_amplitude**(*periods*, *x*, *y*, *yerr=None*, *t0s=None*)
Estimate the RV semi-amplitudes for planets in an RV series

Parameters

- **periods** – The periods of the planets. Assumed to be in `days` if not an AstroPy Quantity.
- **x** – The observation times. Assumed to be in `days` if not an AstroPy Quantity.
- **y** – The radial velocities. Assumed to be in `m/s` if not an AstroPy Quantity.
- **yerr** (*Optional*) – The uncertainty on `y`.
- **t0s** (*Optional*) – The time of a reference transit for each planet, if known.

Returns An estimate of the semi-amplitude of each planet in units of `m/s`.

exoplanet.**estimate_minimum_mass**(*periods*, *x*, *y*, *yerr=None*, *t0s=None*, *m_star=1*)
Estimate the minimum mass(es) for planets in an RV series

Parameters

- **periods** – The periods of the planets. Assumed to be in `days` if not an AstroPy Quantity.
- **x** – The observation times. Assumed to be in `days` if not an AstroPy Quantity.
- **y** – The radial velocities. Assumed to be in `m/s` if not an AstroPy Quantity.
- **yerr** (*Optional*) – The uncertainty on `y`.
- **t0s** (*Optional*) – The time of a reference transit for each planet, if known.
- **m_star** (*Optional*) – The mass of the star. Assumed to be in `M_sun` if not an AstroPy Quantity.

Returns An estimate of the minimum mass of each planet as an AstroPy Quantity with units of `M_jupiter`.

exoplanet.**lomb_scargle_estimator**(*x*, *y*, *yerr=None*, *min_period=None*, *max_period=None*, *filter_period=None*, *max_peaks=2*, ***kwargs*)
Estimate period of a time series using the periodogram

Parameters

- **x** (*ndarray[N]*) – The times of the observations
- **y** (*ndarray[N]*) – The observations at times `x`
- **yerr** (*Optional[ndarray[N]]*) – The uncertainties on `y`
- **min_period** (*Optional[float]*) – The minimum period to consider
- **max_period** (*Optional[float]*) – The maximum period to consider
- **filter_period** (*Optional[float]*) – If given, use a high-pass filter to down-weight period longer than this
- **max_peaks** (*Optional[int]*) – The maximum number of peaks to return (default: 2)

Returns A dictionary with the computed `periodogram` and the parameters for up to `max_peaks` peaks in the periodogram.

exoplanet.**autocorr_estimator**(*x*, *y*, *yerr=None*, *min_period=None*, *max_period=None*, *oversample=2.0*, *smooth=2.0*, *max_peaks=10*)

Estimate the period of a time series using the autocorrelation function

---

Note: The signal is interpolated onto a uniform grid in time so that the autocorrelation function can be computed.

Parameters

- **x** (*ndarray[N]*) – The times of the observations
- **y** (*ndarray[N]*) – The observations at times `x`
- **yerr** (*Optional[ndarray[N]]*) – The uncertainties on `y`
- **min_period** (*Optional[float]*) – The minimum period to consider
- **max_period** (*Optional[float]*) – The maximum period to consider
- **oversample** (*Optional[float]*) – When interpolating, oversample the times by this factor (default: 2.0)
- **smooth** (*Optional[float]*) – Smooth the autocorrelation function by this factor times the minimum period (default: 2.0)
- **max_peaks** (*Optional[int]*) – The maximum number of peaks to identify in the autocorrelation function (default: 10)

Returns A dictionary with the computed autocorrelation function and the estimated period. For compatibility with the *lomb_scargle_estimator()*, the period is returned as a list with the key `peaks`.

1.3.5 Distributions

class exoplanet.distributions.**UnitVector**(**args*, ***kwargs*)

A vector where the sum of squares is fixed to unity

For a multidimensional shape, the normalization is performed along the last dimension.

class exoplanet.distributions.**Angle**(**args*, ***kwargs*)

An angle constrained to be in the range -pi to pi

The actual sampling is performed in the two dimensional vector space `(sin(theta), cos(theta))` so that the sampler doesn't see a discontinuity at pi.

class exoplanet.distributions.**QuadLimbDark**(**args*, ***kwargs*)

An uninformative prior for quadratic limb darkening parameters

This is an implementation of the Kipping (2013) reparameterization of the two-parameter limb darkening model to allow for efficient and uninformative sampling.

class exoplanet.distributions.**RadiusImpact**(**args*, ***kwargs*)

The Espinoza (2018) distribution over radius and impact parameter

This is an implementation of Espinoza (2018) The first axis of the shape of the parameter should be exactly 2. The radius ratio will be in the zeroth entry in the first dimension and the impact parameter will be in the first.

Parameters

- **min_radius** – The minimum allowed radius.

- **max_radius** – The maximum allowed radius.

exoplanet.distributions.**get_joint_radius_impact**(*name=''*, *N_planets=None*, *min_radius=0*, *max_radius=1*, *testval_r=None*, *testval_b=None*, ***kwargs*)

Get the joint distribution over radius and impact parameter

This uses the Espinoza (2018) parameterization of the distribution (see `distributions.RadiusImpact` for more details).

Parameters

- **name** (*Optional[str]*) – A prefix that is added to all distribution names used in this parameterization. For example, if `name` is `param_`, vars will be added to the PyMC3 model with names `param_rb` (for the joint distribution), `param_b`, and `param_r`.
- **N_planets** (*Optional[int]*) – The number of planets. If not provided, it will be inferred from the `testval_*` parameters or assumed to be 1.
- **min_radius** (*Optional[float]*) – The minimum allowed radius.
- **max_radius** (*Optional[float]*) – The maximum allowed radius.
- **testval_r** (*Optional[float or array]*) – An initial guess for the radius parameter. This should be a `float` or an array with `N_planets` entries.
- **testval_b** (*Optional[float or array]*) – An initial guess for the impact parameter. This should be a `float` or an array with `N_planets` entries.

Returns Two `pymc3.Deterministic` variables for the planet radius and impact parameter.

1.3.6 Utilities

exoplanet.**optimize**(*start=None*, *vars=None*, *model=None*, *return_info=False*, *verbose=True*, ***kwargs*)

Maximize the log prob of a PyMC3 model using scipy

All extra arguments are passed directly to the `scipy.optimize.minimize` function.

Parameters

- **start** – The PyMC3 coordinate dictionary of the starting position
- **vars** – The variables to optimize
- **model** – The PyMC3 model
- **return_info** – Return both the coordinate dictionary and the result of `scipy.optimize.minimize`
- **verbose** – Print the success flag and log probability to the screen

exoplanet.**eval_in_model**(*var*, *point=None*, *return_func=False*, *model=None*, ***kwargs*)

Evaluate a Theano tensor or PyMC3 variable in a PyMC3 model

This method builds a Theano function for evaluating a node in the graph given the required parameters. This will also cache the compiled Theano function in the current `pymc3.Model` to reduce the overhead of calling this function many times.

Parameters

- **var** – The variable or tensor to evaluate.

- **point** (*Optional*) – A `dict` of input parameter values. This can be `model.test_point` (default), the result of `pymc3.find_MAP`, a point in a `pymc3.MultiTrace` or any other representation of the input parameters.
- **return_func** (*Optional[bool]*) – If `False` (default), return the evaluated variable. If `True`, return the result, the Theano function and the list of arguments for that function.

Returns Depending on `return_func`, either the value of `var` at `point`, or this value, the Theano function, and the input arguments.

`exoplanet.`**get_samples_from_trace**(*trace*, *size=1*)
Generate random samples from a PyMC3 MultiTrace

Parameters

- **trace** – The `MultiTrace`.
- **size** – The number of samples to generate.

class `exoplanet.`**PyMC3Sampler**(*start=75*, *finish=50*, *window=25*, *dense=True*)
A sampling wrapper for PyMC3 with support for dense mass matrices

This schedule is based on the method used by as described in Section 34.2 of the Stan Manual.

Parameters

- **start** (*int*) – The number of steps to run as an initial burn-in to find the typical set.
- **window** (*int*) – The length of the first mass matrix tuning phase. Subsequent tuning windows will be powers of two times this size.
- **finish** (*int*) – The number of tuning steps to run to learn the step size after tuning the mass matrix.
- **dense** (*bool*) – Fit for the off-diagonal elements of the mass matrix.

extend_tune(*steps*, *start=None*, *step_kwargs=None*, *trace=None*, *step=None*, ***kwargs*)
Extend the tuning phase by a given number of steps

After running the sampling, the mass matrix is re-estimated based on this run.

Parameters **steps** (*int*) – The number of steps to run.

get_step_for_trace(*trace=None*, *model=None*, *regular_window=0*, *regular_variance=0.001*, ***kwargs*)
Get a PyMC3 NUTS step tuned for a given burn-in trace

Parameters

- **trace** – The `MultiTrace` output from a previous run of `pymc3.sample`.
- **regular_window** – The weight (in units of number of steps) to use when regularizing the mass matrix estimate.
- **regular_variance** – The amplitude of the regularization for the mass matrix. This will be added to the diagonal of the covariance matrix with weight given by `regular_window`.

sample(*trace=None*, *step=None*, *start=None*, *step_kwargs=None*, ***kwargs*)
Run the production sampling using the tuned mass matrix

This is a light wrapper around `pymc3.sample` and any arguments used there (for example `draws`) can be used as input to this method too.

tune(*tune=1000*, *start=None*, *step_kwargs=None*, ***kwargs*)
Run the full tuning run for the mass matrix

This will run `start` steps of warmup followed by chains with exponentially increasing chains to tune the mass matrix.

Parameters **tune** (*int*) – The total number of steps to run.

warmup(*start=None*, *step_kwargs=None*, ***kwargs*)
Run an initial warmup phase to find the typical set

1.3.7 Citations

exoplanet.citations.**get_citations_for_model**(*model=None*, *width=79*)
Get the citations for the components used an exoplanet PyMC3

Returns: The acknowledgement text for exoplanet and its dependencies and a string containing the BibTeX entries for the citations in the acknowledgement.

CHAPTER 2

Tutorials

Note: This tutorial was generated from an IPython notebook that can be downloaded here.

```
theano version: 1.0.4
pymc3 version: 3.6
exoplanet version: 0.1.5
```

2.1 A quick intro to PyMC3 for exoplaneteers

Hamiltonian Monte Carlo (HMC) methods haven't been widely used in astrophysics, but they are the standard methods for probabilistic inference using Markov chain Monte Carlo (MCMC) in many other fields. *exoplanet* is designed to provide the building blocks for fitting many exoplanet datasets using this technology, and this tutorial presents some of the basic features of the PyMC3 modeling language and inference engine. The documentation for PyMC3 includes many other tutorials that you should check out to get more familiar with the features that are available.

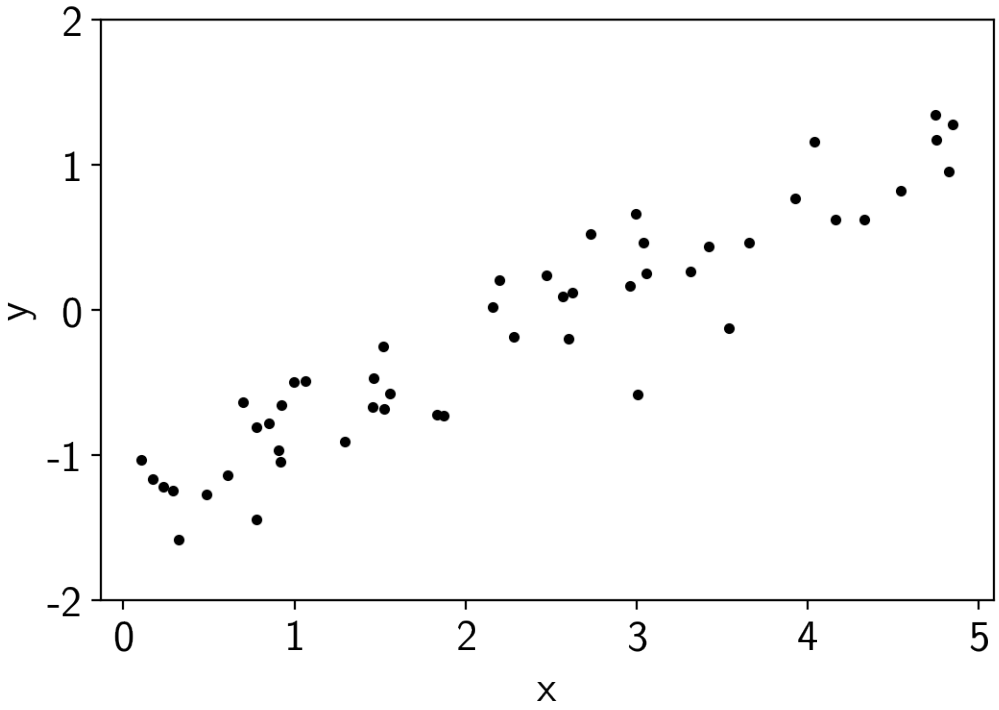
In this tutorial, we will go through two simple examples of fitting some data using PyMC3. The first is the classic fitting a line to data with unknown error bars, and the second is a more relevant example where we fit a radial velocity model to the public radial velocity observations of 51 Peg. You can read more about fitting lines to data in the bible of line fitting and you can see another example of fitting the 51 Peg data using HMC (this time using Stan) here.

2.1.1 Hello world (AKA fitting a line to data)

My standard intro to a new modeling language or inference framework is to fit a line to data. So. Let's do that with PyMC3.

To start, we'll generate some fake data using a linear model. Feel free to change the random number seed to try out a different dataset.

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

np.random.seed(42)

true_m = 0.5
true_b = -1.3
true_logs = np.log(0.3)

x = np.sort(np.random.uniform(0, 5, 50))
y = true_b + true_m * x + np.exp(true_logs) * np.random.randn(len(x))

plt.plot(x, y, ".k")
plt.ylim(-2, 2)
plt.xlabel("x")
plt.ylabel("y");
```



To fit a model to these data, our model will have 3 parameters: the slope m, the intercept b, and the log of the

uncertainty $\log(\sigma)$. To start, let's choose broad uniform priors on these parameters:

$$
\begin{aligned}
& & to \\
& & p(m) = \\
& \begin{cases} 1/10 & \text{if } -5 < m < 5 \\ 0 & \text{otherwise} \end{cases} \\
& & p(b) = \\
& \begin{cases} 1/10 & \text{if } -5 < b < 5 \\ 0 & \text{otherwise} \end{cases} \\
& & p(\log(\sigma)) = \\
& \begin{cases} 1/10 & \text{if } -5 < b < 5 \\ 0 & \text{otherwise} \end{cases} & (2.1)
\end{aligned}
$$

$$
\begin{aligned}
&= \\
&= \begin{cases} 1/10 & \text{if } -5 < m < 5 \\ 0 & \text{otherwise} \end{cases} p(b) \\
&= \begin{cases} 1/10 & \text{if } -5 < b < 5 \\ 0 & \text{otherwise} \end{cases} p(\log(\sigma)) \\
& \begin{cases} 1/10 & \text{if } -5 < b < 5 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

Then, the log-likelihood function will be

$$\log p(\{y_n\} \,|\, m,\, b,\, \log(\sigma)) = -\frac{1}{2} \sum_{n=1}^{N} \left[\frac{(y_n - m\, x_n - b)^2}{\sigma^2} + \log(2\,\pi\,\sigma^2) \right]$$

[**Note:** the second normalization term is needed in this model because we are fitting for σ and the second term is *not* a constant.]

Another way of writing this model that might not be familiar is the following:

$$
\begin{aligned}
&to \\
\\
m &\sim \text{Uniform}(-5,\, 5) \\
b &\sim \text{Uniform}(-5,\, 5) \\
\log(\sigma) &\sim \text{Uniform}(-5,\, 5) \\
y_n &\sim \text{Normal}(m\, x_n + b,\, \sigma)
\end{aligned} \tag{2.1}
$$

$\text{Uniform}(-5,\, 5) b$
$\text{Uniform}(-5,\, 5) \log(\sigma)$
$\text{Uniform}(-5,\, 5) y_n$
$\text{Normal}(m\, x_n + b,\, \sigma)$

This is the way that a model like this is often defined in statistics and it will be useful when we implement out model in PyMC3 so take a moment to make sure that you understand the notation.

Now, let's implement this model in PyMC3. The documentation for the distributions available in PyMC3's modeling language can be found here and these will come in handy as you go on to write your own models.

```
import pymc3 as pm

with pm.Model() as model:

    # Define the priors on each parameter:
    m = pm.Uniform("m", lower=-5, upper=5)
    b = pm.Uniform("b", lower=-5, upper=5)
    logs = pm.Uniform("logs", lower=-5, upper=5)

    # Define the likelihood. A few comments:
    #  1. For mathematical operations like "exp", you can't use
    #     numpy. Instead, use the mathematical operations defined
    #     in "pm.math".
    #  2. To condition on data, you use the "observed" keyword
    #     argument to any distribution. In this case, we want to
    #     use the "Normal" distribution (look up the docs for
    #     this).
    pm.Normal("obs", mu=m*x+b, sd=pm.math.exp(logs), observed=y)
```

(continues on next page)

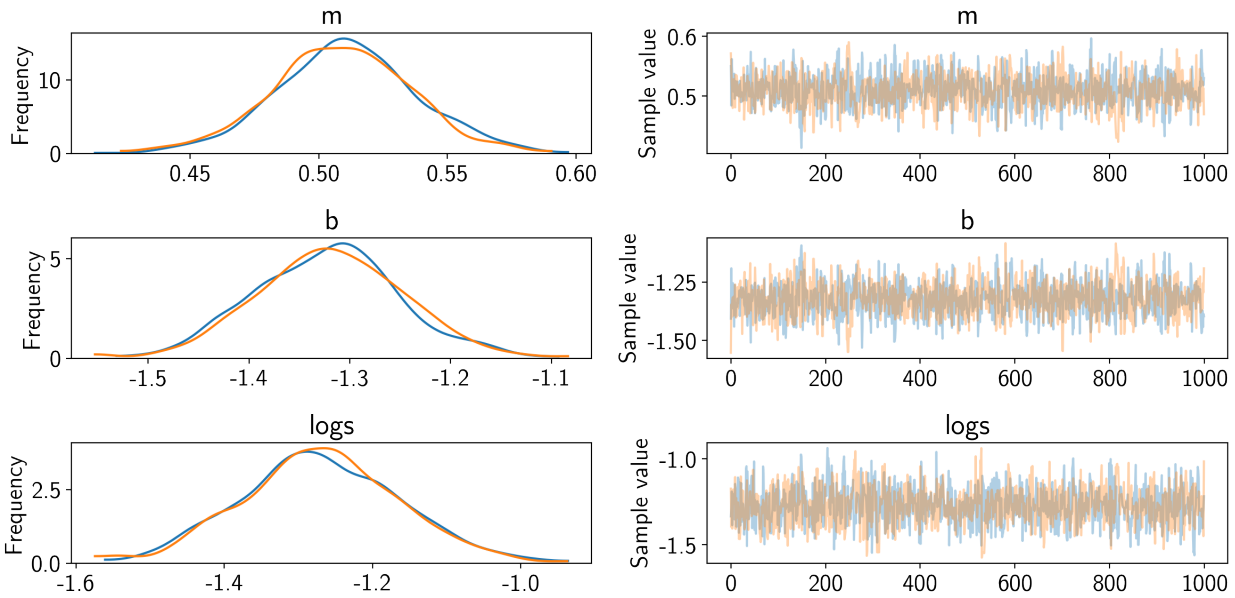
(continued from previous page)

```
    # This is how you will sample the model. Take a look at the
    # docs to see that other parameters that are available.
    trace = pm.sample(draws=1000, tune=1000, chains=2)
```

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (2 chains in 4 jobs)
NUTS: [logs, b, m]
Sampling 2 chains: 100%|| 4000/4000 [00:02<00:00, 1825.93draws/s]
```

Now since we now have samples, let's make some diagnostic plots. The first plot to look at is the "traceplot" implemented in PyMC3. In this plot, you'll see the marginalized distribution for each parameter on the left and the trace plot (parameter value as a function of step number) on the right. In each panel, you should see two lines with different colors. These are the results of different independent chains and if the results are substantially different in the different chains then there is probably something going wrong.

```
pm.traceplot(trace, varnames=["m", "b", "logs"]);
```

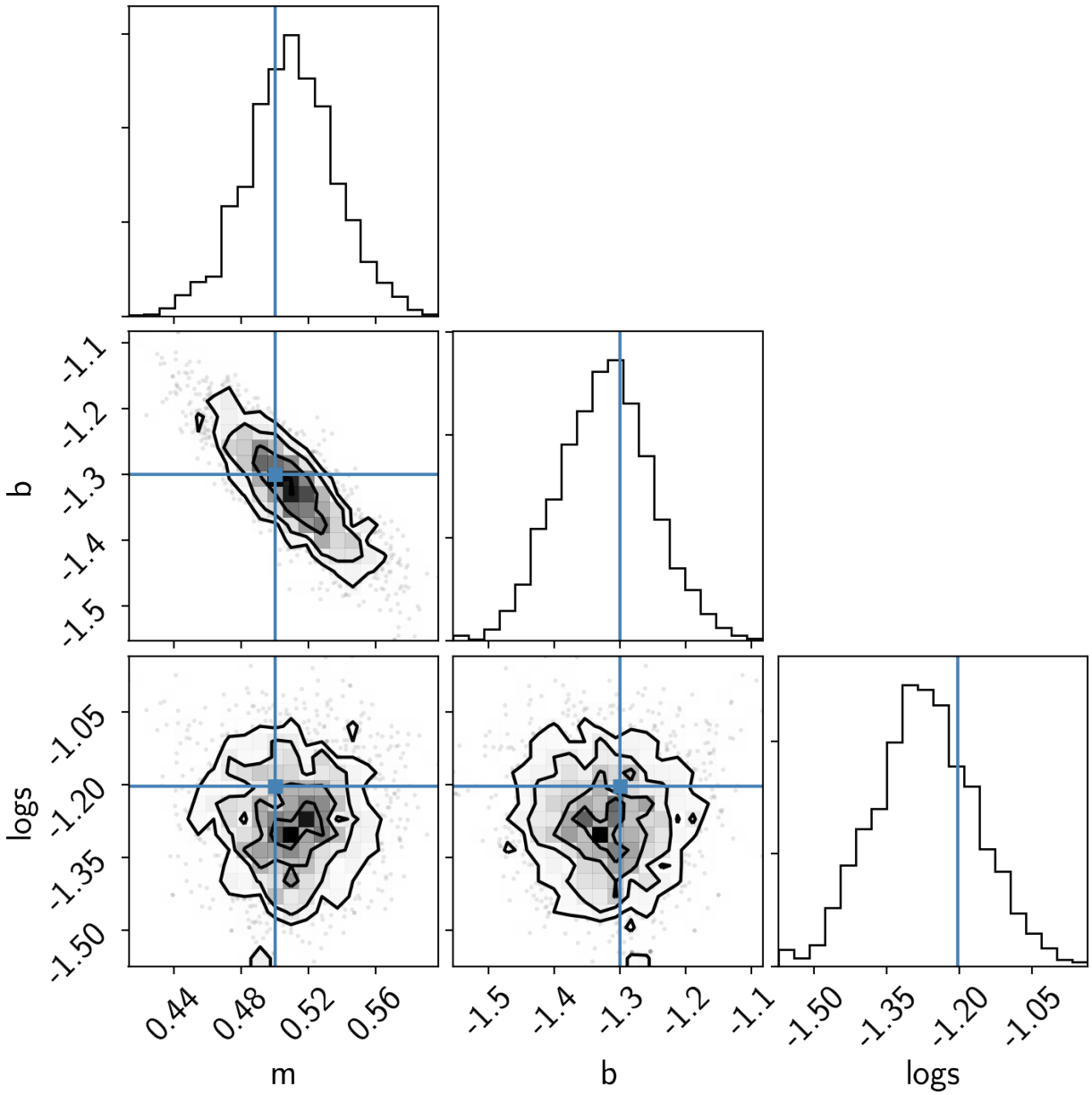


It's also good to quantify that "looking substantially different" argument. This is implemented in PyMC3 as the "summary" function. In this table, some of the key columns to look at are `n_eff` and `Rhat`. * `n_eff` shows an estimate of the number of effective (or independent) samples for that parameter. In this case, `n_eff` should probably be around 500 per chain (there should have been 2 chains run). * `Rhat` shows the Gelman–Rubin statistic and it should be close to 1.

```
pm.summary(trace, varnames=["m", "b", "logs"])
```

The last diagnostic plot that we'll make here is the corner plot made using corner.py. The easiest way to do this using PyMC3 is to first convert the trace to a Pandas DataFrame and then pass that to `corner.py`.

```
import corner  # https://corner.readthedocs.io

samples = pm.trace_to_dataframe(trace, varnames=["m", "b", "logs"])
corner.corner(samples, truths=[true_m, true_b, true_logs]);
```



Extra credit: Here are a few suggestions for things to try out while getting more familiar with PyMC3:

1. Try initializing the parameters using the `testval` argument to the distributions. Does this improve performance in this case? It will substantially improve performance in more complicated examples.
2. Try changing the priors on the parameters. For example, try the "uninformative" prior recommended by Jake VanderPlas on his blog.
3. What happens as you substantially increase or decrease the simulated noise? Does the performance change significantly? Why?

2.1.2 A more realistic example: radial velocity exoplanets

While the above example was cute, it doesn't really fully exploit the power of PyMC3 and it doesn't really show some of the real issues that you will face when you use PyMC3 as an astronomer. To get a better sense of how you might

use PyMC3 in Real Life™, let's take a look at a more realistic example: fitting a Keplerian orbit to radial velocity observations.

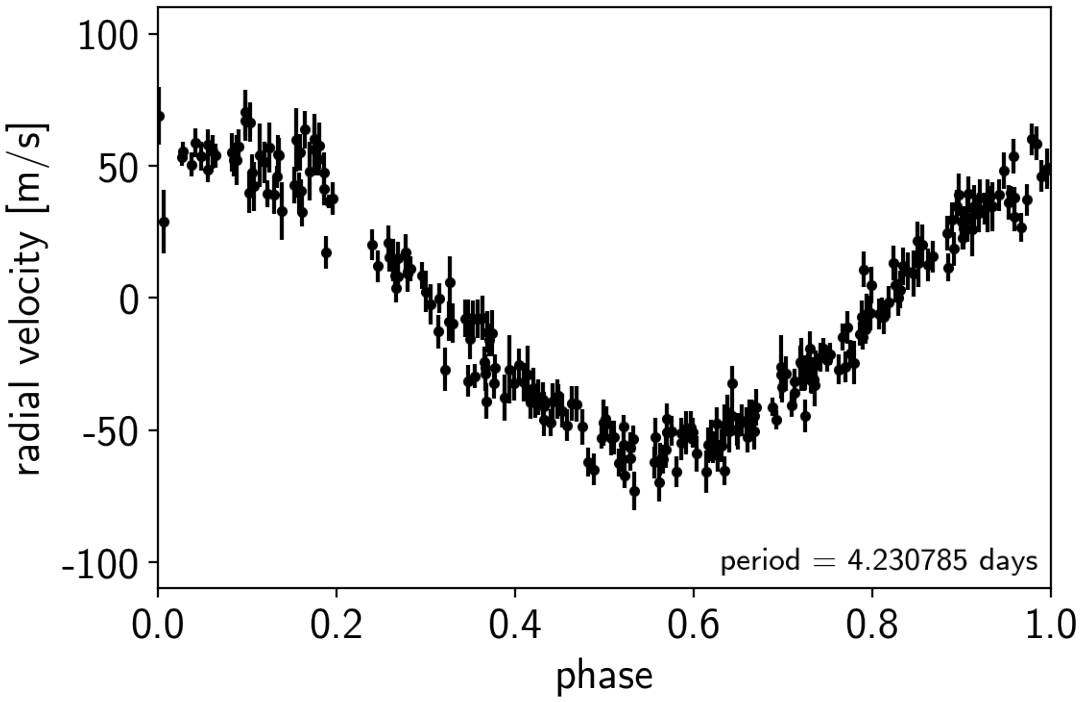
One of the key aspects of this problem that I want to highlight is the fact that PyMC3 (and the underlying model building framework Theano) don't have out-of-the-box support for the root-finding that is required to solve Kepler's equation. As part of the process of computing a Keplerian RV model, we must solve the equation:

$$M = E - e \sin E$$

for the eccentric anomaly E given some mean anomaly M and eccentricity e. There are commonly accepted methods of solving this equation using Newton's method, but if we want to expose that to PyMC3, we have to define a custom Theano operation with a custom gradient. I won't go into the details of the math (because I blogged about it) and I won't go into the details of the implementation (because you can take a look at it on GitHub). So, for this tutorial, we'll use the custom Kepler solver that is implemented as part of *exoplanet* and fit the publicly available radial velocity observations of the famous exoplanetary system 51 Peg using PyMC3.

First, we need to download the data from the exoplanet archive:

```
import requests
import pandas as pd
import matplotlib.pyplot as plt

# Download the dataset from the Exoplanet Archive:
url = "https://exoplanetarchive.ipac.caltech.edu/data/ExoData/0113/0113357/data/UID_
↪0113357_RVC_001.tbl"
r = requests.get(url)
if r.status_code != requests.codes.ok:
    r.raise_for_status()
data = np.array([l.split() for l in r.text.splitlines()
                 if not l.startswith("\\") and not l.startswith("|")],
                dtype=float)
t, rv, rv_err = data.T
t -= np.mean(t)

# Plot the observations "folded" on the published period:
# Butler et al. (2006) https://arxiv.org/abs/astro-ph/0607493
lit_period = 4.230785
plt.errorbar((t % lit_period)/lit_period, rv, yerr=rv_err, fmt=".k", capsize=0)
plt.xlim(0, 1)
plt.ylim(-110, 110)
plt.annotate("period = {0:.6f} days".format(lit_period),
             xy=(1, 0), xycoords="axes fraction",
             xytext=(-5, 5), textcoords="offset points",
             ha="right", va="bottom", fontsize=12)
plt.ylabel("radial velocity [m/s]")
plt.xlabel("phase");
```



Now, here's the implementation of a radial velocity model in PyMC3. Some of this will look familiar after the Hello World example, but things are a bit more complicated now. Take a minute to take a look through this and see if you can follow it. There's a lot going on, so I want to point out a few things to pay attention to:

1. All of the mathematical operations (for example `exp` and `sqrt`) are being performed using Theano instead of NumPy.
2. All of the parameters have initial guesses provided. This is an example where this makes a big difference because some of the parameters (like period) are very tightly constrained.
3. Some of the lines are wrapped in `Deterministic` distributions. This can be useful because it allows us to track values as the chain progresses even if they're not parameters. For example, after sampling, we will have a sample for `bkg` (the background RV trend) for each step in the chain. This can be especially useful for making plots of the results.
4. Similarly, at the end of the model definition, we compute the RV curve for a single orbit on a fine grid. This can be very useful for diagnosing fits gone wrong.
5. For parameters that specify angles (like ω, called `w` in the model below), it can be inefficient to sample in the angle directly because of the fact that the value wraps around at 2π. Instead, it can be better to sample the unit vector specified by the angle. In practice, this can be achieved by sampling a 2-vector from an isotropic Gaussian and normalizing the components by the norm. This is implemented as part of *exoplanet* in the *exoplanet.distributions.Angle* class.

```
import theano.tensor as tt

from exoplanet.orbits import get_true_anomaly
from exoplanet.distributions import Angle

with pm.Model() as model:

    # Parameters
    logK = pm.Uniform("logK", lower=0, upper=np.log(200),
```

(continues on next page)

(continued from previous page)

```
                    testval=np.log(0.5*(np.max(rv) - np.min(rv))))
    logP = pm.Uniform("logP", lower=0, upper=np.log(10),
                      testval=np.log(lit_period))
    phi = pm.Uniform("phi", lower=0, upper=2*np.pi, testval=0.1)
    e = pm.Uniform("e", lower=0, upper=1, testval=0.1)
    w = Angle("w")
    logjitter = pm.Uniform("logjitter", lower=-10, upper=5,
                           testval=np.log(np.mean(rv_err)))
    rv0 = pm.Normal("rv0", mu=0.0, sd=10.0, testval=0.0)
    rvtrend = pm.Normal("rvtrend", mu=0.0, sd=10.0, testval=0.0)

    # Deterministic transformations
    n = 2*np.pi*tt.exp(-logP)
    P = pm.Deterministic("P", tt.exp(logP))
    K = pm.Deterministic("K", tt.exp(logK))
    cosw = tt.cos(w)
    sinw = tt.sin(w)
    s2 = tt.exp(2*logjitter)
    t0 = (phi + w) / n

    # The RV model
    bkg = pm.Deterministic("bkg", rv0 + rvtrend * t / 365.25)
    M = n * t - (phi + w)

    # This is the line that uses the custom Kepler solver
    f = get_true_anomaly(M, e + tt.zeros_like(M))
    rvmodel = pm.Deterministic(
        "rvmodel", bkg + K * (cosw*(tt.cos(f) + e) - sinw*tt.sin(f)))

    # Condition on the observations
    err = tt.sqrt(rv_err**2 + tt.exp(2*logjitter))
    pm.Normal("obs", mu=rvmodel, sd=err, observed=rv)

    # Compute the phased RV signal
    phase = np.linspace(0, 1, 500)
    M_pred = 2*np.pi * phase - (phi + w)
    f_pred = get_true_anomaly(M_pred, e + tt.zeros_like(M_pred))
    rvphase = pm.Deterministic(
        "rvphase", K * (cosw*(tt.cos(f_pred) + e) - sinw*tt.sin(f_pred)))
```

In this case, I've found that it is useful to first optimize the parameters to find the "maximum a posteriori" (MAP) parameters and then start the sampler from there. This is useful here because MCMC is not designed to *find* the maximum of the posterior; it's just meant to sample the shape of the posterior. The performance of all MCMC methods can be really bad when the initialization isn't good (especially when some parameters are very well constrained). To find the maximum a posteriori parameters using PyMC3, you can use the `exoplanet.optimize()` function:

```
from exoplanet import optimize
with model:
    map_params = optimize()
```

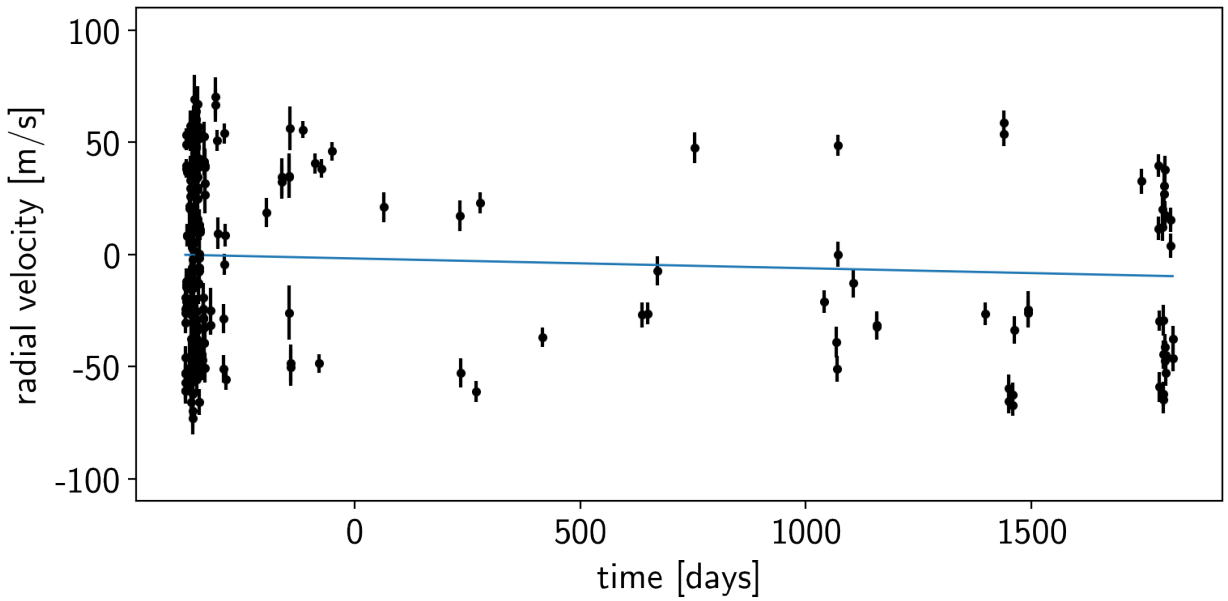
```
optimizing logp for variables: ['rvtrend', 'rv0', 'logjitter_interval__', 'w_angle__',
↪ 'e_interval__', 'phi_interval__', 'logP_interval__', 'logK_interval__']
message: Desired error not necessarily achieved due to precision loss.
logp: -1325.8433398976658 -> -835.1912653583948
```

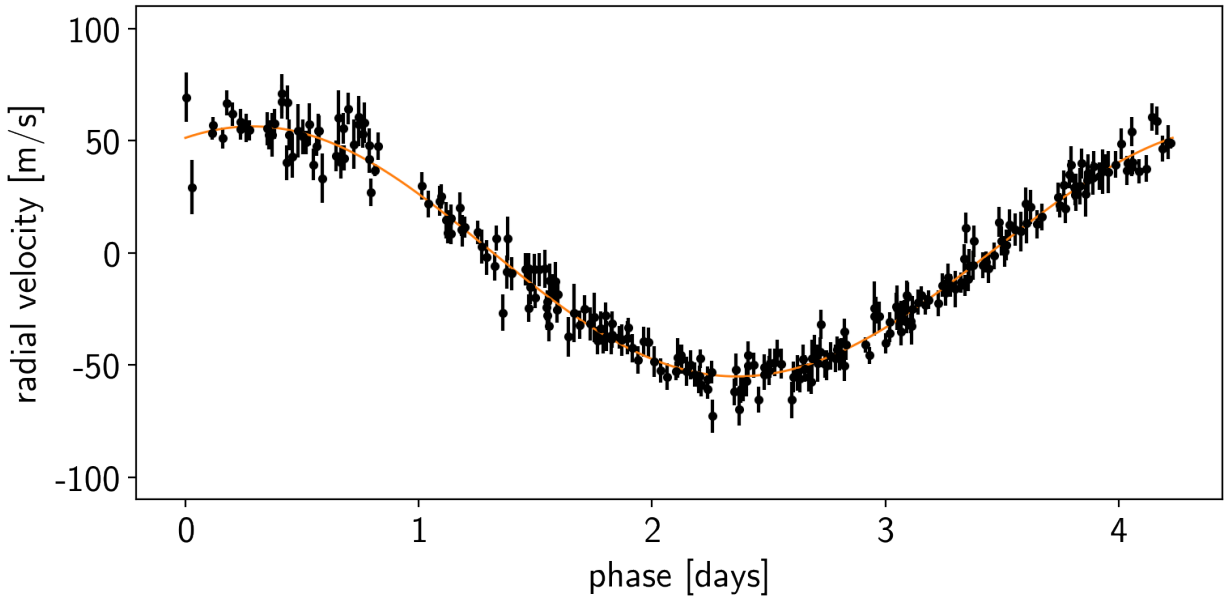
Let's make a plot to check that this initialization looks reasonable. In the top plot, we're looking at the RV observations

as a function of time with the initial guess for the long-term trend overplotted in blue. In the lower panel, we plot the "folded" curve where we have wrapped the observations onto the best-fit period and the prediction for a single overplotted in orange. If this doesn't look good, try adjusting the initial guesses for the parameters and see if you can get a better fit.

Exercise: Try changing the initial guesses for the parameters (as specified by the `testval` argument) and see how sensitive the results are to these values. Are there some parameters that are less important? Why is this?

```
fig, axes = plt.subplots(2, 1, figsize=(8, 8))

period = map_params["P"]

ax = axes[0]
ax.errorbar(t, rv, yerr=rv_err, fmt=".k")
ax.plot(t, map_params["bkg"], color="C0", lw=1)
ax.set_ylim(-110, 110)
ax.set_ylabel("radial velocity [m/s]")
ax.set_xlabel("time [days]")

ax = axes[1]
ax.errorbar(t % period, rv - map_params["bkg"], yerr=rv_err, fmt=".k")
ax.plot(phase * period, map_params["rvphase"], color="C1", lw=1)
ax.set_ylim(-110, 110)
ax.set_ylabel("radial velocity [m/s]")
ax.set_xlabel("phase [days]")

plt.tight_layout()
```





Now let's sample the posterior starting from our MAP estimate.

```
with model:
    trace = pm.sample(draws=2000, tune=1000, start=map_params, chains=2)
```

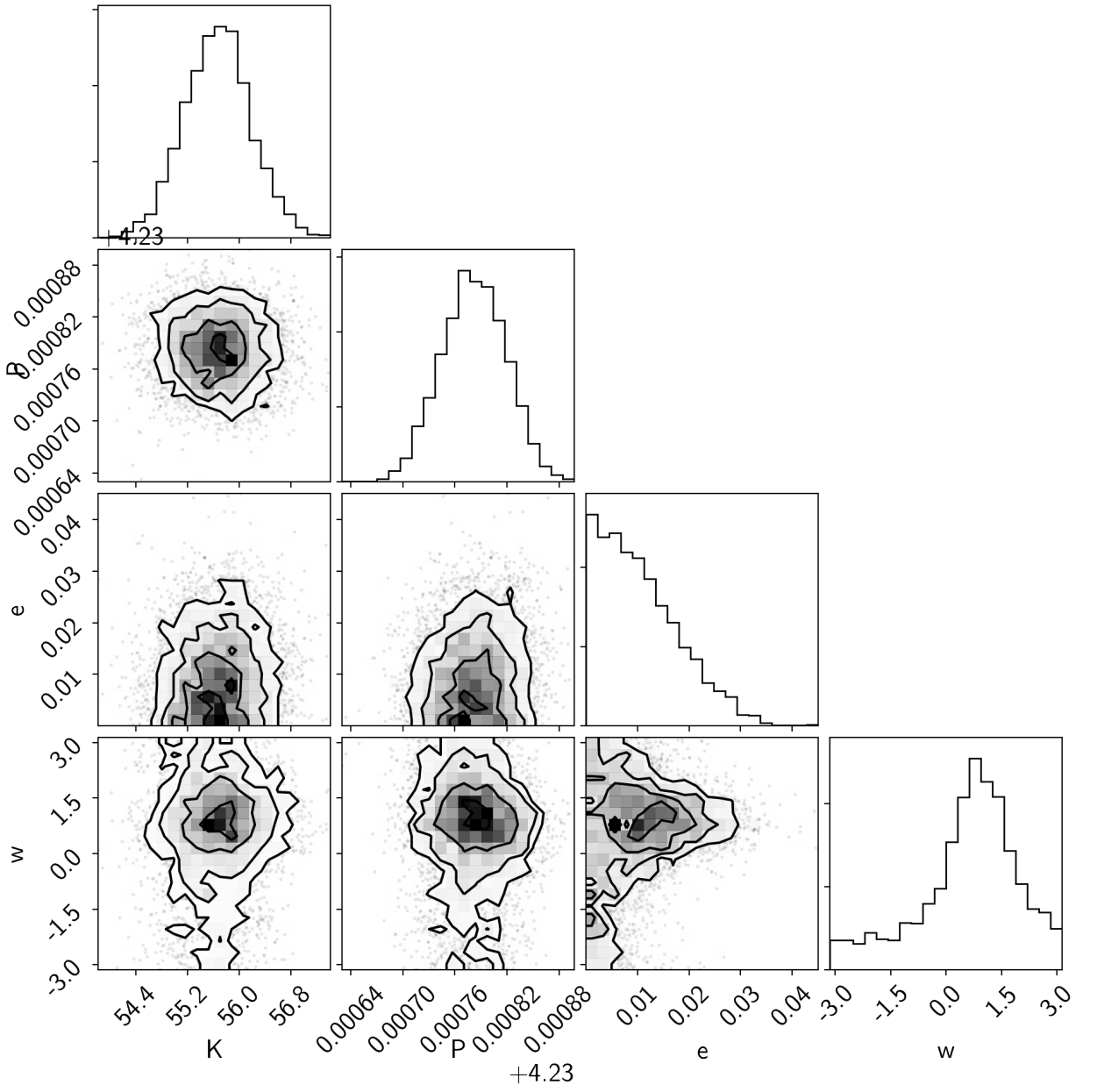
```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (2 chains in 4 jobs)
NUTS: [rvtrend, rv0, logjitter, w, e, phi, logP, logK]
Sampling 2 chains: 100%|| 6000/6000 [00:23<00:00, 251.77draws/s]
There were 37 divergences after tuning. Increase target_accept or␣
↪reparameterize.
There were 79 divergences after tuning. Increase target_accept or␣
↪reparameterize.
```

As above, it's always a good idea to take a look at the summary statistics for the chain. If everything went as planned, there should be more than 1000 effective samples per chain and the Rhat values should be close to 1. (Not too bad for less than 30 seconds of run time!)

```
pm.summary(trace, varnames=["logK", "logP", "phi", "e", "w", "logjitter", "rv0",
↪"rvtrend"])
```

Similarly, we can make the corner plot again for this model.

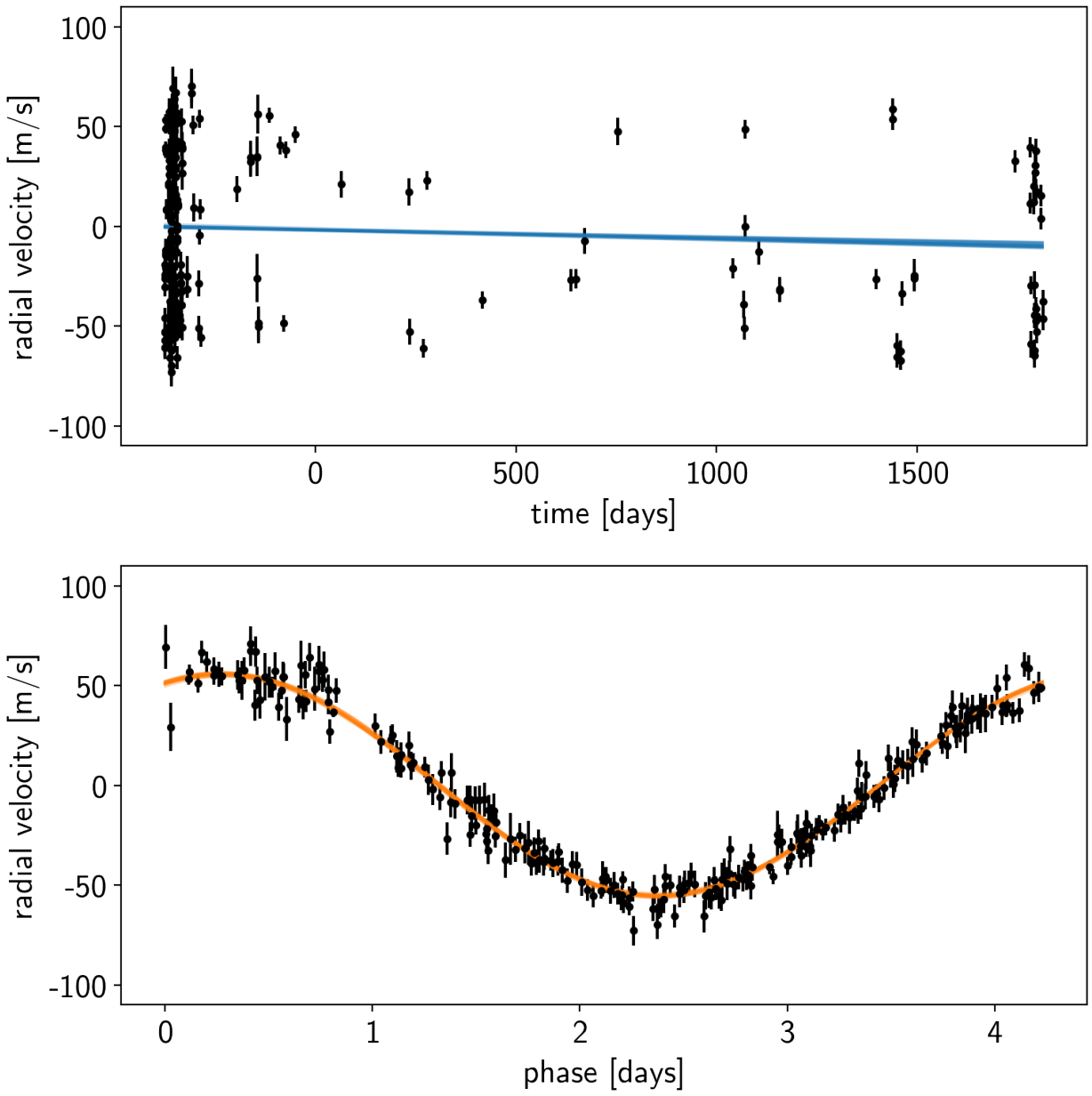
```
samples = pm.trace_to_dataframe(trace, varnames=["K", "P", "e", "w"])
corner.corner(samples);
```



Finally, the last plot that we'll make here is of the posterior predictive density. In this case, this means that we want to look at the distribution of predicted models that are consistent with the data. As above, the top plot shows the raw observations as black error bars and the RV trend model is overplotted in blue. But, this time, the blue line is actually

composed of 25 lines that are samples from the posterior over trends that are consistent with the data. In the bottom panel, the orange lines indicate the same 25 posterior samples for the RV curve of one orbit.

```
fig, axes = plt.subplots(2, 1, figsize=(8, 8))

period = map_params["P"]

ax = axes[0]
ax.errorbar(t, rv, yerr=rv_err, fmt=".k")
ax.set_ylabel("radial velocity [m/s]")
ax.set_xlabel("time [days]")

ax = axes[1]
ax.errorbar(t % period, rv - map_params["bkg"], yerr=rv_err, fmt=".k")
ax.set_ylabel("radial velocity [m/s]")
ax.set_xlabel("phase [days]")

for i in np.random.randint(len(trace) * trace.nchains, size=25):
    axes[0].plot(t, trace["bkg"][i], color="C0", lw=1, alpha=0.3)
    axes[1].plot(phase * period, trace["rvphase"][i], color="C1", lw=1, alpha=0.3)

axes[0].set_ylim(-110, 110)
axes[1].set_ylim(-110, 110)

plt.tight_layout()
```



Note: This tutorial was generated from an IPython notebook that can be downloaded here.

```
theano version: 1.0.4
pymc3 version: 3.6
exoplanet version: 0.1.5
```

2.2 PyMC3 extras

exoplanet comes bundled with a few utilities that can make it easier to use and debug PyMC3 models for fitting exoplanet data. This tutorial briefly describes these features and their use.

2.2.1 Custom tuning schedule

The main extra is the *exoplanet.PyMC3Sampler* class that wraps the PyMC3 sampling procedure to include support for learning off-diagonal elements of the mass matrix. This is *very* important for any problems where there are covariances between the parameters (this is true for pretty much all exoplanet models). A thorough discussion of this can be found elsewhere online, but here is a simple demo where we sample a covariant Gaussian using *exoplanet.PyMC3Sampler*.

First, we generate a random positive definite covariance matrix for the Gaussian:

```
import numpy as np

ndim = 5
np.random.seed(42)
L = np.random.randn(ndim, ndim)
L[np.diag_indices_from(L)] = 0.1*np.exp(L[np.diag_indices_from(L)])
L[np.triu_indices_from(L, 1)] = 0.0
cov = np.dot(L, L.T)
```

And then we can sample this using PyMC3 and *exoplanet.PyMC3Sampler*:

```
import pymc3 as pm
import exoplanet as xo

sampler = xo.PyMC3Sampler()

with pm.Model() as model:
    pm.MvNormal("x", mu=np.zeros(ndim), chol=L, shape=(ndim,))

    # Run the burn-in and learn the mass matrix
    step_kwargs = dict(target_accept=0.9)
    sampler.tune(tune=2000, step_kwargs=step_kwargs, chains=4)

    # Run the production chain
    trace = sampler.sample(draws=2000, chains=4)
```

```
Sampling 4 chains: 100%|| 308/308 [00:02<00:00, 131.26draws/s]
Sampling 4 chains: 100%|| 108/108 [00:01<00:00, 91.15draws/s]
Sampling 4 chains: 100%|| 208/208 [00:00<00:00, 845.99draws/s]
Sampling 4 chains: 100%|| 408/408 [00:00<00:00, 1873.31draws/s]
Sampling 4 chains: 100%|| 808/808 [00:00<00:00, 2470.84draws/s]
Sampling 4 chains: 100%|| 1608/1608 [00:00<00:00, 2621.14draws/s]
Sampling 4 chains: 100%|| 4608/4608 [00:01<00:00, 2531.78draws/s]
Sampling 4 chains: 100%|| 208/208 [00:00<00:00, 1681.06draws/s]
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [x]
Sampling 4 chains: 100%|| 8000/8000 [00:02<00:00, 2989.70draws/s]
```

This is a little more verbose than the standard use of PyMC3, but the performance is several orders of magnitude better than you would get without the mass matrix tuning. As you can see from the `pymc3.summary`, the autocorrelation time of this chain is about 1 as we would expect for a simple problem like this.

```
pm.summary(trace)
```

2.2.2 Evaluating model components for specific samples

I find that when I'm debugging a PyMC3 model, I often want to inspect the value of some part of the model for a given set of parameters. As far as I can tell, there isn't a simple way to do this in PyMC3, so *exoplanet* comes with a hack for doing this: *exoplanet.eval_in_model()*. This function handles the mapping between named PyMC3 variables and the input required by the Theano function that can evaluate the requested variable or tensor.

As a demo, let's say that we're fitting a parabola to some data:

```
np.random.seed(42)
x = np.sort(np.random.uniform(-1, 1, 50))
with pm.Model() as model:
    logs = pm.Normal("logs", mu=-3.0, sd=1.0)
    a0 = pm.Normal("a0")
    a1 = pm.Normal("a1")
    a2 = pm.Normal("a2")
    mod = a0 + a1 * x + a2 * x**2

    # Sample from the prior
    prior_sample = pm.sample_prior_predictive(samples=1)
    y = xo.eval_in_model(mod, prior_sample)
    y += np.exp(prior_sample["logs"]) * np.random.randn(len(y))

    # Add the likelihood
    pm.Normal("obs", mu=mod, sd=pm.math.exp(logs), observed=y)

    # Fit the data
    map_soln = xo.optimize()
    trace = pm.sample()
```

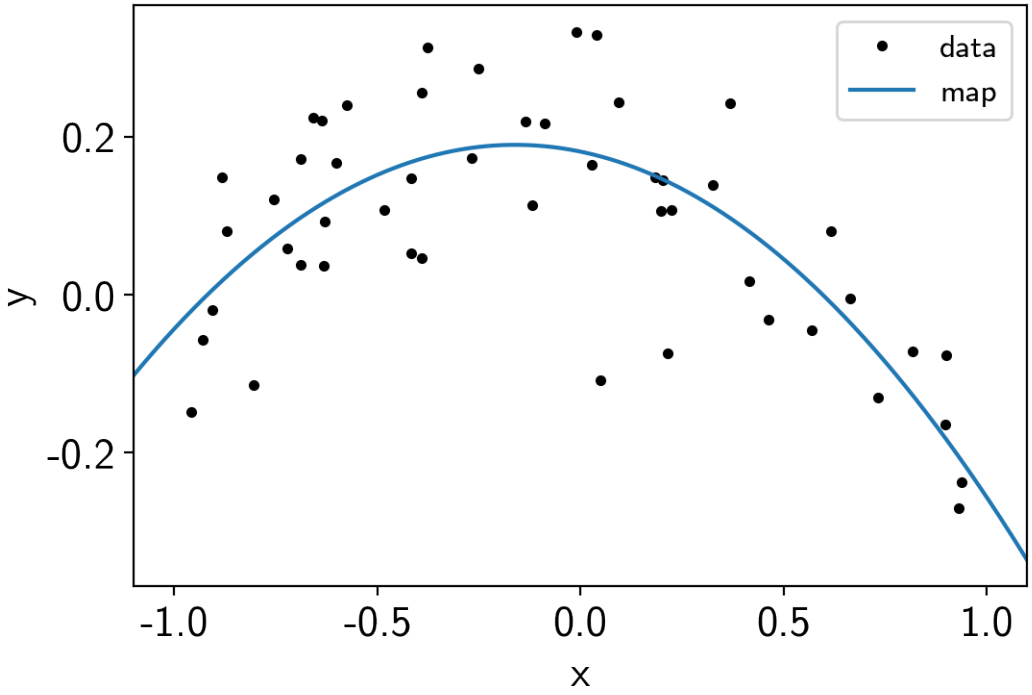
```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
```

```
optimizing logp for variables: ['a2', 'a1', 'a0', 'logs']
message: Optimization terminated successfully.
logp: -180.51036900636572 -> 42.72311721910288
```

```
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [a2, a1, a0, logs]
Sampling 4 chains: 100%|| 4000/4000 [00:01<00:00, 3244.40draws/s]
The acceptance probability does not match the target. It is 0.8788002200800235, but␣
↪should be close to 0.8. Try to increase the number of tuning steps.
The acceptance probability does not match the target. It is 0.8834353310005734, but␣
↪should be close to 0.8. Try to increase the number of tuning steps.
The acceptance probability does not match the target. It is 0.8827839735098203, but␣
↪should be close to 0.8. Try to increase the number of tuning steps.
```

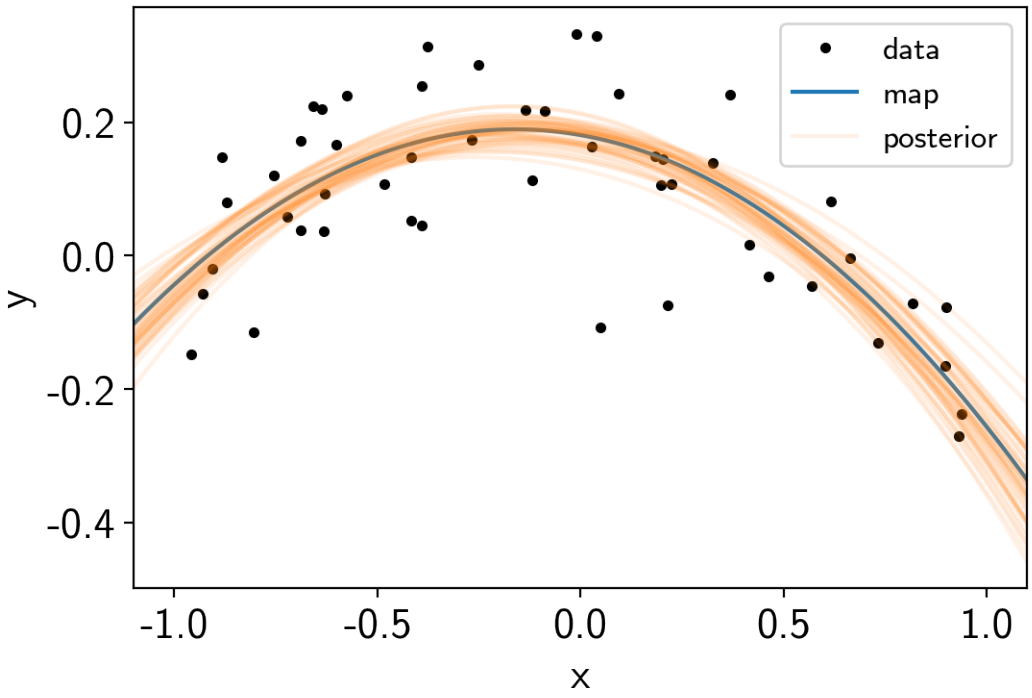
After running the fit, it might be interesting to look at the predictions of the model. We could have added a `pymc3.Deterministic` node for eveything, but that can end up taking up a lot of memory and sometimes its useful to be able to experiement with different outputs. Using `exoplanet.utils.eval_in_model()` we can, for example, evaluate the maximum a posteriori (MAP) model prediction on a fine grid:

```
import matplotlib.pyplot as plt

x_grid = np.linspace(-1.1, 1.1, 5000)
with model:
    pred = xo.eval_in_model(a0 + a1 * x_grid + a2 * x_grid**2, map_soln)

plt.plot(x, y, ".k", label="data")
plt.plot(x_grid, pred, label="map")
plt.legend(fontsize=12)
plt.xlabel("x")
plt.ylabel("y")
plt.xlim(-1.1, 1.1);
```



We can also combine this with *exoplanet.get_samples_from_trace()* to plot this prediction for a set of samples in the trace.

```
samples = np.empty((50, len(x_grid)))
with model:
    y_grid = a0 + a1 * x_grid + a2 * x_grid**2
    for i, sample in enumerate(xo.get_samples_from_trace(trace, size=50)):
        samples[i] = xo.eval_in_model(y_grid, sample)

plt.plot(x, y, ".k", label="data")
plt.plot(x_grid, pred, label="map")
plt.plot(x_grid, samples[0], color="C1", alpha=0.1, label="posterior")
plt.plot(x_grid, samples[1:].T, color="C1", alpha=0.1)
plt.legend(fontsize=12)
plt.xlabel("x")
plt.ylabel("y")
plt.xlim(-1.1, 1.1);
```



Note: This tutorial was generated from an IPython notebook that can be downloaded here.

```
theano version: 1.0.4
pymc3 version: 3.6
exoplanet version: 0.1.5
```

2.3 Radial velocity fitting

In this tutorial, we will demonstrate how to fit radial velocity observations of an exoplanetary system using *exoplanet*. We will follow the getting started tutorial from the exellent RadVel package where they fit for the parameters of the two planets in the K2-24 system.

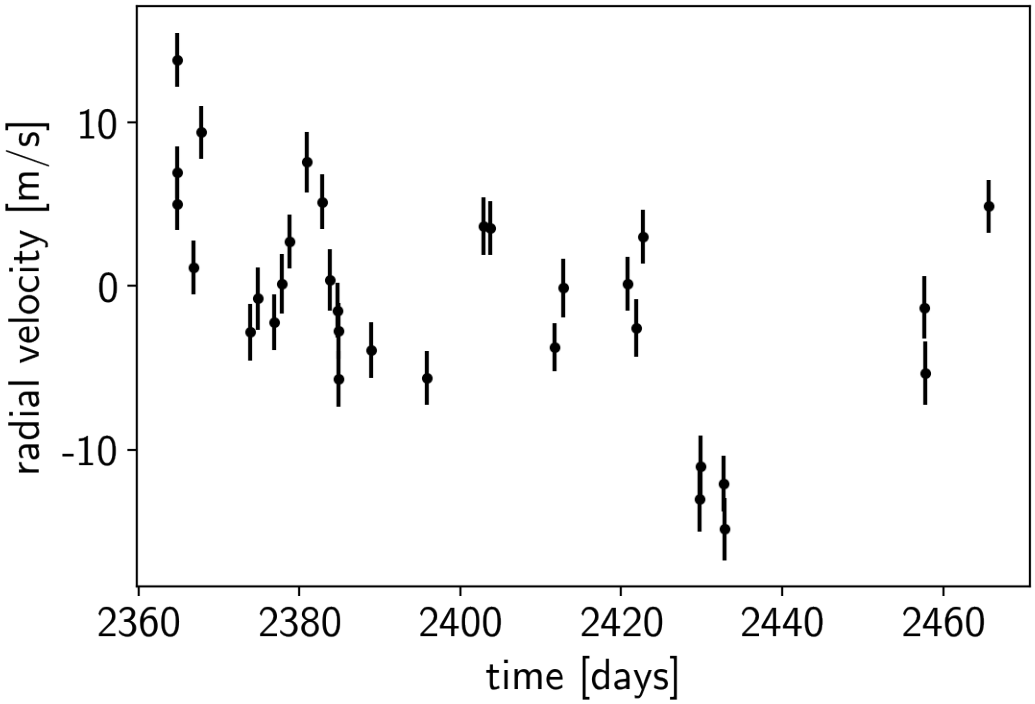
First, let's download the data from RadVel:

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

url = "https://raw.githubusercontent.com/California-Planet-Search/radvel/master/
↪example_data/epic203771098.csv"
data = pd.read_csv(url, index_col=0)

x = np.array(data.t)
y = np.array(data.vel)
yerr = np.array(data.errvel)

plt.errorbar(x, y, yerr=yerr, fmt=".k")
```

(continues on next page)

(continued from previous page)

```
plt.xlabel("time [days]")
plt.ylabel("radial velocity [m/s]");
```



Now, we know the periods and transit times for the planets from the K2 light curve, so let's start by using the *exoplanet.estimate_semi_amplitude()* function to estimate the expected RV semi-amplitudes for the planets.

```
import exoplanet as xo

periods = [20.8851, 42.3633]
period_errs = [0.0003, 0.0006]
t0s = [2072.7948, 2082.6251]
t0_errs = [0.0007, 0.0004]
Ks = xo.estimate_semi_amplitude(periods, x, y, yerr, t0s=t0s)
print(Ks, "m/s")
```

```
[5.05069163 5.50983542] m/s
```

2.3.1 The radial velocity model in PyMC3

Now that we have the data and an estimate of the initial values for the parameters, let's start defining the probabilistic model in PyMC3 (take a look at intro-to-pymc3 if you're new to PyMC3). First, we'll define our priors on the parameters:

```
import pymc3 as pm
import theano.tensor as tt

with pm.Model() as model:
```

(continues on next page)

(continued from previous page)

```
    # Gaussian priors based on transit data (from Petigura et al.)
    t0 = pm.Normal("t0", mu=np.array(t0s), sd=np.array(t0_errs), shape=2)
    P = pm.Normal("P", mu=np.array(periods), sd=np.array(period_errs), shape=2)

    # Wide log-normal prior for semi-amplitude
    logK = pm.Normal("logK", mu=np.log(Ks), sd=10.0, shape=2)

    # This is a sanity check that restricts the semiamplitude to reasonable
    # values because things can get ugly as K -> 0
    pm.Potential("logK_bound", tt.switch(logK < 0, -np.inf, 0.0))

    # We also want to keep period physical but this probably won't be hit
    pm.Potential("P_bound", tt.switch(P <= 0, -np.inf, 0.0))

    # Eccentricity & argument of periasteron
    ecc = pm.Uniform("ecc", lower=0, upper=0.99, shape=2,
                     testval=np.array([0.1, 0.1]))
    omega = xo.distributions.Angle("omega", shape=2, testval=np.zeros(2))

    # Jitter & a quadratic RV trend
    logs = pm.Normal("logs", mu=np.log(np.median(yerr)), sd=5.0)
    trend = pm.Normal("trend", mu=0, sd=10.0**-np.arange(3)[::-1], shape=3)
```

Now we'll define the orbit model:

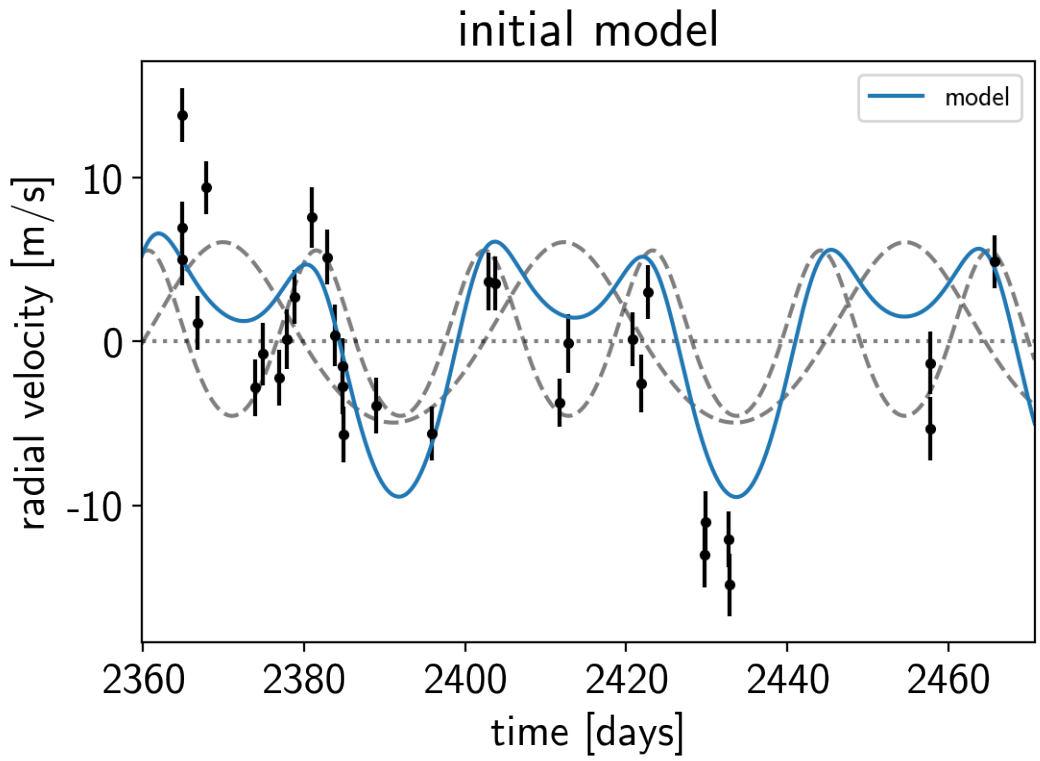
```
with model:

    # Set up the orbit
    orbit = xo.orbits.KeplerianOrbit(
        period=P, t0=t0,
        ecc=ecc, omega=omega)

    # Set up the RV model and save it as a deterministic
    # for plotting purposes later
    vrad = orbit.get_radial_velocity(x, K=tt.exp(logK))
    pm.Deterministic("vrad", vrad)

    # Define the background model
    A = np.vander(x - 0.5*(x.min() + x.max()), 3)
    bkg = pm.Deterministic("bkg", tt.dot(A, trend))

    # Sum over planets and add the background to get the full model
    rv_model = pm.Deterministic("rv_model", tt.sum(vrad, axis=-1) + bkg)
```

For plotting purposes, it can be useful to also save the model on a fine grid in time.

```
t = np.linspace(x.min()-5, x.max()+5, 1000)

with model:
    vrad_pred = orbit.get_radial_velocity(t, K=tt.exp(logK))
    pm.Deterministic("vrad_pred", vrad_pred)
    A_pred = np.vander(t - 0.5*(x.min() + x.max()), 3)
    bkg_pred = pm.Deterministic("bkg_pred", tt.dot(A_pred, trend))
    rv_model_pred = pm.Deterministic("rv_model_pred",
                                     tt.sum(vrad_pred, axis=-1) + bkg_pred)
```

Now, we can plot the initial model:

```
plt.errorbar(x, y, yerr=yerr, fmt=".k")

with model:
    plt.plot(t, xo.eval_in_model(vrad_pred), "--k", alpha=0.5)
    plt.plot(t, xo.eval_in_model(bkg_pred), ":k", alpha=0.5)
    plt.plot(t, xo.eval_in_model(rv_model_pred), label="model")

plt.legend(fontsize=10)
plt.xlim(t.min(), t.max())
plt.xlabel("time [days]")
plt.ylabel("radial velocity [m/s]")
plt.title("initial model");
```



In this plot, the background is the dotted line, the individual planets are the dashed lines, and the full model is the blue line.

It doesn't look amazing so let's add in the likelihood and fit for the maximum a posterior parameters.

```
with model:

    err = tt.sqrt(yerr**2 + tt.exp(2*logs))
    pm.Normal("obs", mu=rv_model, sd=err, observed=y)

    map_soln = xo.optimize(start=model.test_point, vars=[trend])
    map_soln = xo.optimize(start=map_soln)
```

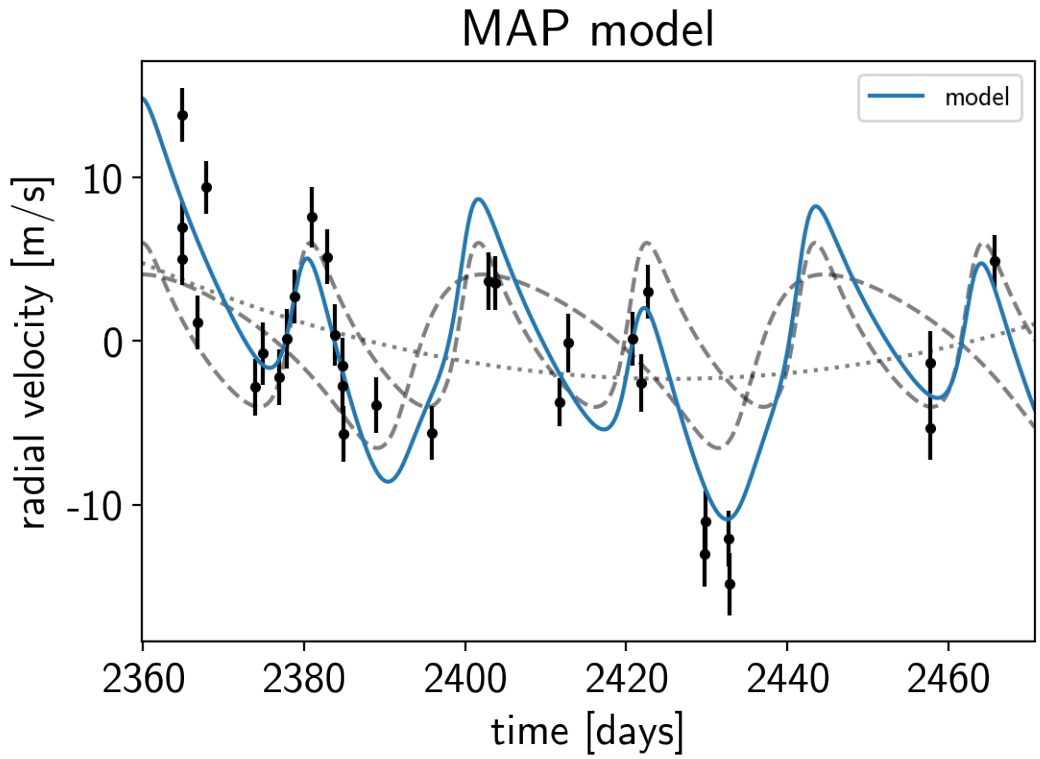
```
optimizing logp for variables: ['trend']
message: Optimization terminated successfully.
```

(continues on next page)

(continued from previous page)

```
logp: -87.51672438985798 -> -72.63226415698486
optimizing logp for variables: ['trend', 'logs', 'omega_angle__', 'ecc_interval__',
↪'logK', 'P', 't0']
message: Desired error not necessarily achieved due to precision loss.
logp: -72.63226415698486 -> -22.056179633452878
```

```
plt.errorbar(x, y, yerr=yerr, fmt=".k")
plt.plot(t, map_soln["vrad_pred"], "--k", alpha=0.5)
plt.plot(t, map_soln["bkg_pred"], ":k", alpha=0.5)
plt.plot(t, map_soln["rv_model_pred"], label="model")

plt.legend(fontsize=10)
plt.xlim(t.min(), t.max())
plt.xlabel("time [days]")
plt.ylabel("radial velocity [m/s]")
plt.title("MAP model");
```



That looks better.

2.3.2 Sampling

Now that we have our model set up and a good estimate of the initial parameters, let's start sampling. There are substantial covariances between some of the parameters so we'll use a *exoplanet.PyMC3Sampler* to tune the sampler (see the pymc3-extras tutorial for more information).

```
np.random.seed(42)
sampler = xo.PyMC3Sampler(start=200, window=100, finish=300)
```

(continues on next page)

(continued from previous page)

```
with model:
    burnin = sampler.tune(tune=3000, start=model.test_point,
                          step_kwargs=dict(target_accept=0.9))
    trace = sampler.sample(draws=3000)
```

```
Sampling 4 chains: 100%|| 808/808 [00:10<00:00, 74.18draws/s]
Sampling 4 chains: 100%|| 408/408 [00:05<00:00, 80.47draws/s]
Sampling 4 chains: 100%|| 808/808 [00:04<00:00, 167.87draws/s]
Sampling 4 chains: 100%|| 1608/1608 [00:03<00:00, 413.90draws/s]
Sampling 4 chains: 100%|| 8408/8408 [00:15<00:00, 534.00draws/s]
Sampling 4 chains: 100%|| 1208/1208 [00:02<00:00, 443.56draws/s]
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [trend, logs, omega, ecc, logK, P, t0]
Sampling 4 chains: 100%|| 12000/12000 [00:18<00:00, 645.68draws/s]
There were 108 divergences after tuning. Increase target_accept or␣
↪reparameterize.
The acceptance probability does not match the target. It is 0.788312912990659,
↪ but should be close to 0.9. Try to increase the number of tuning steps.
There were 10 divergences after tuning. Increase target_accept or␣
↪reparameterize.
The acceptance probability does not match the target. It is 0.
↪8402226010187613, but should be close to 0.9. Try to increase the number␣
↪of tuning steps.
There were 27 divergences after tuning. Increase target_accept or␣
↪reparameterize.
The acceptance probability does not match the target. It is 0.
↪8258238424536315, but should be close to 0.9. Try to increase the number␣
↪of tuning steps.
There were 28 divergences after tuning. Increase target_accept or␣
↪reparameterize.
The acceptance probability does not match the target. It is 0.
↪8206821919006299, but should be close to 0.9. Try to increase the number␣
↪of tuning steps.
The number of effective samples is smaller than 10% for some parameters.
```

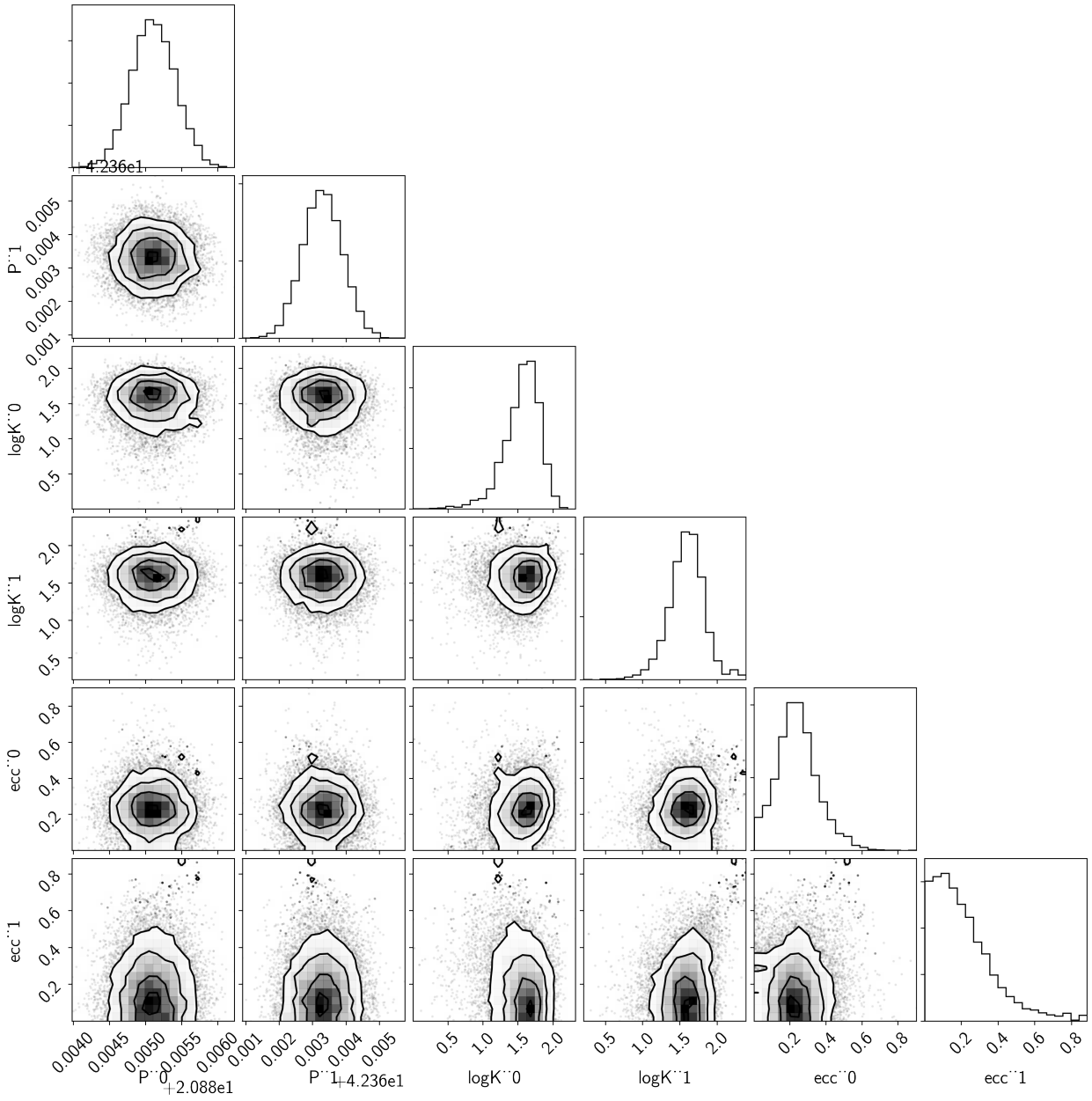
After sampling, it's always a good idea to do some convergence checks. First, let's check the number of effective samples and the Gelman-Rubin statistic for our parameters of interest:

```
pm.summary(trace, varnames=["trend", "logs", "omega", "ecc", "t0", "logK", "P"])
```

It looks like everything is pretty much converged here. Not bad for 14 parameters and about a minute of runtime...

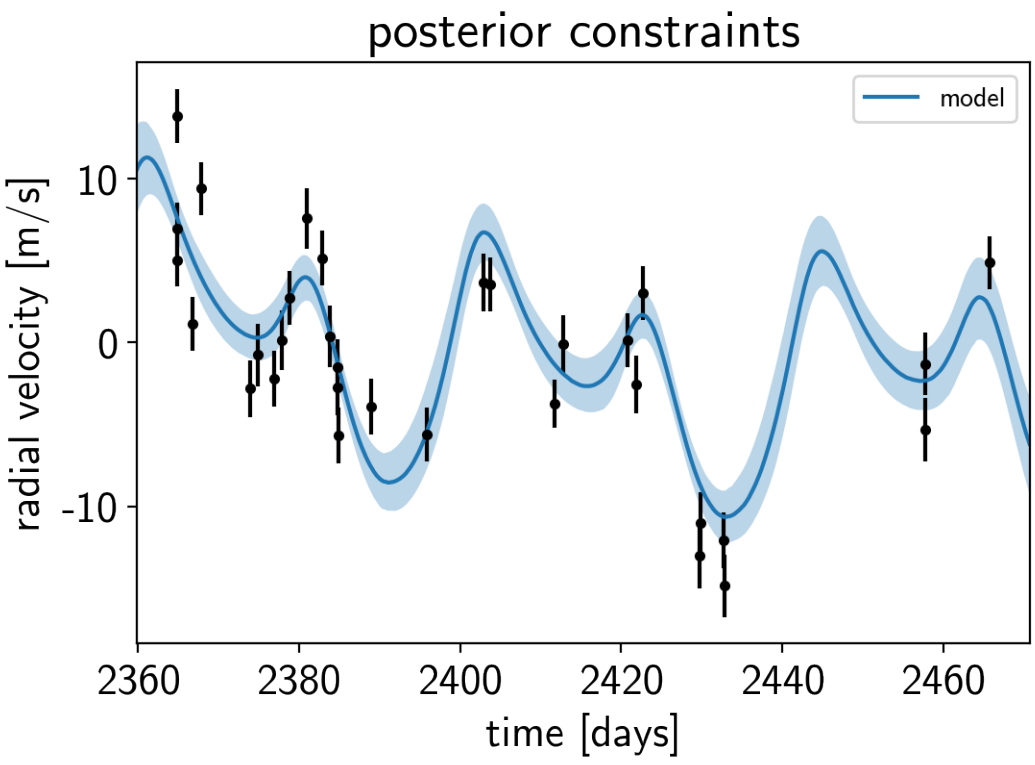
Then we can make a corner plot of any combination of the parameters. For example, let's look at period, semi-amplitude, and eccentricity:

```
import corner
samples = pm.trace_to_dataframe(trace, varnames=["P", "logK", "ecc"])
corner.corner(samples);
```



Finally, let's plot the plosterior constraints on the RV model and compare those to the data:

```
plt.errorbar(x, y, yerr=yerr, fmt=".k")

# Compute the posterior predictions for the RV model
pred = np.percentile(trace["rv_model_pred"], [16, 50, 84], axis=0)
plt.plot(t, pred[1], color="C0", label="model")
art = plt.fill_between(t, pred[0], pred[2], color="C0", alpha=0.3)
art.set_edgecolor("none")

plt.legend(fontsize=10)
plt.xlim(t.min(), t.max())
plt.xlabel("time [days]")
plt.ylabel("radial velocity [m/s]")
plt.title("posterior constraints");
```



2.3.3 Phase plots

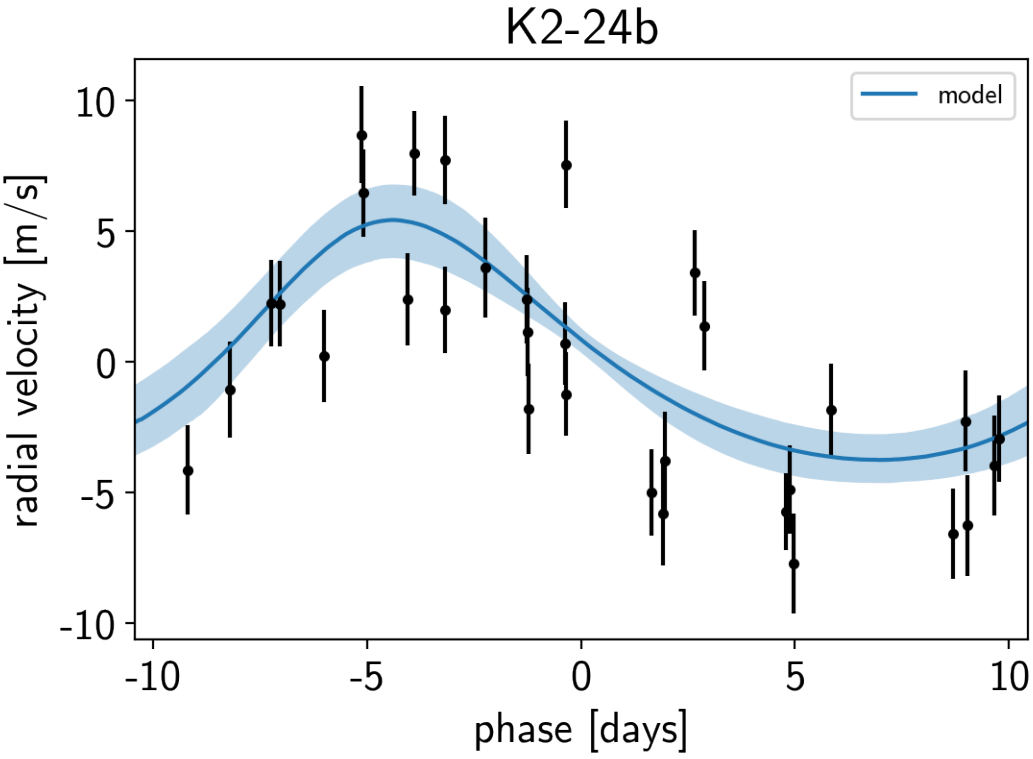
It might be also be interesting to look at the phased plots for this system. Here we'll fold the dataset on the median of posterior period and then overplot the posterior constraint on the folded model orbits.

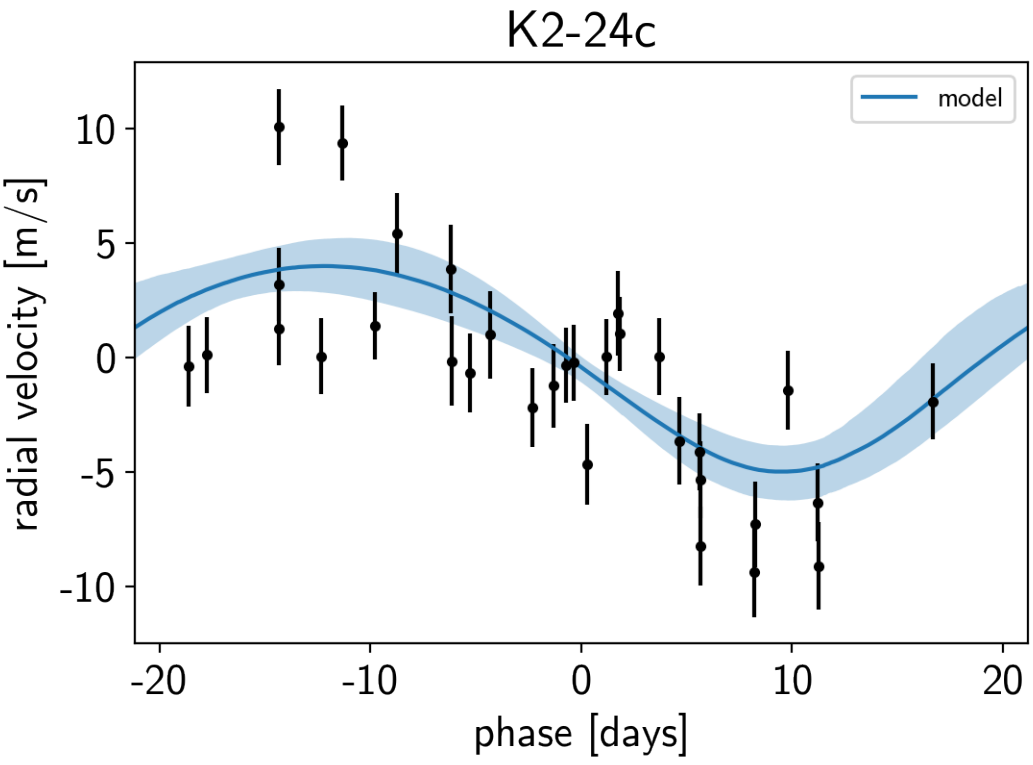
```
for n, letter in enumerate("bc"):
    plt.figure()

    # Get the posterior median orbital parameters
    p = np.median(trace["P"][:, n])
    t0 = np.median(trace["t0"][:, n])

    # Compute the median of posterior estimate of the background RV
    # and the contribution from the other planet. Then we can remove
    # this from the data to plot just the planet we care about.
    other = np.median(trace["vrad"][:, :, (n + 1) % 2], axis=0)
    other += np.median(trace["bkg"], axis=0)

    # Plot the folded data
    x_fold = (x - t0 + 0.5*p) % p - 0.5*p
    plt.errorbar(x_fold, y - other, yerr=yerr, fmt=".k")

    # Compute the posterior prediction for the folded RV model for this
    # planet
    t_fold = (t - t0 + 0.5*p) % p - 0.5*p
    inds = np.argsort(t_fold)
    pred = np.percentile(trace["vrad_pred"][:, inds, n], [16, 50, 84], axis=0)
    plt.plot(t_fold[inds], pred[1], color="C0", label="model")
    art = plt.fill_between(t_fold[inds], pred[0], pred[2], color="C0", alpha=0.3)
```

(continues on next page)

(continued from previous page)

```
    art.set_edgecolor("none")

    plt.legend(fontsize=10)
    plt.xlim(-0.5*p, 0.5*p)
    plt.xlabel("phase [days]")
    plt.ylabel("radial velocity [m/s]")
    plt.title("K2-24{0}".format(letter));
```





Note: This tutorial was generated from an IPython notebook that can be downloaded here.

```
theano version: 1.0.4
pymc3 version: 3.6
exoplanet version: 0.1.5
```

2.4 Transit fitting

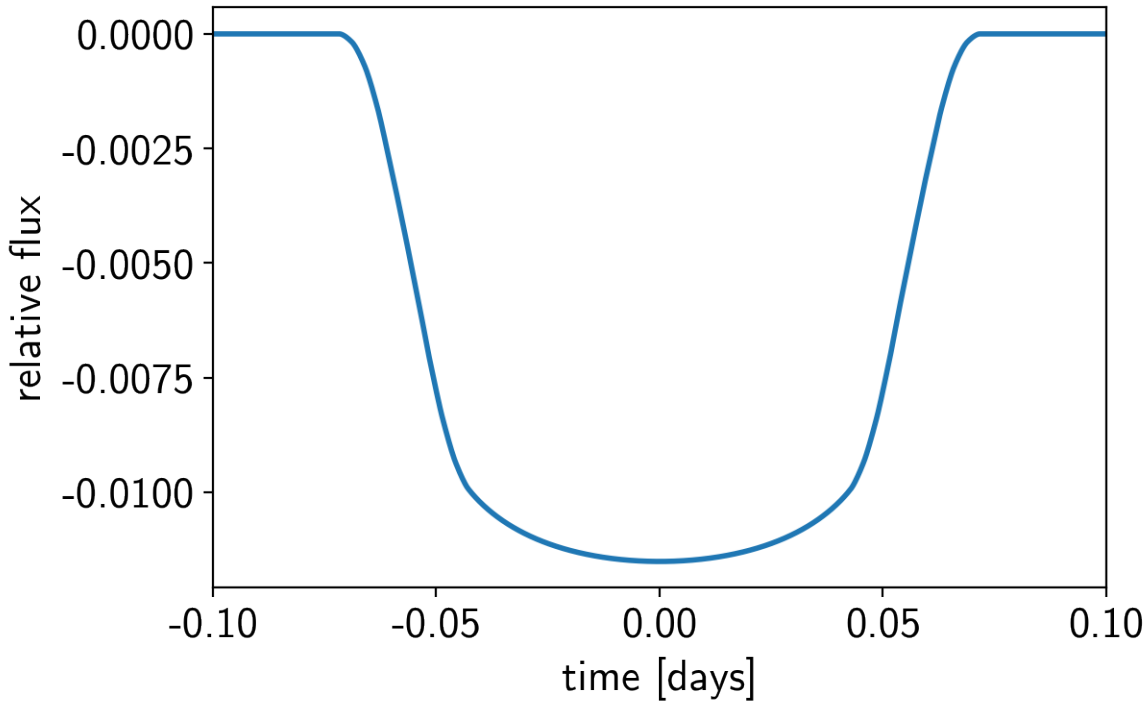
exoplanet includes methods for computing the light curves transiting planets. In its simplest form this can be used to evaluate a light curve like you would do with batman, for example:

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

import exoplanet as xo

# The light curve calculation requires an orbit
orbit = xo.orbits.KeplerianOrbit(period=3.456)

# Compute a limb-darkened light curve using starry
t = np.linspace(-0.1, 0.1, 1000)
u = [0.3, 0.2]
light_curve = xo.StarryLightCurve(u).get_light_curve(
    orbit=orbit, r=0.1, t=t, texp=0.02).eval()
# Note: the `eval` is needed because this is using Theano in
# the background
```

(continues on next page)

(continued from previous page)

```
plt.plot(t, light_curve, color="C0", lw=2)
plt.ylabel("relative flux")
plt.xlabel("time [days]")
plt.xlim(t.min(), t.max());
```



But the real power comes from the fact that this is defined as a Theano operation so it can be combined with PyMC3 to do transit inference using Hamiltonian Monte Carlo.

2.4.1 The transit model in PyMC3

In this section, we will construct a simple transit fit model using *PyMC3* and then we will fit a two planet model to simulated data. To start, let's randomly sample some periods and phases and then define the time sampling:

```
np.random.seed(123)
periods = np.random.uniform(5, 20, 2)
t0s = periods * np.random.rand(2)
t = np.arange(0, 80, 0.02)
yerr = 5e-4
```

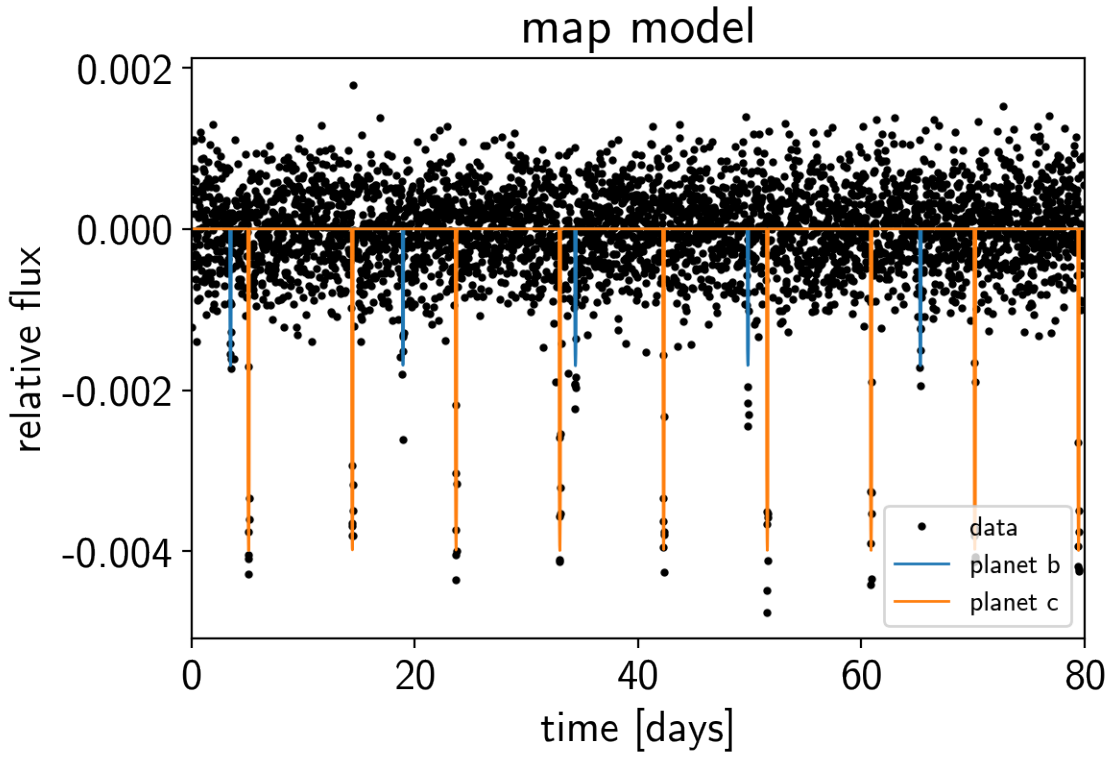
Then, define the parameters. In this simple model, we'll just fit for the limb darkening parameters of the star, and the period, phase, impact parameter, and radius ratio of the planets (note: this is already 10 parameters and running MCMC to convergence using emcee would probably take at least an hour). For the limb darkening, we'll use a quadratic law as parameterized by Kipping (2013) and for the joint radius ratio and impact parameter distribution we'll use the parameterization from Espinoza (2018). Both of these reparameterizations are implemented in *exoplanet* as custom *PyMC3* distributions (`exoplanet.distributions.QuadLimbDark` and `exoplanet.distributions.RadiusImpact` respectively).

```
import pymc3 as pm

with pm.Model() as model:

    # The baseline flux
    mean = pm.Normal("mean", mu=0.0, sd=1.0)

    # The time of a reference transit for each planet
    t0 = pm.Normal("t0", mu=t0s, sd=1.0, shape=2)

    # The log period; also tracking the period itself
    logP = pm.Normal("logP", mu=np.log(periods), sd=0.1, shape=2)
    period = pm.Deterministic("period", pm.math.exp(logP))

    # The Kipping (2013) parameterization for quadratic limb darkening paramters
    u = xo.distributions.QuadLimbDark("u", testval=np.array([0.3, 0.2]))

    # The Espinoza (2018) parameterization for the joint radius ratio and
    # impact parameter distribution
    r, b = xo.distributions.get_joint_radius_impact(
        min_radius=0.01, max_radius=0.1,
        testval_r=np.array([0.04, 0.06]),
        testval_b=np.random.rand(2)
    )

    # This shouldn't make a huge difference, but I like to put a uniform
    # prior on the *log* of the radius ratio instead of the value. This
    # can be implemented by adding a custom "potential" (log probability).
    pm.Potential("r_prior", -pm.math.log(r))

    # Set up a Keplerian orbit for the planets
    orbit = xo.orbits.KeplerianOrbit(period=period, t0=t0, b=b)

    # Compute the model light curve using starry
    light_curves = xo.StarryLightCurve(u).get_light_curve(
        orbit=orbit, r=r, t=t)
    light_curve = pm.math.sum(light_curves, axis=-1) + mean

    # Here we track the value of the model light curve for plotting
    # purposes
    pm.Deterministic("light_curves", light_curves)

    # In this line, we simulate the dataset that we will fit
    y = xo.eval_in_model(light_curve)
    y += yerr * np.random.randn(len(y))

    # The likelihood function assuming known Gaussian uncertainty
    pm.Normal("obs", mu=light_curve, sd=yerr, observed=y)

    # Fit for the maximum a posteriori parameters given the simuated
    # dataset
    map_soln = xo.optimize(start=model.test_point)
```

```
optimizing logp for variables: ['rb_radiusimpact__', 'u_quadlimbdark__', 'logP', 't0',
↪ 'mean']
message: Desired error not necessarily achieved due to precision loss.
logp: 24793.972586707856 -> 24799.526978939903
```

Now we can plot the simulated data and the maximum a posteriori model to make sure that our initialization looks ok.

```
plt.plot(t, y, ".k", ms=4, label="data")
for i, l in enumerate("bc"):
    plt.plot(t, map_soln["light_curves"][:, i], lw=1,
             label="planet {0}".format(l))
plt.xlim(t.min(), t.max())
plt.ylabel("relative flux")
plt.xlabel("time [days]")
plt.legend(fontsize=10)
plt.title("map model");
```



2.4.2 Sampling

Now, let's sample from the posterior defined by this model. As usual, there are strong covariances between some of the parameters so we'll use the *exoplanet.PyMC3Sampler* to sample.

```
np.random.seed(42)
sampler = xo.PyMC3Sampler(window=100, finish=200)
with model:
    burnin = sampler.tune(tune=2500, start=map_soln, step_kwargs=dict(target_accept=0.
↪9))
    trace = sampler.sample(draws=3000)
```

```
Sampling 4 chains: 100%|| 308/308 [00:24<00:00,  3.83draws/s]
Sampling 4 chains: 100%|| 408/408 [00:30<00:00,  1.84draws/s]
Sampling 4 chains: 100%|| 808/808 [00:49<00:00,  4.26draws/s]
Sampling 4 chains: 100%|| 1608/1608 [00:08<00:00, 188.10draws/s]
```

(continues on next page)

(continued from previous page)

```
Sampling 4 chains: 100%|| 6908/6908 [00:35<00:00, 192.18draws/s]
Sampling 4 chains: 100%|| 808/808 [00:04<00:00, 166.64draws/s]
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [rb, u, logP, t0, mean]
Sampling 4 chains: 100%|| 12000/12000 [00:54<00:00, 220.23draws/s]
```

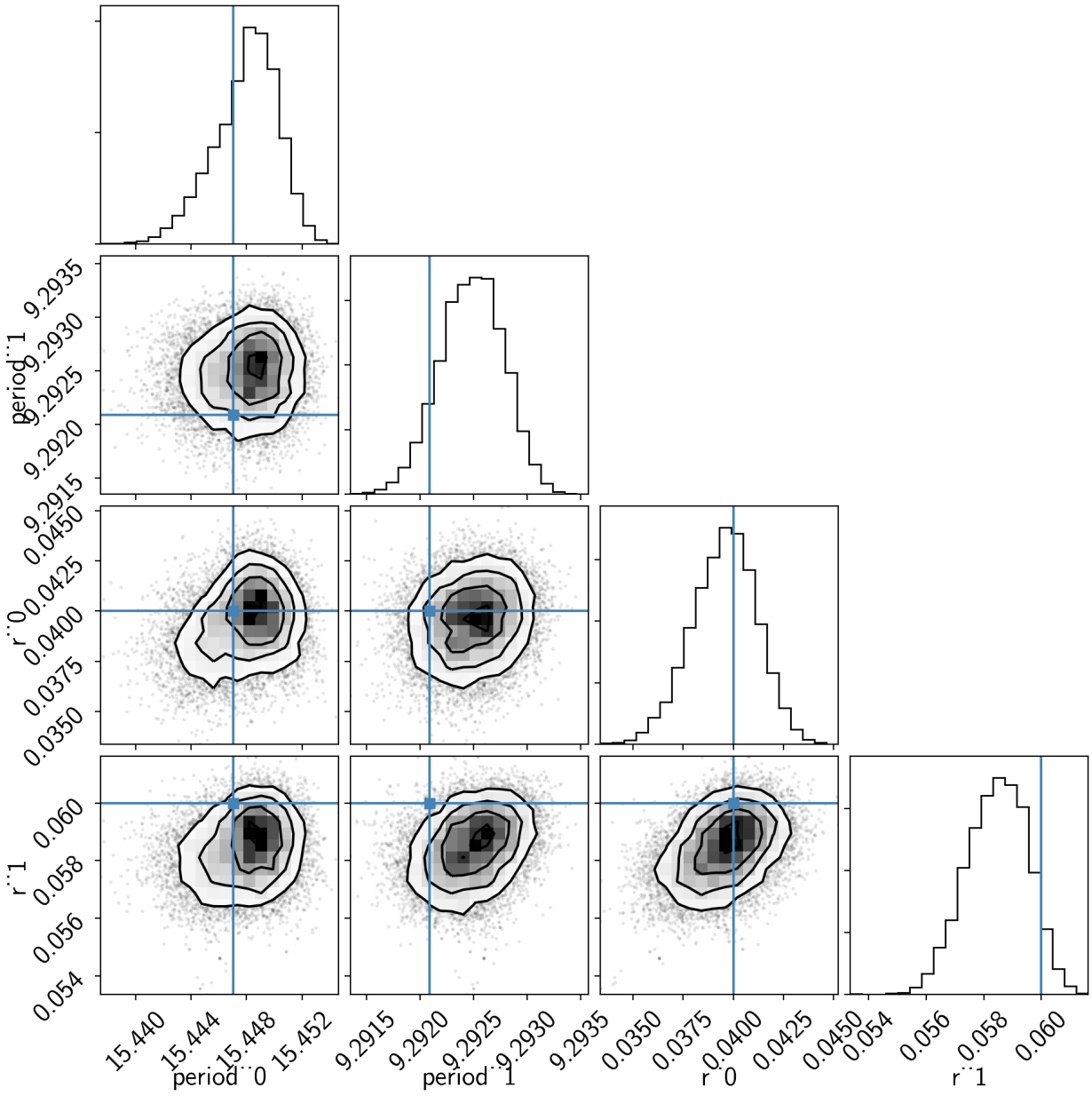
After sampling, it's important that we assess convergence. We can do that using the `pymc3.summary` function:

```
pm.summary(trace, varnames=["period", "t0", "r", "b", "u", "mean"])
```

That looks pretty good! Fitting this without *exoplanet* would have taken a lot more patience.

Now we can also look at the corner plot of some of that parameters of interest:

```
import corner
samples = pm.trace_to_dataframe(trace, varnames=["period", "r"])
truth = np.concatenate(xo.eval_in_model([period, r], model.test_point, model=model))
corner.corner(samples, truths=truth);
```



2.4.3 Phase plots

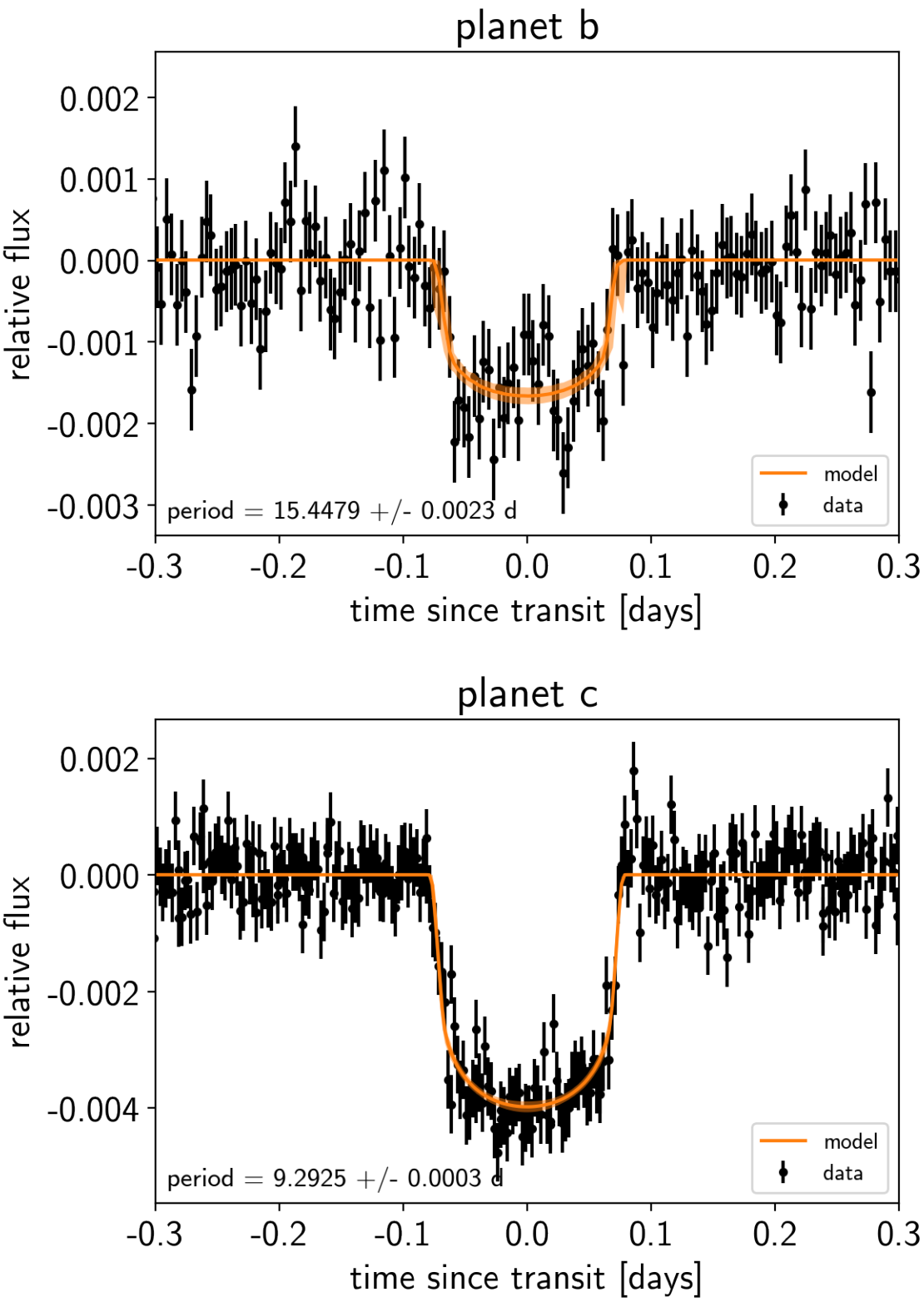
Like in the radial velocity tutorial (rv), we can make plots of the model predictions for each planet.

```
for n, letter in enumerate("bc"):
    plt.figure()

    # Get the posterior median orbital parameters
    p = np.median(trace["period"][:, n])
    t0 = np.median(trace["t0"][:, n])

    # Compute the median of posterior estimate of the contribution from
    # the other planet. Then we can remove this from the data to plot
```

(continues on next page)

(continued from previous page)

```
    # just the planet we care about.
    other = np.median(trace["light_curves"][:, :, (n + 1) % 2], axis=0)

    # Plot the folded data
    x_fold = (t - t0 + 0.5*p) % p - 0.5*p
    plt.errorbar(x_fold, y - other, yerr=yerr, fmt=".k", label="data",
                 zorder=-1000)

    # Plot the folded model
    inds = np.argsort(x_fold)
    inds = inds[np.abs(x_fold)[inds] < 0.3]
    pred = trace["light_curves"][:, inds, n] + trace["mean"][:, None]
    pred = np.percentile(pred, [16, 50, 84], axis=0)
    plt.plot(x_fold[inds], pred[1], color="C1", label="model")
    art = plt.fill_between(x_fold[inds], pred[0], pred[2], color="C1", alpha=0.5,
                           zorder=1000)
    art.set_edgecolor("none")

    # Annotate the plot with the planet's period
    txt = "period = {0:.4f} +/- {1:.4f} d".format(
        np.mean(trace["period"][:, n]), np.std(trace["period"][:, n]))
    plt.annotate(txt, (0, 0), xycoords="axes fraction",
                 xytext=(5, 5), textcoords="offset points",
                 ha="left", va="bottom", fontsize=12)

    plt.legend(fontsize=10, loc=4)
    plt.xlim(-0.5*p, 0.5*p)
    plt.xlabel("time since transit [days]")
    plt.ylabel("relative flux")
    plt.title("planet {0}".format(letter));
    plt.xlim(-0.3, 0.3)
```


planet b
relative flux
0.002
0.001
0.000
-0.001
-0.002
-0.003
period = 15.4479 +/- 0.0023 d
model
data
-0.3
-0.2
-0.1
0.0
0.1
0.2
0.3
time since transit [days]
planet c
relative flux
0.002
0.000
-0.002
-0.004
period = 9.2925 +/- 0.0003 d
model
data
-0.3
-0.2
-0.1
0.0
0.1
0.2
0.3
time since transit [days]

2.4.4 Citations

As described in the citation tutorial, we can use *exoplanet.citations.get_citations_for_model()* to construct an acknowledgement and BibTeX listing that includes the relevant citations for this model. This is especially important here because we have used quite a few model components that should be cited.

```
with model:
    txt, bib = xo.citations.get_citations_for_model()
print(txt)
```

```
This research made use of textsf{exoplanet} citep{exoplanet} and its
dependencies citep{exoplanet:astropy13, exoplanet:astropy18,
exoplanet:espinoza18, exoplanet:exoplanet, exoplanet:kipping13,
exoplanet:luger18, exoplanet:pymc3, exoplanet:theano}.
```

```
print("\n".join(bib.splitlines()[:10]) + "\n...")
```

```
@misc{exoplanet:exoplanet,
  author = {Dan Foreman-Mackey and
            Geert Barentsen and
            Tom Barclay},
   title = {dfm/exoplanet: exoplanet v0.1.4},
   month = feb,
    year = 2019,
     doi = {10.5281/zenodo.2561395},
     url = {https://doi.org/10.5281/zenodo.2561395}
...
```

Note: This tutorial was generated from an IPython notebook that can be downloaded here.

```
theano version: 1.0.4
pymc3 version: 3.6
exoplanet version: 0.1.5
```

2.5 Scalable Gaussian processes in PyMC3

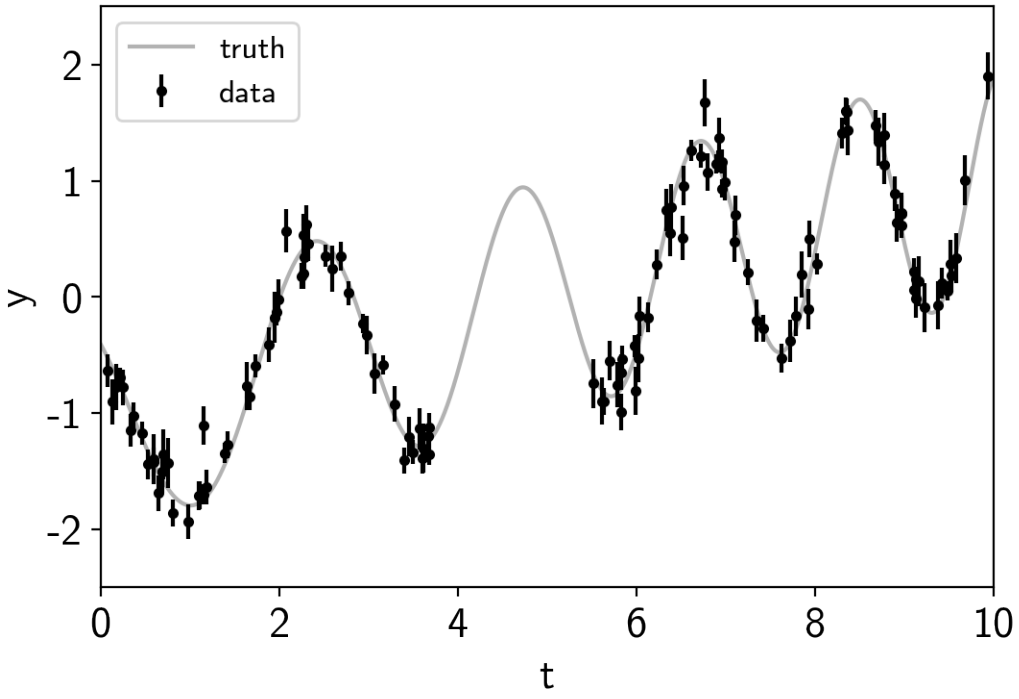
PyMC3 has support for Gaussian Processes (GPs), but this implementation is too slow for many applications in time series astrophysics. So *exoplanet* comes with an implementation of scalable GPs powered by celerite. More information about the algorithm can be found in the celerite docs and in the papers (Paper 1 and Paper 2), but this tutorial will give a hands on demo of how to use celerite in PyMC3.

Note: For the best results, we generally recommend the use of the *exoplanet.gp.terms.SHOTerm*, *exoplanet.gp.terms.Matern32Term*, and *exoplanet.gp.terms.RotationTerm* "terms" because the other terms tend to have unphysical behavior at high frequency.

Let's start with the quickstart demo from the celerite docs. We'll fit the following simulated dataset using the sum of two *exoplanet.gp.terms.SHOTerm* objects.

First, generate the simulated data:

```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(42)

t = np.sort(np.append(
    np.random.uniform(0, 3.8, 57),
    np.random.uniform(5.5, 10, 68),
))  # The input coordinates must be sorted
yerr = np.random.uniform(0.08, 0.22, len(t))
y = 0.2 * (t-5) + np.sin(3*t + 0.1*(t-5)**2) + yerr * np.random.randn(len(t))

true_t = np.linspace(0, 10, 5000)
true_y = 0.2 * (true_t-5) + np.sin(3*true_t + 0.1*(true_t-5)**2)

plt.errorbar(t, y, yerr=yerr, fmt=".k", capsize=0, label="data")
plt.plot(true_t, true_y, "k", lw=1.5, alpha=0.3, label="truth")
plt.legend(fontsize=12)
plt.xlabel("t")
plt.ylabel("y")
plt.xlim(0, 10)
plt.ylim(-2.5, 2.5);
```



This plot shows the simulated data as black points with error bars and the true function is shown as a gray line.

Now let's build the PyMC3 model that we'll use to fit the data. We can see that there's some roughly periodic signal in the data as well as a longer term trend. To capture these two features, we will model this as a mixture of two stochastically driven simple harmonic oscillators (SHO) with the power spectrum:

$$S(\omega) = \sqrt{\frac{2}{\pi}}\frac{{S_1}\,{\omega_1}^4}{(\omega^2 - {\omega_1}^2)^2 + 2\,{\omega_1}^2\,\omega^2} + \sqrt{\frac{2}{\pi}}\frac{{S_2}\,{\omega_2}^4}{(\omega^2 - {\omega_2}^2)^2 + {\omega_2}^2\,\omega^2/Q^2}$$

The first term is `exoplanet.gp.terms.SHOterm` with $Q = 1/\sqrt{2}$ and the second is regular `exoplanet.gp.terms.SHOterm`. This model has 5 free parameters (S_1, ω_1, S_2, ω_2, and Q) and they must all be positive so we'll fit

for the log of each parameter. Using *exoplanet*, this is how you would build this model, choosing more or less arbitrary initial values for the parameters.

```
import pymc3 as pm
import theano.tensor as tt
from exoplanet.gp import terms, GP

with pm.Model() as model:

    logS1 = pm.Normal("logS1", mu=0.0, sd=15.0, testval=np.log(np.var(y)))
    logw1 = pm.Normal("logw1", mu=0.0, sd=15.0, testval=np.log(3.0))
    logS2 = pm.Normal("logS2", mu=0.0, sd=15.0, testval=np.log(np.var(y)))
    logw2 = pm.Normal("logw2", mu=0.0, sd=15.0, testval=np.log(3.0))
    logQ = pm.Normal("logQ", mu=0.0, sd=15.0, testval=0)

    # Set up the kernel an GP
    kernel = terms.SHOTerm(log_S0=logS1, log_w0=logw1, Q=1.0/np.sqrt(2))
    kernel += terms.SHOTerm(log_S0=logS2, log_w0=logw2, log_Q=logQ)
    gp = GP(kernel, t, yerr**2)

    # Add a custom "potential" (log probability function) with the GP likelihood
    pm.Potential("gp", gp.log_likelihood(y))
```

A few comments here:

1. The `term` interface in *exoplanet* only accepts keyword arguments with names given by the `parameter_names` property of the term. But it will also interpret keyword arguments with the name prefaced by `log_` to be the log of the parameter. For example, in this case, we used `log_S0` as the parameter for each term, but `S0=tt.exp(log_S0)` would have been equivalent. This is useful because many of the parameters are required to be positive so fitting the log of those parameters is often best.
2. The third argument to the *exoplanet.gp.GP* constructor should be the *variance* to add along the diagonal, not the standard deviation as in the original celerite implementation.
3. Finally, the *exoplanet.gp.GP* constructor takes an optional argument `J` which specifies the width of the problem if it is known at compile time. Just to be confusing, this is actually two times the `J` from the celerite paper. There are various technical reasons why this is difficult to work out in general and this code will always work if you don't provide a value for `J`, but you can get much better performance (especially for small `J`) if you know what it will be for your problem. In general, most terms cost `J=2` with the exception of a *exoplanet.gp.terms.RealTerm* (which costs *J=1*) and a *exoplanet.gp.terms.RotationTerm* (which costs *J=4*).

To start, let's fit for the maximum a posteriori (MAP) parameters and look the the predictions that those make.

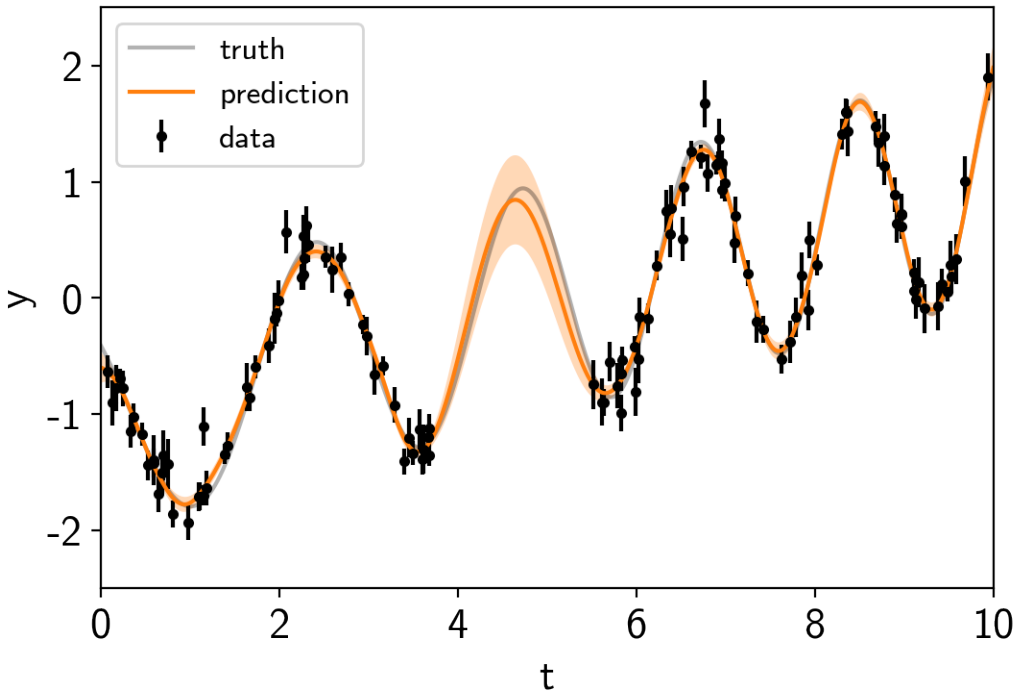
```
import exoplanet as xo
with model:
    map_soln = xo.optimize(start=model.test_point)
```

```
optimizing logp for variables: ['logQ', 'logw2', 'logS2', 'logw1', 'logS1']
message: Optimization terminated successfully.
logp: -41.068631281249985 -> -1.6509594267222276
```

We'll use the *exoplanet.eval_in_model()* function to evaluate the MAP GP model.

```
with model:
    mu, var = xo.eval_in_model(gp.predict(true_t, return_var=True), map_soln)
```

```
plt.errorbar(t, y, yerr=yerr, fmt=".k", capsize=0, label="data")
plt.plot(true_t, true_y, "k", lw=1.5, alpha=0.3, label="truth")

# Plot the prediction and the 1-sigma uncertainty
sd = np.sqrt(var)
art = plt.fill_between(true_t, mu+sd, mu-sd, color="C1", alpha=0.3)
art.set_edgecolor("none")
plt.plot(true_t, mu, color="C1", label="prediction")

plt.legend(fontsize=12)
plt.xlabel("t")
plt.ylabel("y")
plt.xlim(0, 10)
plt.ylim(-2.5, 2.5);
```



Now we can sample this model using PyMC3. There are strong covariances between the parameters so we'll use the custom *exoplanet.PyMC3Sampler* to fit for these covariances during burn-in.

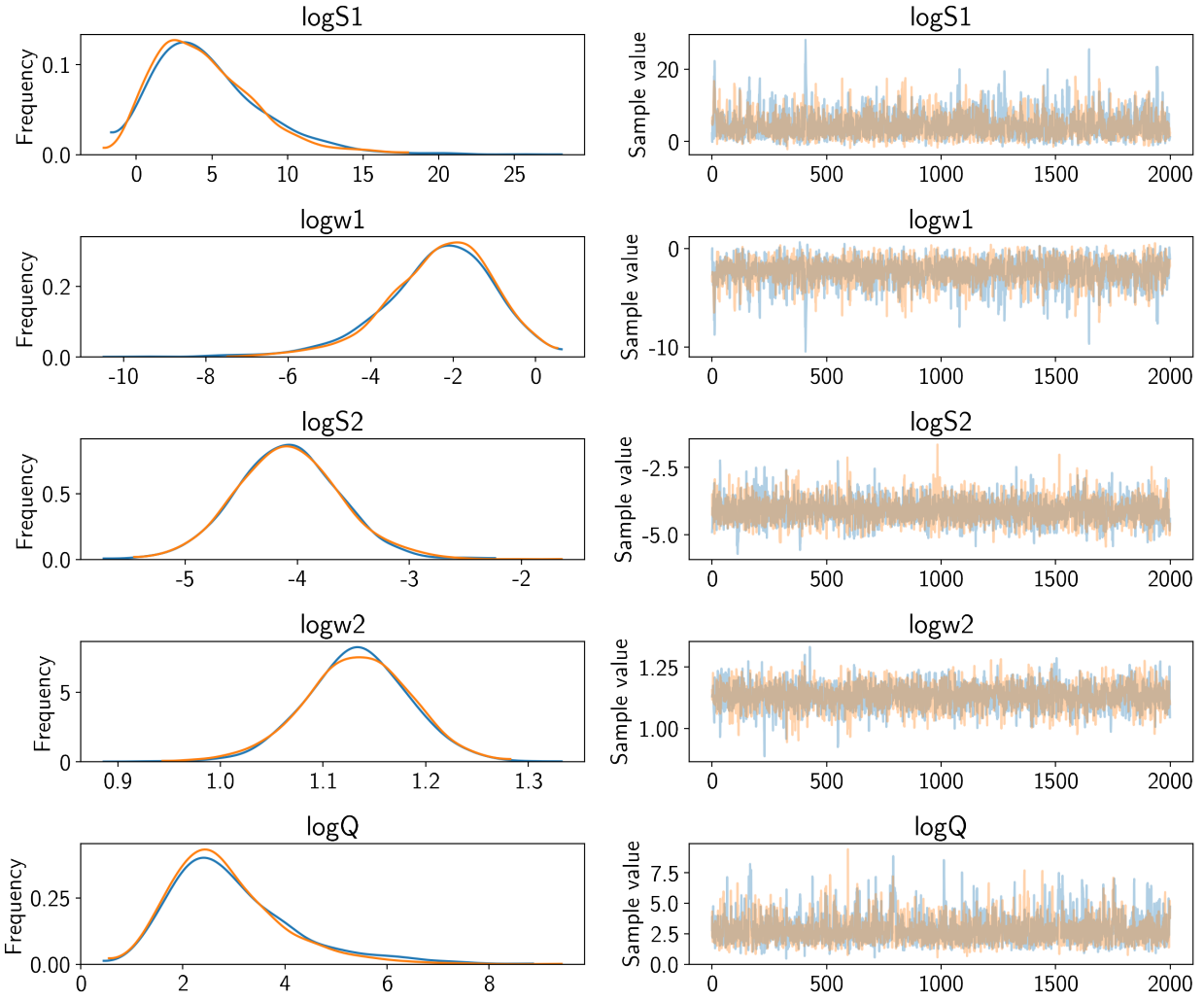
```
from exoplanet.sampling import PyMC3Sampler

sampler = PyMC3Sampler()

with model:
    burnin = sampler.tune(tune=1000, chains=2, start=map_soln)
    trace = sampler.sample(draws=2000, chains=2)
```

```
Sampling 2 chains: 100%|| 154/154 [00:03<00:00, 23.31draws/s]
Sampling 2 chains: 100%|| 54/54 [00:00<00:00, 78.86draws/s]
Sampling 2 chains: 100%|| 104/104 [00:00<00:00, 171.81draws/s]
Sampling 2 chains: 100%|| 204/204 [00:00<00:00, 289.52draws/s]
Sampling 2 chains: 100%|| 404/404 [00:01<00:00, 337.00draws/s]
Sampling 2 chains: 100%|| 1104/1104 [00:02<00:00, 369.35draws/s]
```

```
Sampling 2 chains: 100%|| 104/104 [00:00<00:00, 234.69draws/s]
Multiprocess sampling (2 chains in 4 jobs)
NUTS: [logQ, logw2, logS2, logw1, logS1]
Sampling 2 chains: 100%|| 4000/4000 [00:12<00:00, 315.60draws/s]
There were 2 divergences after tuning. Increase target_accept or␣
↪reparameterize.
The acceptance probability does not match the target. It is 0.
↪9047641255645913, but should be close to 0.8. Try to increase the number␣
↪of tuning steps.
The acceptance probability does not match the target. It is 0.
↪9050065543704073, but should be close to 0.8. Try to increase the number␣
↪of tuning steps.
```

Now we can compute the standard PyMC3 convergence statistics (using `pymc3.summary`) and make a trace plot (using `pymc3.traceplot`).

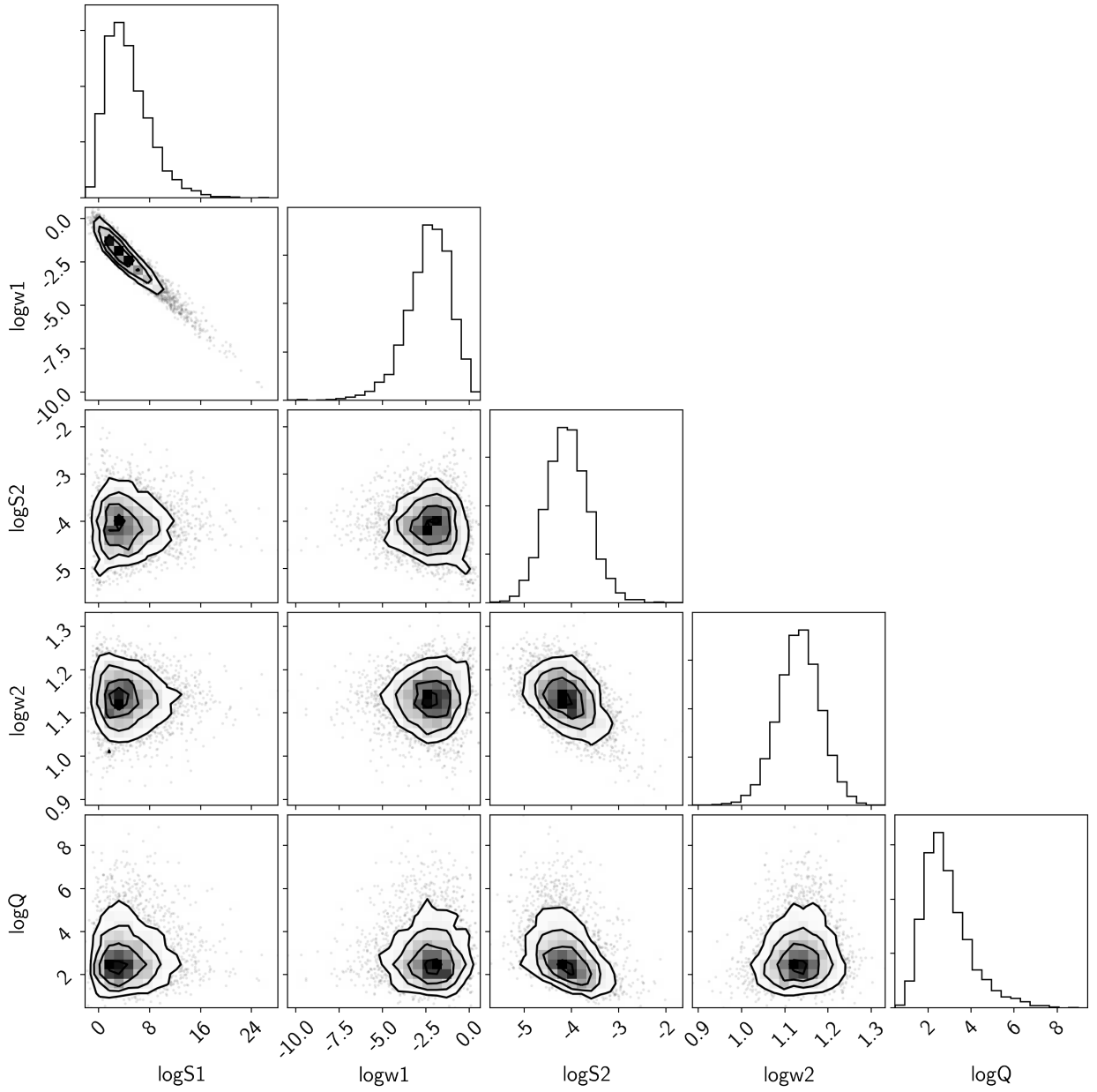
```
pm.traceplot(trace)
pm.summary(trace)
```



That all looks pretty good, but I like to make two other results plots: (1) a corner plot and (2) a posterior predictive plot.

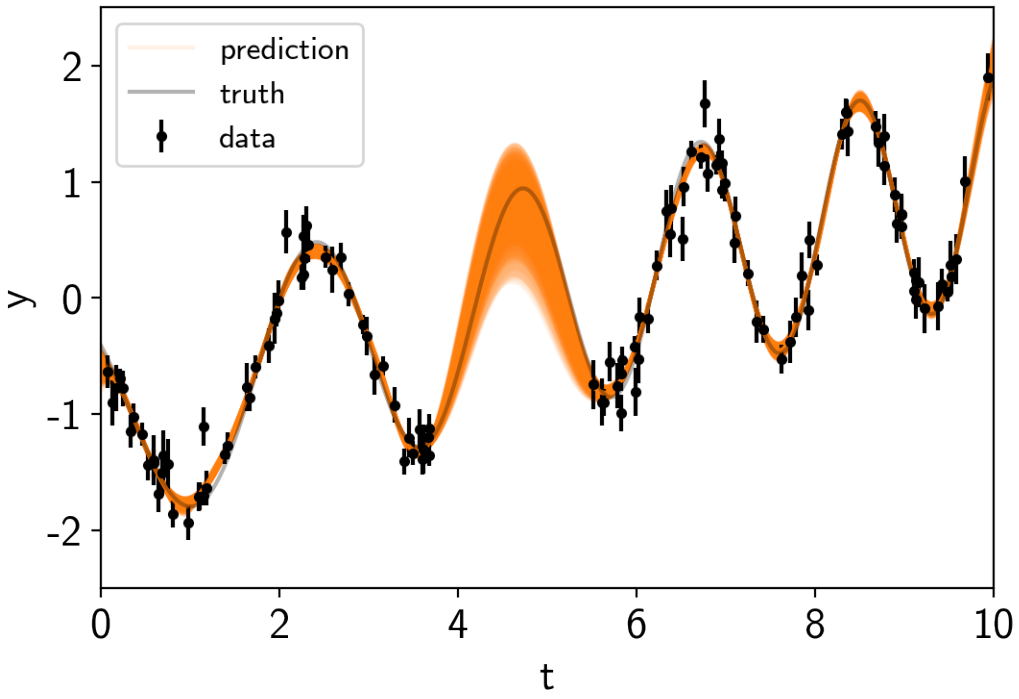
The corner plot is easy using `pymc3.trace_to_dataframe` and I find it useful for understanding the covariances between parameters when debugging.

```
import corner
samples = pm.trace_to_dataframe(trace)
corner.corner(samples);
```



The "posterior predictive" plot that I like to make isn't the same as a "posterior predictive check" (which can be a good thing to do too). Instead, I like to look at the predictions of the model in the space of the data. We could have saved these predictions using a `pymc3.Deterministic` distribution, but that adds some overhead to each evaluation of the model so instead, we can use `exoplanet.utils.get_samples_from_trace()` to loop over a few random samples from the chain and then the *`exoplanet.eval_in_model()`* function to evaluate the prediction just for those samples.

```
# Generate 50 realizations of the prediction sampling randomly from the chain
N_pred = 50
pred_mu = np.empty((N_pred, len(true_t)))
pred_var = np.empty((N_pred, len(true_t)))
with model:
    pred = gp.predict(true_t, return_var=True)
    for i, sample in enumerate(xo.get_samples_from_trace(trace, size=N_pred)):
        pred_mu[i], pred_var[i] = xo.eval_in_model(pred, sample)

# Plot the predictions
for i in range(len(pred_mu)):
    mu = pred_mu[i]
    sd = np.sqrt(pred_var[i])
    label = None if i else "prediction"
    art = plt.fill_between(true_t, mu+sd, mu-sd, color="C1", alpha=0.1)
    art.set_edgecolor("none")
    plt.plot(true_t, mu, color="C1", label=label, alpha=0.1)

plt.errorbar(t, y, yerr=yerr, fmt=".k", capsize=0, label="data")
plt.plot(true_t, true_y, "k", lw=1.5, alpha=0.3, label="truth")
plt.legend(fontsize=12)
plt.xlabel("t")
plt.ylabel("y")
plt.xlim(0, 10)
plt.ylim(-2.5, 2.5);
```



Note: This tutorial was generated from an IPython notebook that can be downloaded here.

```
theano version: 1.0.4
pymc3 version: 3.6
```

(continues on next page)

(continued from previous page)

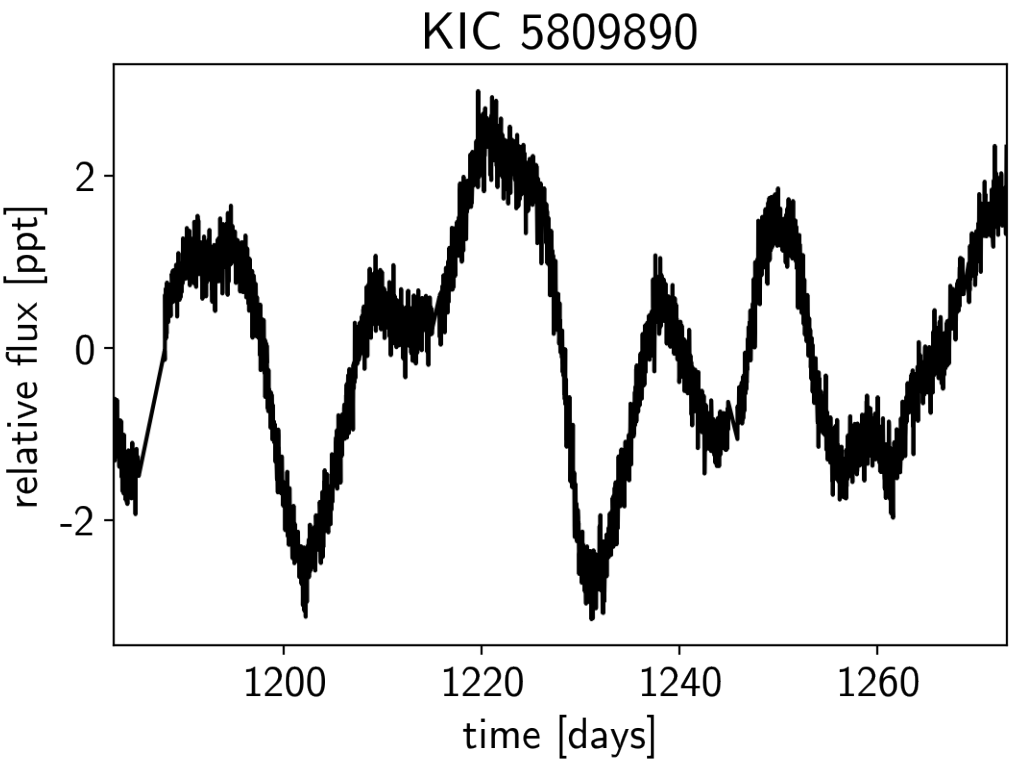
```
exoplanet version: 0.1.5
```

2.6 Gaussian process models for stellar variability

When fitting exoplanets, we also need to fit for the stellar variability and Gaussian Processes (GPs) are often a good descriptive model for this variation. PyMC3 has support for all sorts of general GP models, but *exoplanet* includes support for scalable 1D GPs (see gp for more info) that can work with large datasets. In this tutorial, we go through the process of modeling the light curve of a rotating star observed by Kepler using *exoplanet*.

First, let's download and plot the data:

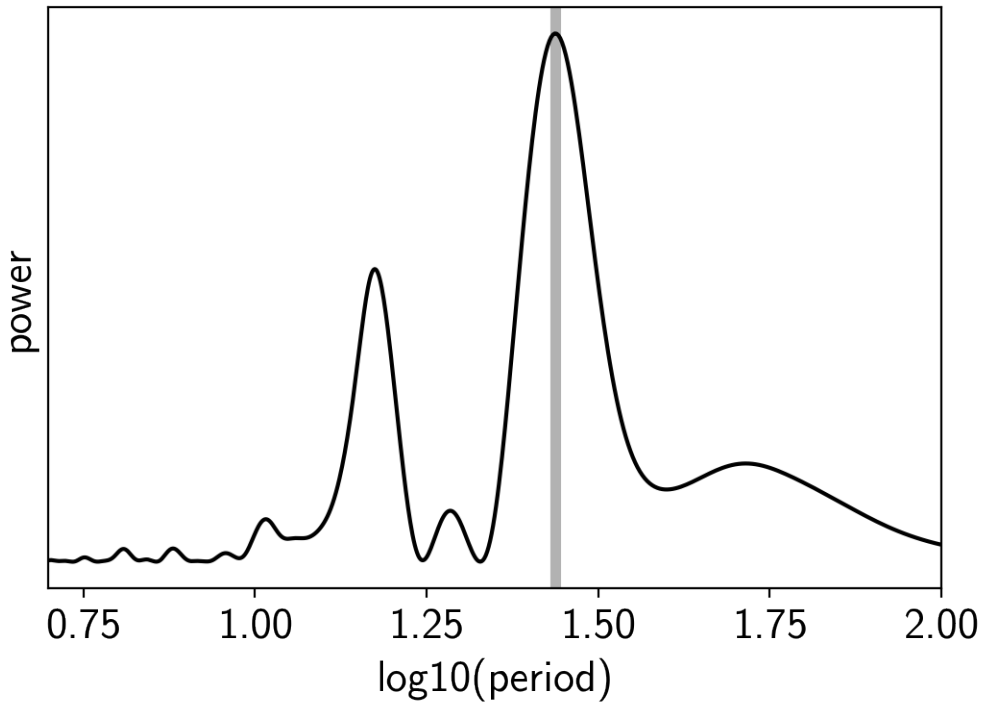
```
import numpy as np
import matplotlib.pyplot as plt
from astropy.io import fits

url = "https://archive.stsci.edu/missions/kepler/lightcurves/0058/005809890/
↪kplr005809890-2012179063303_llc.fits"
with fits.open(url) as hdus:
    data = hdus[1].data

x = data["TIME"]
y = data["PDCSAP_FLUX"]
yerr = data["PDCSAP_FLUX_ERR"]
m = (data["SAP_QUALITY"] == 0) & np.isfinite(x) & np.isfinite(y)

x = np.ascontiguousarray(x[m], dtype=np.float64)
y = np.ascontiguousarray(y[m], dtype=np.float64)
yerr = np.ascontiguousarray(yerr[m], dtype=np.float64)
mu = np.mean(y)
y = (y / mu - 1) * 1e3
yerr = yerr * 1e3 / mu

plt.plot(x, y, "k")
plt.xlim(x.min(), x.max())
plt.xlabel("time [days]")
plt.ylabel("relative flux [ppt]")
plt.title("KIC 5809890");
```



2.6.1 A Gaussian process model for stellar variability

This looks like the light curve of a rotating star, and it has been shown that it is possible to model this variability by using a quasiperiodic Gaussian process. To start with, let's get an estimate of the rotation period using the Lomb-Scargle periodogram:

```
import exoplanet as xo

results = xo.estimators.lomb_scargle_estimator(
    x, y, max_peaks=1, min_period=5.0, max_period=100.0,
    samples_per_peak=50)

peak = results["peaks"][0]
freq, power = results["periodogram"]
plt.plot(-np.log10(freq), power, "k")
plt.axvline(np.log10(peak["period"]), color="k", lw=4, alpha=0.3)
plt.xlim((-np.log10(freq)).min(), (-np.log10(freq)).max())
plt.yticks([])
plt.xlabel("log10(period)")
plt.ylabel("power");
```



Now, using this initialization, we can set up the GP model in *exoplanet*. We'll use the *exoplanet.gp.terms.RotationTerm* kernel that is a mixture of two simple harmonic oscillators with periods separated by a factor of two. As you can see from the periodogram above, this might be a good model for this light curve and I've found that it works well in many cases.

```
import pymc3 as pm
import theano.tensor as tt

with pm.Model() as model:

    # The mean flux of the time series
    mean = pm.Normal("mean", mu=0.0, sd=10.0)

    # A jitter term describing excess white noise
    logs2 = pm.Normal("logs2", mu=2*np.log(np.min(yerr)), sd=5.0)

    # The parameters of the RotationTerm kernel
    logamp = pm.Normal("logamp", mu=np.log(np.var(y)), sd=5.0)
    logperiod = pm.Normal("logperiod", mu=np.log(peak["period"]), sd=5.0)
    logQ0 = pm.Normal("logQ0", mu=1.0, sd=10.0)
    logdeltaQ = pm.Normal("logdeltaQ", mu=2.0, sd=10.0)
    mix = pm.Uniform("mix", lower=0, upper=1.0)

    # Track the period as a deterministic
    period = pm.Deterministic("period", tt.exp(logperiod))

    # Set up the Gaussian Process model
    kernel = xo.gp.terms.RotationTerm(
        log_amp=logamp,
        period=period,
        log_Q0=logQ0,
        log_deltaQ=logdeltaQ,
```

(continues on next page)

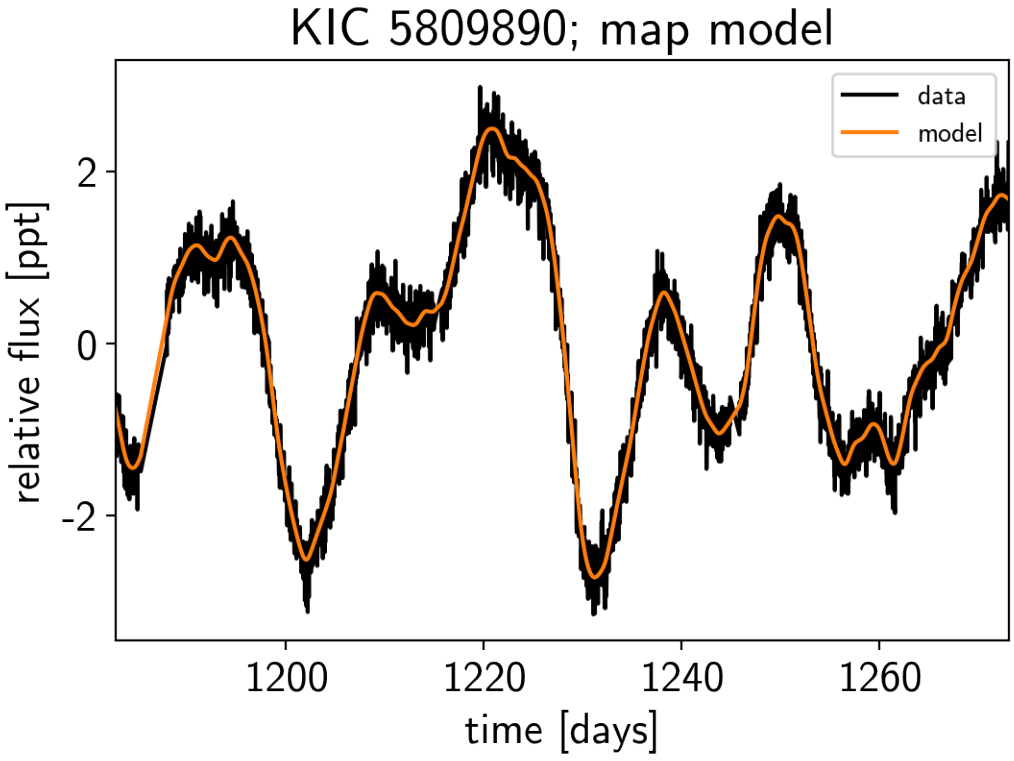
(continued from previous page)

```
        mix=mix
    )
    gp = xo.gp.GP(kernel, x, yerr**2 + tt.exp(logs2), J=4)

    # Compute the Gaussian Process likelihood and add it into the
    # the PyMC3 model as a "potential"
    pm.Potential("loglike", gp.log_likelihood(y - mean))

    # Compute the mean model prediction for plotting purposes
    pm.Deterministic("pred", gp.predict())

    # Optimize to find the maximum a posteriori parameters
    map_soln = xo.optimize(start=model.test_point)
```

```
optimizing logp for variables: ['mix_interval__', 'logdeltaQ', 'logQ0', 'logperiod',
↪'logamp', 'logs2', 'mean']
message: Optimization terminated successfully.
logp: 515.8061433750984 -> 692.7159093512395
```

Now that we have the model set up, let's plot the maximum a posteriori model prediction.

```
plt.plot(x, y, "k", label="data")
plt.plot(x, map_soln["pred"], color="C1", label="model")
plt.xlim(x.min(), x.max())
plt.legend(fontsize=10)
plt.xlabel("time [days]")
plt.ylabel("relative flux [ppt]")
plt.title("KIC 5809890; map model");
```



That looks pretty good! Now let's sample from the posterior using a *exoplanet.PyMC3Sampler*.

```
np.random.seed(42)
sampler = xo.PyMC3Sampler(finish=200)
with model:
    sampler.tune(tune=2000, start=map_soln, step_kwargs=dict(target_accept=0.9))
    trace = sampler.sample(draws=2000)
```

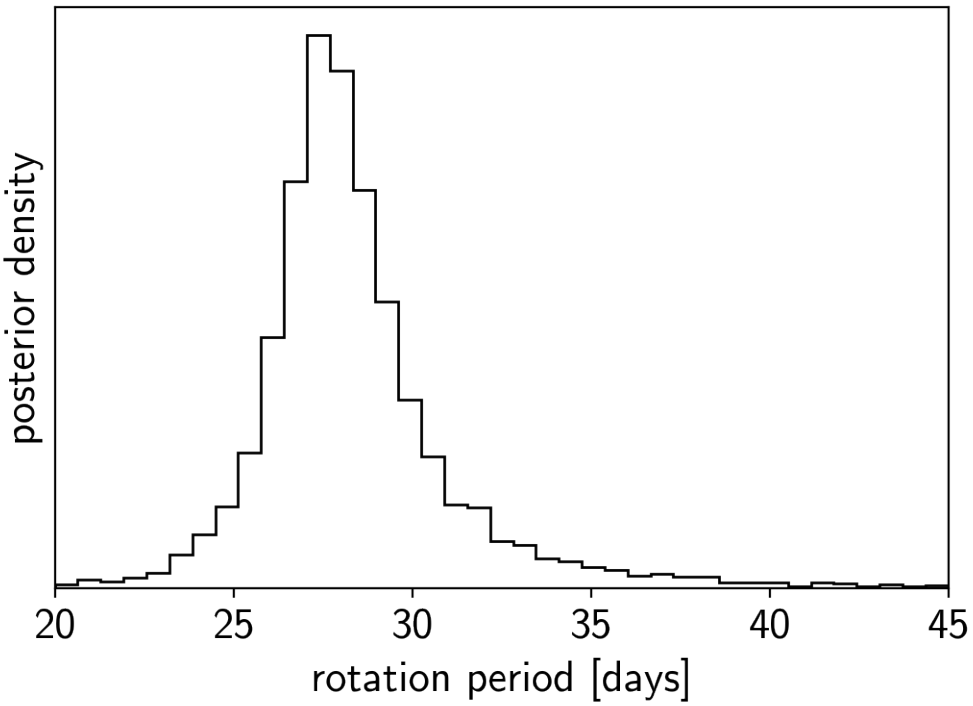
```
Sampling 4 chains: 100%|| 308/308 [00:26<00:00,  3.07draws/s]
Sampling 4 chains: 100%|| 108/108 [00:06<00:00, 17.79draws/s]
Sampling 4 chains: 100%|| 208/208 [00:03<00:00, 61.30draws/s]
Sampling 4 chains: 100%|| 408/408 [00:06<00:00, 61.59draws/s]
Sampling 4 chains: 100%|| 808/808 [00:11<00:00, 72.43draws/s]
Sampling 4 chains: 100%|| 1608/1608 [00:24<00:00, 65.44draws/s]
Sampling 4 chains: 100%|| 4608/4608 [01:00<00:00, 24.49draws/s]
Sampling 4 chains: 100%|| 808/808 [00:13<00:00, 62.03draws/s]
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [mix, logdeltaQ, logQ0, logperiod, logamp, logs2, mean]
Sampling 4 chains: 100%|| 8000/8000 [01:32<00:00, 20.91draws/s]
There was 1 divergence after tuning. Increase target_accept or reparameterize.
There were 3 divergences after tuning. Increase target_accept or␣
↪reparameterize.
There were 3 divergences after tuning. Increase target_accept or␣
↪reparameterize.
There were 2 divergences after tuning. Increase target_accept or␣
↪reparameterize.
The number of effective samples is smaller than 25% for some parameters.
```

Now we can do the usual convergence checks:

```
pm.summary(trace, varnames=["mix", "logdeltaQ", "logQ0", "logperiod", "logamp", "logs2
↪", "mean"])
```

And plot the posterior distribution over rotation period:

```
period_samples = trace["period"]
bins = np.linspace(20, 45, 40)
plt.hist(period_samples, bins, histtype="step", color="k")
plt.yticks([])
plt.xlim(bins.min(), bins.max())
plt.xlabel("rotation period [days]")
plt.ylabel("posterior density");
```



Note: This tutorial was generated from an IPython notebook that can be downloaded here.

```
theano version: 1.0.4
pymc3 version: 3.6
exoplanet version: 0.1.5
```

2.7 Case study: K2-24, putting it all together

In this tutorial, we will combine many of the previous tutorials to perform a fit of the K2-24 system using the K2 transit data and the RVs from Petigura et al. (2016). This is the same system that we fit in the rv tutorial and we'll combine that model with the transit model from the transit tutorial and the Gaussian Process noise model from the stellar-variability tutorial.

2.7.1 Datasets and initializations

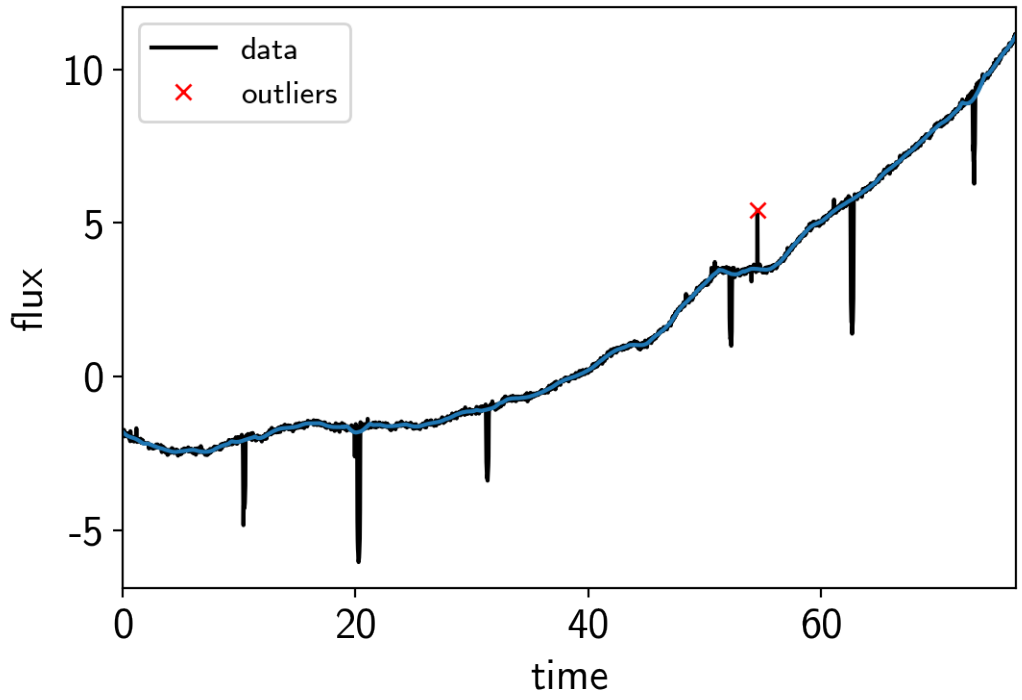
To get started, let's download the relevant datasets. First, the transit light curve from Everest:

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

from astropy.io import fits
from scipy.signal import savgol_filter

# Download the data
lc_url = "https://archive.stsci.edu/hlsps/everest/v2/c02/203700000/71098/hlsp_everest_
↪k2_llc_203771098-c02_kepler_v2.0_lc.fits"
```

(continues on next page)

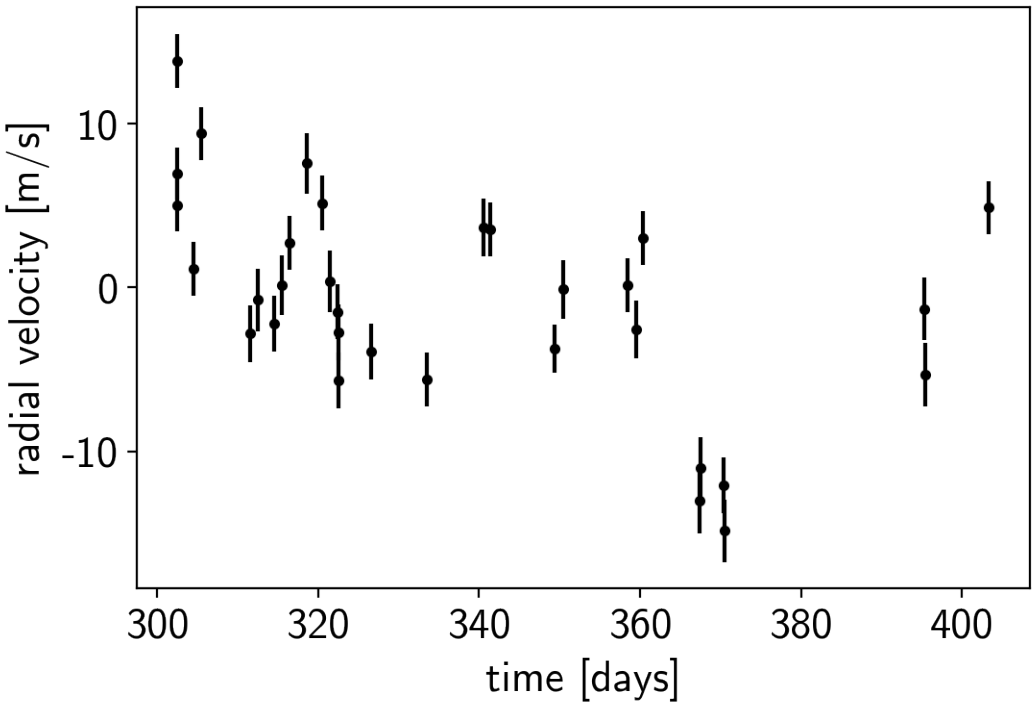
(continued from previous page)

```
with fits.open(lc_url) as hdus:
    lc = hdus[1].data
    lc_hdr = hdus[1].header

# Work out the exposure time
texp = lc_hdr["FRAMETIM"] * lc_hdr["NUM_FRM"]
texp /= 60.0 * 60.0 * 24.0

# Mask bad data
m = (np.arange(len(lc)) > 100) & np.isfinite(lc["FLUX"]) & np.isfinite(lc["TIME"])
bad_bits=[1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 16, 17]
qual = lc["QUALITY"]
for b in bad_bits:
    m &= qual & 2 ** (b - 1) == 0

# Convert to parts per thousand
x = lc["TIME"][m]
y = lc["FLUX"][m]
mu = np.median(y)
y = (y / mu - 1) * 1e3

# Identify outliers
m = np.ones(len(y), dtype=bool)
for i in range(10):
    y_prime = np.interp(x, x[m], y[m])
    smooth = savgol_filter(y_prime, 101, polyorder=3)
    resid = y - smooth
    sigma = np.sqrt(np.mean(resid**2))
    m0 = np.abs(resid) < 3*sigma
    if m.sum() == m0.sum():
        m = m0
        break
    m = m0

# Only discard positive outliers
m = resid < 3*sigma

# Shift the data so that the K2 data start at t=0. This tends to make the fit
# better behaved since t0 covaries with period.
x_ref = np.min(x[m])
x -= x_ref

# Plot the data
plt.plot(x, y, "k", label="data")
plt.plot(x, smooth)
plt.plot(x[~m], y[~m], "xr", label="outliers")
plt.legend(fontsize=12)
plt.xlim(x.min(), x.max())
plt.xlabel("time")
plt.ylabel("flux")

# Make sure that the data type is consistent
x = np.ascontiguousarray(x[m], dtype=np.float64)
y = np.ascontiguousarray(y[m], dtype=np.float64)
smooth = np.ascontiguousarray(smooth[m], dtype=np.float64)
```



Then the RVs from RadVel:

```
import pandas as pd

url = "https://raw.githubusercontent.com/California-Planet-Search/radvel/master/
↪example_data/epic203771098.csv"
data = pd.read_csv(url, index_col=0)

# Don't forget to remove the time offset from above!
x_rv = np.array(data.t) - x_ref
y_rv = np.array(data.vel)
yerr_rv = np.array(data.errvel)

plt.errorbar(x_rv, y_rv, yerr=yerr_rv, fmt=".k")
plt.xlabel("time [days]")
plt.ylabel("radial velocity [m/s]");
```

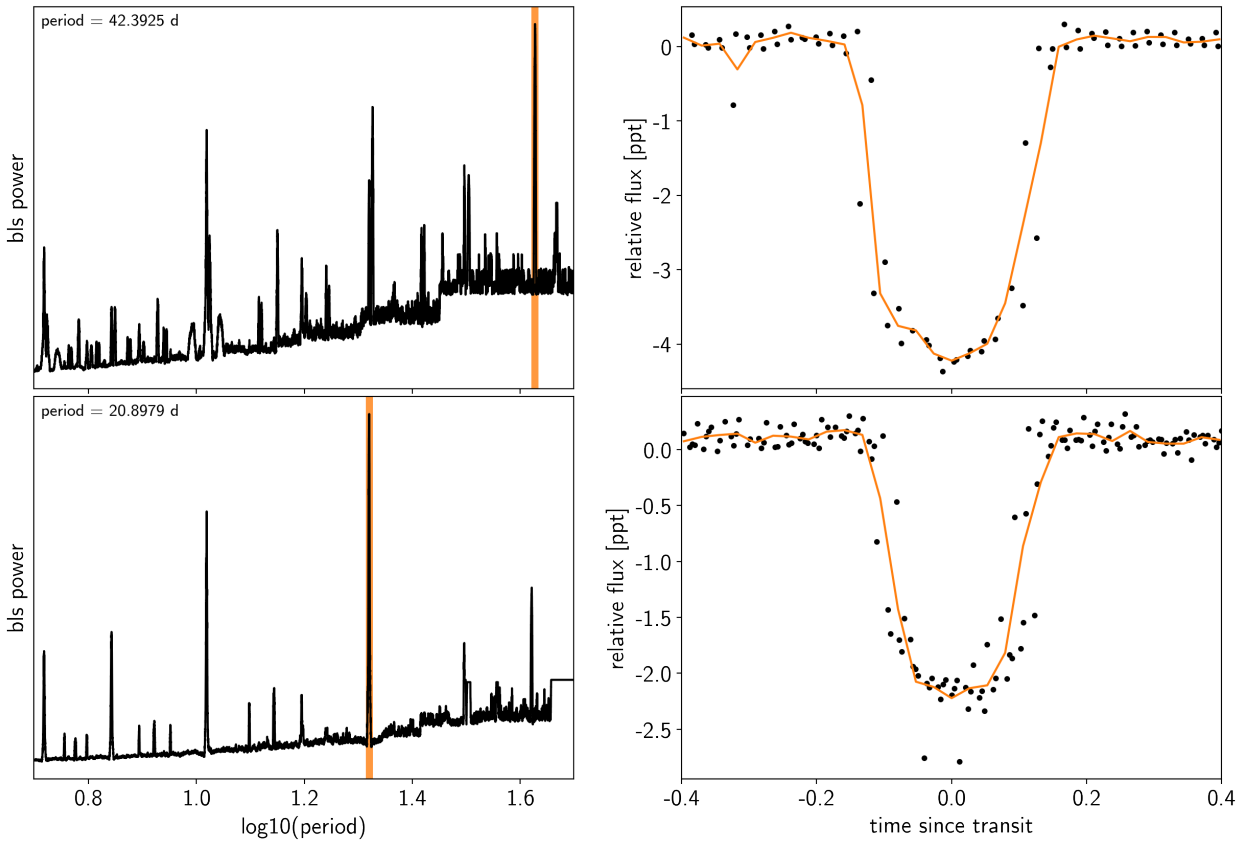


We can initialize the transit parameters using the box least squares periodogram from AstroPy. (Note: you'll need AstroPy v3.1 or more recent to use this feature.) A full discussion of transit detection and vetting is beyond the scope of this tutorial so let's assume that we know that there are two periodic transiting planets in this dataset.

```
from astropy.stats import BoxLeastSquares

m = np.zeros(len(x), dtype=bool)
period_grid = np.exp(np.linspace(np.log(5), np.log(50), 50000))
bls_results = []
periods = []
t0s = []
depths = []

# Compute the periodogram for each planet by iteratively masking out
# transits from the higher signal to noise planets. Here we're assuming
# that we know that there are exactly two planets.
for i in range(2):
    bls = BoxLeastSquares(x[~m], y[~m] - smooth[~m])
    bls_power = bls.power(period_grid, 0.1, oversample=20)
    bls_results.append(bls_power)

    # Save the highest peak as the planet candidate
    index = np.argmax(bls_power.power)
    periods.append(bls_power.period[index])
    t0s.append(bls_power.transit_time[index])
    depths.append(bls_power.depth[index])

    # Mask the data points that are in transit for this candidate
    m |= bls.transit_mask(x, periods[-1], 0.5, t0s[-1])
```

Let's plot the initial transit estimates based on these periodograms:

```
fig, axes = plt.subplots(len(bls_results), 2, figsize=(15, 10))

for i in range(len(bls_results)):
    # Plot the periodogram
    ax = axes[i, 0]
    ax.axvline(np.log10(periods[i]), color="C1", lw=5, alpha=0.8)
    ax.plot(np.log10(bls_results[i].period), bls_results[i].power, "k")
    ax.annotate("period = {0:.4f} d".format(periods[i]),
                (0, 1), xycoords="axes fraction",
                xytext=(5, -5), textcoords="offset points",
                va="top", ha="left", fontsize=12)
    ax.set_ylabel("bls power")
    ax.set_yticks([])
    ax.set_xlim(np.log10(period_grid.min()), np.log10(period_grid.max()))
    if i < len(bls_results) - 1:
        ax.set_xticklabels([])
    else:
        ax.set_xlabel("log10(period)")

    # Plot the folded transit
    ax = axes[i, 1]
    p = periods[i]
    x_fold = (x - t0s[i] + 0.5*p) % p - 0.5*p
    m = np.abs(x_fold) < 0.4
    ax.plot(x_fold[m], y[m] - smooth[m], ".k")

    # Overplot the phase binned light curve
    bins = np.linspace(-0.41, 0.41, 32)
    denom, _ = np.histogram(x_fold, bins)
    num, _ = np.histogram(x_fold, bins, weights=y - smooth)
    denom[num == 0] = 1.0
    ax.plot(0.5*(bins[1:] + bins[:-1]), num / denom, color="C1")

    ax.set_xlim(-0.4, 0.4)
    ax.set_ylabel("relative flux [ppt]")
    if i < len(bls_results) - 1:
        ax.set_xticklabels([])
    else:
        ax.set_xlabel("time since transit")

fig.subplots_adjust(hspace=0.02)
```



The discovery paper for K2-24 (Petigura et al. (2016)) includes the following estimates of the stellar mass and radius in Solar units:

```
M_star_petigura = 1.12, 0.05
R_star_petigura = 1.21, 0.11
```

Finally, using this stellar mass, we can also estimate the minimum masses of the planets given these transit parameters.

```
import exoplanet as xo
import astropy.units as u

msini = xo.estimate_minimum_mass(periods, x_rv, y_rv, yerr_rv, t0s=t0s, m_star=M_star_
↪petigura[0])
msini = msini.to(u.M_earth)
print(msini)
```

```
[32.80060146 23.89885976] earthMass
```

2.7.2 A joint transit and radial velocity model in PyMC3

Now, let's define our full model in *PyMC3*. There's a lot going on here, but I've tried to comment it and most of it should be familiar from the previous tutorials (rv, transit, gp, and stellar-variability). In this case, I've put the model inside a model "factory" function because we'll do some sigma clipping below.

```
import pymc3 as pm
import theano.tensor as tt
```

(continues on next page)

(continued from previous page)

```
t_rv = np.linspace(x_rv.min()-5, x_rv.max()+5, 1000)

def build_model(mask=None, start=None):
    if mask is None:
        mask = np.ones(len(x), dtype=bool)
    with pm.Model() as model:

        # Parameters for the stellar properties
        mean = pm.Normal("mean", mu=0.0, sd=10.0)
        u_star = xo.distributions.QuadLimbDark("u_star")
        m_star = pm.Normal("m_star", mu=M_star_petigura[0], sd=M_star_petigura[1])
        r_star = pm.Normal("r_star", mu=R_star_petigura[0], sd=R_star_petigura[1])

        # Prior to require physical parameters
        pm.Potential("m_star_prior", tt.switch(m_star > 0, 0, -np.inf))
        pm.Potential("r_star_prior", tt.switch(r_star > 0, 0, -np.inf))

        # Orbital parameters for the planets
        logm = pm.Normal("logm", mu=np.log(msini.value), sd=1, shape=2)
        logP = pm.Normal("logP", mu=np.log(periods), sd=1, shape=2)
        t0 = pm.Normal("t0", mu=np.array(t0s), sd=1, shape=2)
        logr = pm.Normal("logr", mu=0.5*np.log(1e-3*np.array(depths)) + np.log(R_star_
↪petigura[0]),
                         sd=1.0, shape=2)
        r_pl = pm.Deterministic("r_pl", tt.exp(logr))
        ror = pm.Deterministic("ror", r_pl / r_star)
        b = pm.Uniform("b", lower=0, upper=1, testval=0.5+np.zeros(2),
                       shape=2)
        ecc = pm.Uniform("ecc", lower=0, upper=0.99, shape=2,
                         testval=np.array([0.1, 0.1]))
        omega = xo.distributions.Angle("omega", shape=2)

        # RV jitter & a quadratic RV trend
        logs_rv = pm.Normal("logs_rv", mu=np.log(np.median(yerr_rv)), sd=5)
        trend = pm.Normal("trend", mu=0, sd=10.0**-np.arange(3)[::-1], shape=3)

        # Transit jitter & GP parameters
        logs2 = pm.Normal("logs2", mu=np.log(np.var(y[mask])), sd=10)
        logw0_guess = np.log(2*np.pi/10)
        logw0 = pm.Normal("logw0", mu=logw0_guess, sd=10)

        # We'll parameterize using the maximum power (S_0 * w_0^4) instead of
        # S_0 directly because this removes some of the degeneracies between
        # S_0 and omega_0
        logpower = pm.Normal("logpower",
                             mu=np.log(np.var(y[mask]))+4*logw0_guess,
                             sd=10)
        logS0 = pm.Deterministic("logS0", logpower - 4 * logw0)

        # Tracking planet parameters
        period = pm.Deterministic("period", tt.exp(logP))
        m_pl = pm.Deterministic("m_pl", tt.exp(logm))

        # Orbit model
        orbit = xo.orbits.KeplerianOrbit(
            r_star=r_star, m_star=m_star,
```

(continues on next page)

(continued from previous page)

```
            period=period, t0=t0, b=b, m_planet=m_pl,
            ecc=ecc, omega=omega,
            m_planet_units=msini.unit)

        # Compute the model light curve using starry
        light_curves = xo.StarryLightCurve(u_star).get_light_curve(
            orbit=orbit, r=r_pl, t=x[mask], texp=texp, oversample=15)*1e3
        light_curve = pm.math.sum(light_curves, axis=-1) + mean
        pm.Deterministic("light_curves", light_curves)

        # GP model for the light curve
        kernel = xo.gp.terms.SHOTerm(log_S0=logS0, log_w0=logw0, Q=1/np.sqrt(2))
        gp = xo.gp.GP(kernel, x[mask], tt.exp(logs2) + tt.zeros(mask.sum()), J=2)
        pm.Potential("transit_obs", gp.log_likelihood(y[mask] - light_curve))
        pm.Deterministic("gp_pred", gp.predict())

        # Set up the RV model and save it as a deterministic
        # for plotting purposes later
        vrad = orbit.get_radial_velocity(x_rv)
        pm.Deterministic("vrad", vrad)

        # Define the background RV model
        A = np.vander(x_rv - 0.5*(x_rv.min() + x_rv.max()), 3)
        bkg = pm.Deterministic("bkg", tt.dot(A, trend))

        # The likelihood for the RVs
        rv_model = pm.Deterministic("rv_model", tt.sum(vrad, axis=-1) + bkg)
        err = tt.sqrt(yerr_rv**2 + tt.exp(2*logs_rv))
        pm.Normal("obs", mu=rv_model, sd=err, observed=y_rv)

        vrad_pred = orbit.get_radial_velocity(t_rv)
        pm.Deterministic("vrad_pred", vrad_pred)
        A_pred = np.vander(t_rv - 0.5*(x_rv.min() + x_rv.max()), 3)
        bkg_pred = pm.Deterministic("bkg_pred", tt.dot(A_pred, trend))
        pm.Deterministic("rv_model_pred", tt.sum(vrad_pred, axis=-1) + bkg_pred)

        # Fit for the maximum a posteriori parameters, I've found that I can get
        # a better solution by trying different combinations of parameters in turn
        if start is None:
            start = model.test_point
        map_soln = xo.optimize(start=start, vars=[trend])
        map_soln = xo.optimize(start=map_soln, vars=[logs2])
        map_soln = xo.optimize(start=map_soln, vars=[logr, b])
        map_soln = xo.optimize(start=map_soln, vars=[logP, t0])
        map_soln = xo.optimize(start=map_soln, vars=[logs2, logpower])
        map_soln = xo.optimize(start=map_soln, vars=[logw0])
        map_soln = xo.optimize(start=map_soln)
        map_soln = xo.optimize(start=map_soln, vars=[logm, ecc, omega])
        map_soln = xo.optimize(start=map_soln)

    return model, map_soln
```

```
model0, map_soln0 = build_model()
```

```
optimizing logp for variables: ['trend']
message: Optimization terminated successfully.
```

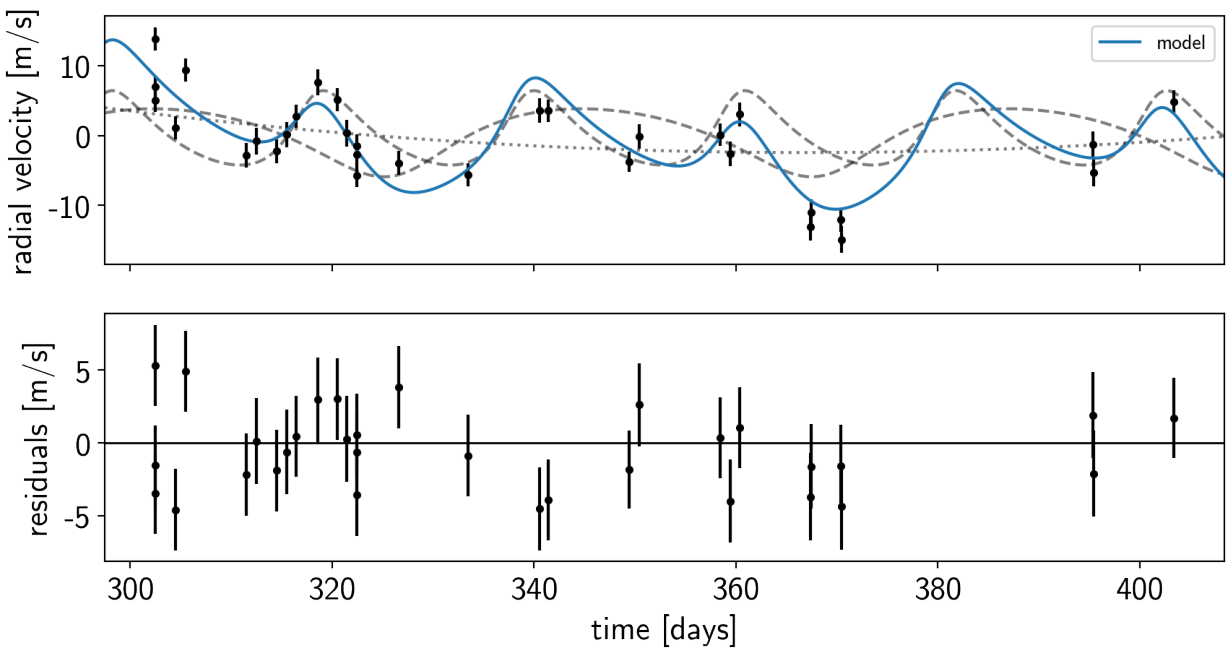
(continues on next page)

(continued from previous page)

```
logp: -8216.986023857186 -> -8201.731058138012
optimizing logp for variables: ['logs2']
message: Optimization terminated successfully.
logp: -8201.731058138012 -> 2595.6621154808026
optimizing logp for variables: ['b_interval__', 'logr']
message: Desired error not necessarily achieved due to precision loss.
logp: 2595.6621154808026 -> 2818.9312788464613
optimizing logp for variables: ['t0', 'logP']
message: Desired error not necessarily achieved due to precision loss.
logp: 2818.9312788464613 -> 3696.09961998513
optimizing logp for variables: ['logpower', 'logs2']
message: Optimization terminated successfully.
logp: 3696.09961998513 -> 4310.350888676674
optimizing logp for variables: ['logw0']
message: Desired error not necessarily achieved due to precision loss.
logp: 4310.350888676674 -> 4381.909298235777
optimizing logp for variables: ['logpower', 'logw0', 'logs2', 'trend', 'logs_rv',
↪'omega_angle__', 'ecc_interval__', 'b_interval__', 'logr', 't0', 'logP', 'logm', 'r_
↪star', 'm_star', 'u_star_quadlimbdark__', 'mean']
message: Desired error not necessarily achieved due to precision loss.
logp: 4381.909298235777 -> 4381.92717865675
optimizing logp for variables: ['omega_angle__', 'ecc_interval__', 'logm']
message: Optimization terminated successfully.
logp: 4381.92717865675 -> 4468.096852994163
optimizing logp for variables: ['logpower', 'logw0', 'logs2', 'trend', 'logs_rv',
↪'omega_angle__', 'ecc_interval__', 'b_interval__', 'logr', 't0', 'logP', 'logm', 'r_
↪star', 'm_star', 'u_star_quadlimbdark__', 'mean']
message: Desired error not necessarily achieved due to precision loss.
logp: 4468.096852994163 -> 4776.768827256388
```

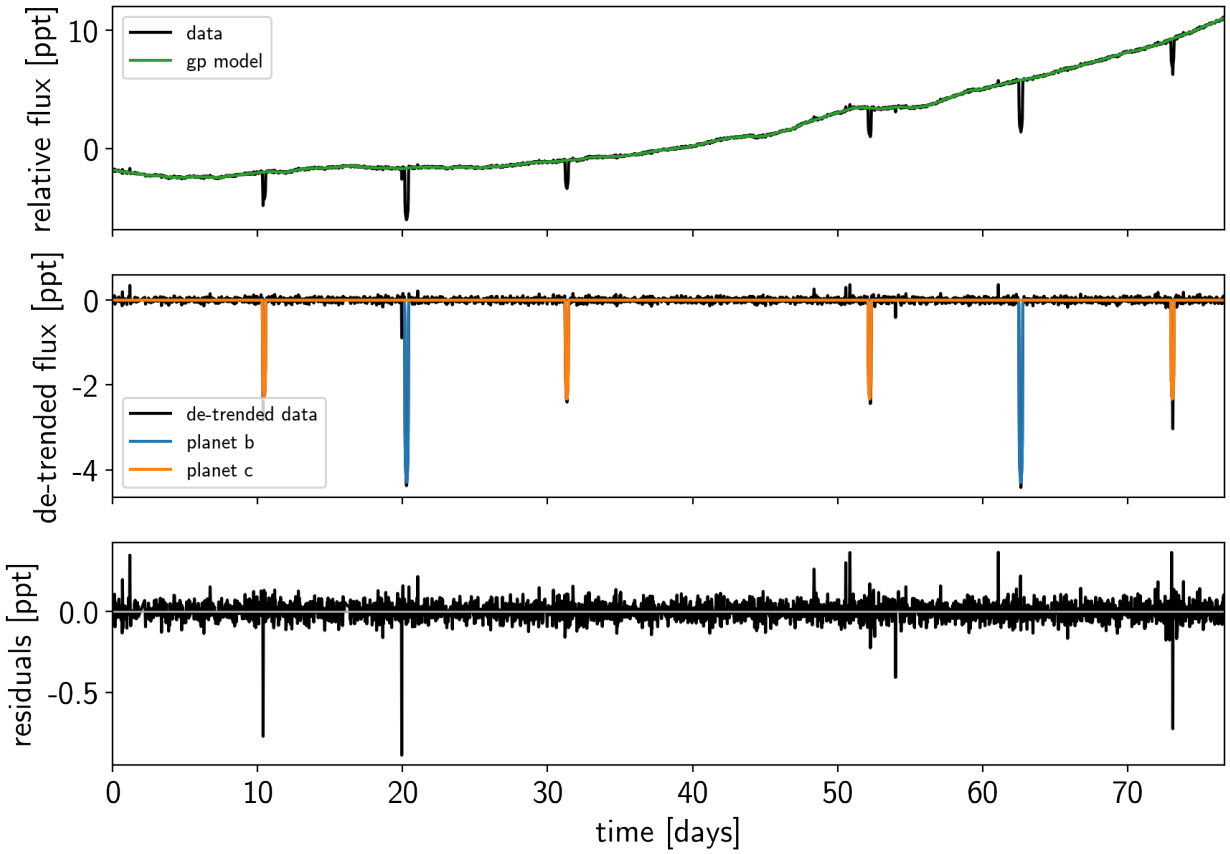
Now let's plot the map radial velocity model.

```
def plot_rv_curve(soln):
    fig, axes = plt.subplots(2, 1, figsize=(10, 5), sharex=True)

    ax = axes[0]
    ax.errorbar(x_rv, y_rv, yerr=yerr_rv, fmt=".k")
    ax.plot(t_rv, soln["vrad_pred"], "--k", alpha=0.5)
    ax.plot(t_rv, soln["bkg_pred"], ":k", alpha=0.5)
    ax.plot(t_rv, soln["rv_model_pred"], label="model")
    ax.legend(fontsize=10)
    ax.set_ylabel("radial velocity [m/s]")

    ax = axes[1]
    err = np.sqrt(yerr_rv**2+np.exp(2*soln["logs_rv"]))
    ax.errorbar(x_rv, y_rv - soln["rv_model"], yerr=err, fmt=".k")
    ax.axhline(0, color="k", lw=1)
    ax.set_ylabel("residuals [m/s]")
    ax.set_xlim(t_rv.min(), t_rv.max())
    ax.set_xlabel("time [days]")

plot_rv_curve(map_soln0)
```



That looks pretty similar to what we got in rv. Now let's also plot the transit model.

```
def plot_light_curve(soln, mask=None):
    if mask is None:
        mask = np.ones(len(x), dtype=bool)

    fig, axes = plt.subplots(3, 1, figsize=(10, 7), sharex=True)

    ax = axes[0]
    ax.plot(x[mask], y[mask], "k", label="data")
    gp_mod = soln["gp_pred"] + soln["mean"]
    ax.plot(x[mask], gp_mod, color="C2", label="gp model")
    ax.legend(fontsize=10)
    ax.set_ylabel("relative flux [ppt]")

    ax = axes[1]
    ax.plot(x[mask], y[mask] - gp_mod, "k", label="de-trended data")
    for i, l in enumerate("bc"):
        mod = soln["light_curves"][:, i]
        ax.plot(x[mask], mod, label="planet {0}".format(l))
    ax.legend(fontsize=10, loc=3)
    ax.set_ylabel("de-trended flux [ppt]")

    ax = axes[2]
    mod = gp_mod + np.sum(soln["light_curves"], axis=-1)
    ax.plot(x[mask], y[mask] - mod, "k")
    ax.axhline(0, color="#aaaaaa", lw=1)
    ax.set_ylabel("residuals [ppt]")
    ax.set_xlim(x[mask].min(), x[mask].max())
    ax.set_xlabel("time [days]")

    return fig

plot_light_curve(map_soln0);
```

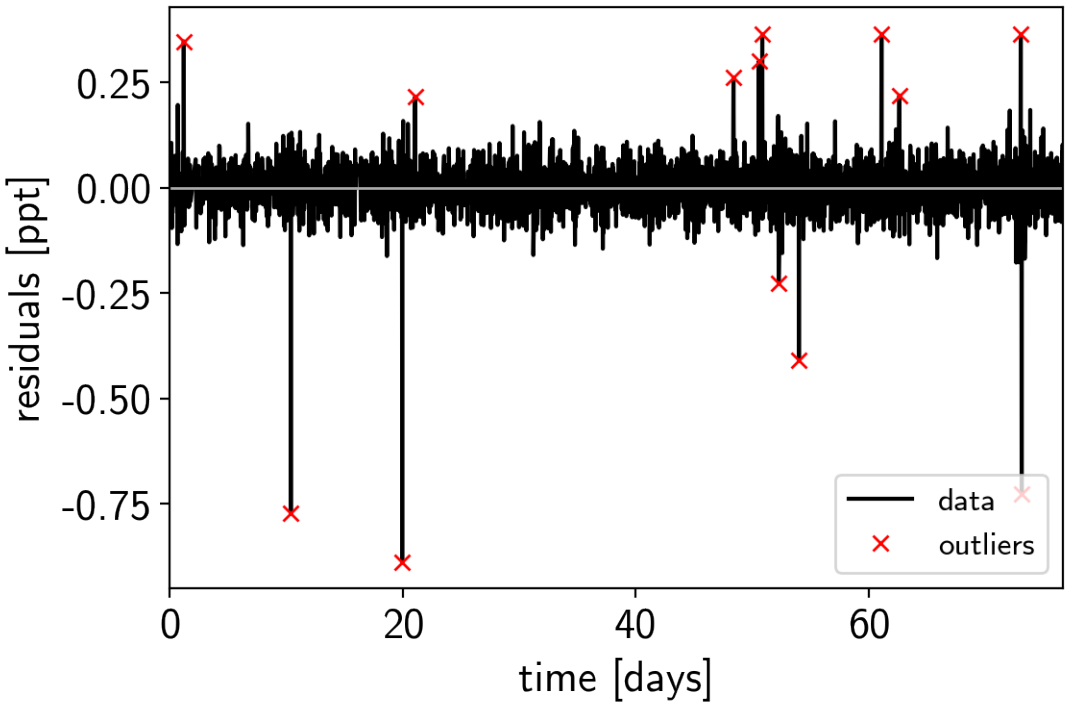


There are still a few outliers in the light curve and it can be useful to remove those before doing the full fit because both the GP and transit parameters can be sensitive to this.

2.7.3 Sigma clipping

To remove the outliers, we'll look at the empirical RMS of the residuals away from the GP + transit model and remove anything that is more than a 7-sigma outlier.

```
mod = map_soln0["gp_pred"] + map_soln0["mean"] + np.sum(map_soln0["light_curves"],␣
↪axis=-1)
resid = y - mod
rms = np.sqrt(np.median(resid**2))
mask = np.abs(resid) < 7 * rms

plt.plot(x, resid, "k", label="data")
plt.plot(x[~mask], resid[~mask], "xr", label="outliers")
plt.axhline(0, color="#aaaaaa", lw=1)
plt.ylabel("residuals [ppt]")
plt.xlabel("time [days]")
plt.legend(fontsize=12, loc=4)
plt.xlim(x.min(), x.max());
```



That looks better. Let's re-build our model with this sigma-clipped dataset.

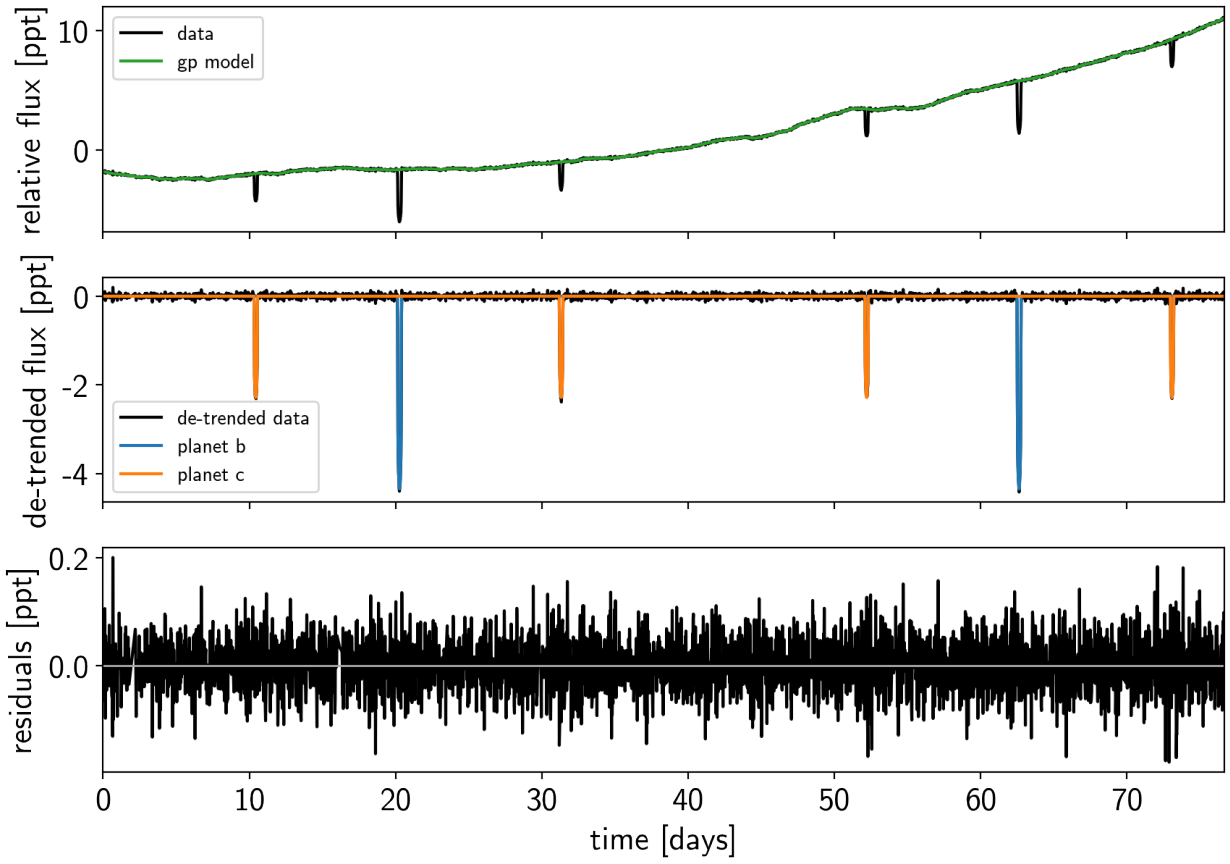
```
model, map_soln = build_model(mask, map_soln0)
plot_light_curve(map_soln, mask);
```

```
optimizing logp for variables: ['trend']
message: Optimization terminated successfully.
logp: 5226.925440678209 -> 5226.92544067821
optimizing logp for variables: ['logs2']
message: Desired error not necessarily achieved due to precision loss.
logp: 5226.92544067821 -> 5309.094883167375
optimizing logp for variables: ['b_interval__', 'logr']
message: Desired error not necessarily achieved due to precision loss.
logp: 5309.094883167375 -> 5320.14909739983
optimizing logp for variables: ['t0', 'logP']
message: Desired error not necessarily achieved due to precision loss.
logp: 5320.14909739983 -> 5321.564832945187
optimizing logp for variables: ['logpower', 'logs2']
message: Optimization terminated successfully.
logp: 5321.564832945187 -> 5322.302087822063
optimizing logp for variables: ['logw0']
message: Optimization terminated successfully.
logp: 5322.302087822063 -> 5322.336502468334
optimizing logp for variables: ['logpower', 'logw0', 'logs2', 'trend', 'logs_rv',
↪'omega_angle__', 'ecc_interval__', 'b_interval__', 'logr', 't0', 'logP', 'logm', 'r_
↪star', 'm_star', 'u_star_quadlimbdark__', 'mean']
message: Desired error not necessarily achieved due to precision loss.
logp: 5322.336502468334 -> 5324.074012099883
optimizing logp for variables: ['omega_angle__', 'ecc_interval__', 'logm']
message: Desired error not necessarily achieved due to precision loss.
logp: 5324.074012099883 -> 5324.074012099883
optimizing logp for variables: ['logpower', 'logw0', 'logs2', 'trend', 'logs_rv',
↪'omega_angle__', 'ecc_interval__', 'b_interval__', 'logr', 't0', 'logP', 'logm', 'r_
↪star', 'm_star', 'u_star_quadlimbdark__', 'mean']
```

(continues on next page)

(continued from previous page)

```
message: Desired error not necessarily achieved due to precision loss.
logp: 5324.074012099883 -> 5324.074012099883
```



Great! Now we're ready to sample.

2.7.4 Sampling

The sampling for this model is the same as for all the previous tutorials, but it takes a bit longer (about 2 hours on my laptop). This is partly because the model is more expensive to compute than the previous ones and partly because there are some non-affine degeneracies in the problem (for example between impact parameter and eccentricity). It might be worth thinking about reparameterizations (in terms of duration instead of eccentricity), but that's beyond the scope of this tutorial. Besides, using more traditional MCMC methods, this would have taken a lot more than 2 hours to get >1000 effective samples!

```
np.random.seed(123)
sampler = xo.PyMC3Sampler(window=200, start=500, finish=1000)
with model:
    burnin = sampler.tune(tune=4500, start=map_soln,
                          step_kwargs=dict(target_accept=0.9),
                          chains=4)
```

```
Sampling 4 chains: 100%|| 2008/2008 [23:33<00:00,  3.93s/draws]
Sampling 4 chains: 100%|| 808/808 [03:01<00:00,  1.20draws/s]
Sampling 4 chains: 100%|| 1608/1608 [05:41<00:00,  1.20draws/s]
```

(continues on next page)

(continued from previous page)

```
Sampling 4 chains: 100%|| 3208/3208 [11:21<00:00,  1.09draws/s]
The chain contains only diverging samples. The model is probably misspecified.
Sampling 4 chains: 100%|| 10408/10408 [39:05<00:00,  1.06draws/s]
Sampling 4 chains: 100%|| 4008/4008 [15:01<00:00,  1.10draws/s]
The chain contains only diverging samples. The model is probably misspecified.
The chain contains only diverging samples. The model is probably misspecified.
```

```
with model:
    trace = sampler.sample(draws=3000, chains=4)
```

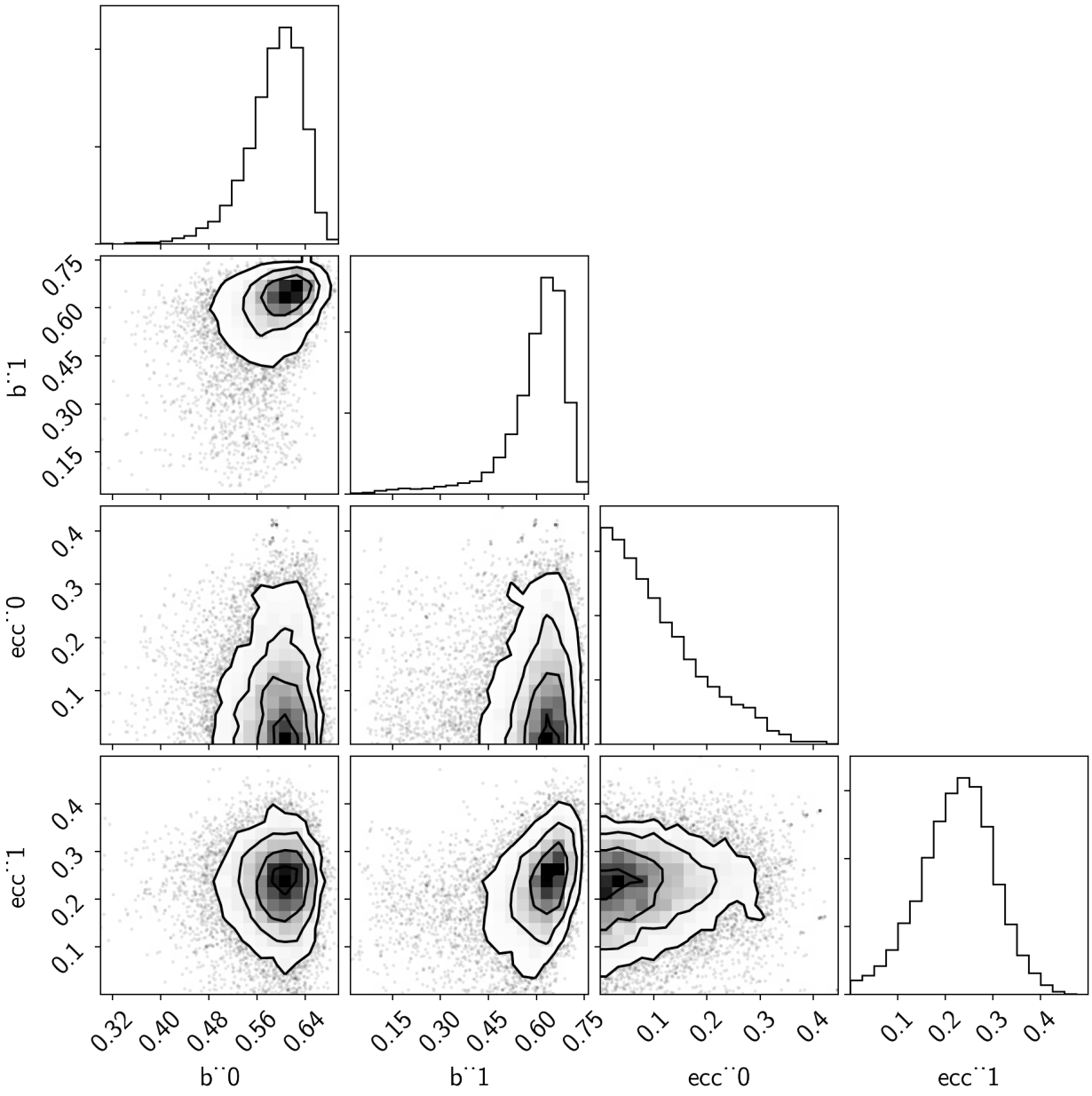
```
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [logpower, logw0, logs2, trend, logs_rv, omega, ecc, b, logr, t0, logP,␣
↪logm, r_star, m_star, u_star, mean]
Sampling 4 chains: 100%|| 12000/12000 [43:44<00:00,  1.03draws/s]
There were 603 divergences after tuning. Increase target_accept or␣
↪reparameterize.
There were 646 divergences after tuning. Increase target_accept or␣
↪reparameterize.
There were 581 divergences after tuning. Increase target_accept or␣
↪reparameterize.
There were 653 divergences after tuning. Increase target_accept or␣
↪reparameterize.
The number of effective samples is smaller than 10% for some parameters.
```

Let's look at the convergence diagnostics for some of the key parameters:

```
pm.summary(trace, varnames=["period", "r_pl", "m_pl", "ecc", "b"])
```

As you see, the effective number of samples for the impact parameters and eccentricites are lower than for the other parameters. This is because of the correlations that I mentioned above:

```
import corner
varnames = ["b", "ecc"]
samples = pm.trace_to_dataframe(trace, varnames=varnames)
corner.corner(samples);
```



2.7.5 Phase plots

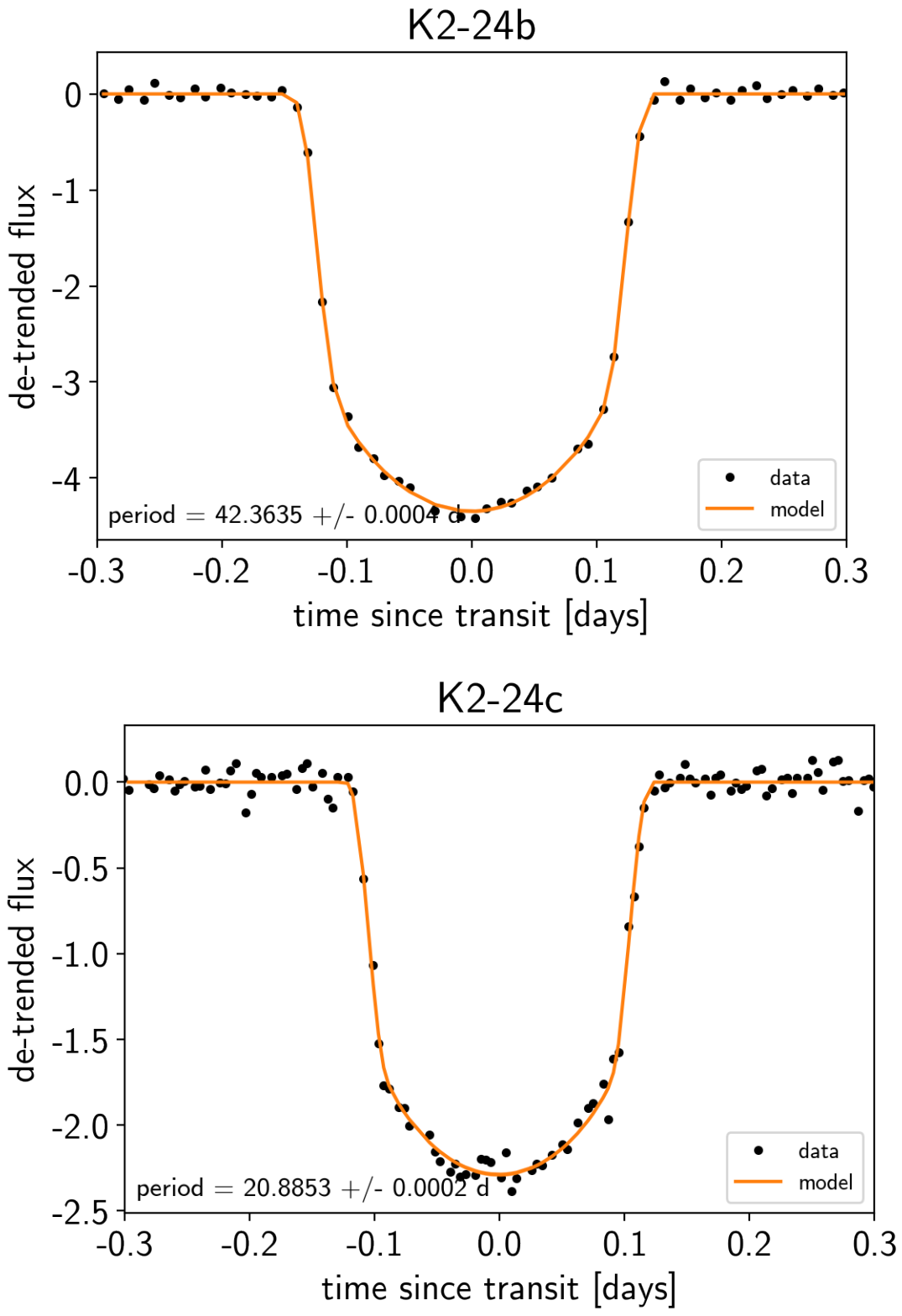
Finally, as in the rv and transit tutorials, we can make folded plots of the transits and the radial velocities and compare to the posterior model predictions. (Note: planets b and c in this tutorial are swapped compared to the labels from Petigura et al. (2016))

```
for n, letter in enumerate("bc"):
    plt.figure()

    # Compute the GP prediction
    gp_mod = np.median(trace["gp_pred"] + trace["mean"][:, None], axis=0)

    # Get the posterior median orbital parameters
```

(continues on next page)

(continued from previous page)

```
    p = np.median(trace["period"][:, n])
    t0 = np.median(trace["t0"][:, n])

    # Compute the median of posterior estimate of the contribution from
    # the other planet. Then we can remove this from the data to plot
    # just the planet we care about.
    other = np.median(trace["light_curves"][:, :, (n + 1) % 2], axis=0)

    # Plot the folded data
    x_fold = (x[mask] - t0 + 0.5*p) % p - 0.5*p
    plt.plot(x_fold, y[mask] - gp_mod - other, ".k", label="data", zorder=-1000)

    # Plot the folded model
    inds = np.argsort(x_fold)
    inds = inds[np.abs(x_fold)[inds] < 0.3]
    pred = trace["light_curves"][:, inds, n]
    pred = np.percentile(pred, [16, 50, 84], axis=0)
    plt.plot(x_fold[inds], pred[1], color="C1", label="model")
    art = plt.fill_between(x_fold[inds], pred[0], pred[2], color="C1", alpha=0.5,
                           zorder=1000)
    art.set_edgecolor("none")

    # Annotate the plot with the planet's period
    txt = "period = {0:.4f} +/- {1:.4f} d".format(
        np.mean(trace["period"][:, n]), np.std(trace["period"][:, n]))
    plt.annotate(txt, (0, 0), xycoords="axes fraction",
                 xytext=(5, 5), textcoords="offset points",
                 ha="left", va="bottom", fontsize=12)

    plt.legend(fontsize=10, loc=4)
    plt.xlim(-0.5*p, 0.5*p)
    plt.xlabel("time since transit [days]")
    plt.ylabel("de-trended flux")
    plt.title("K2-24{0}".format(letter));
    plt.xlim(-0.3, 0.3)
```

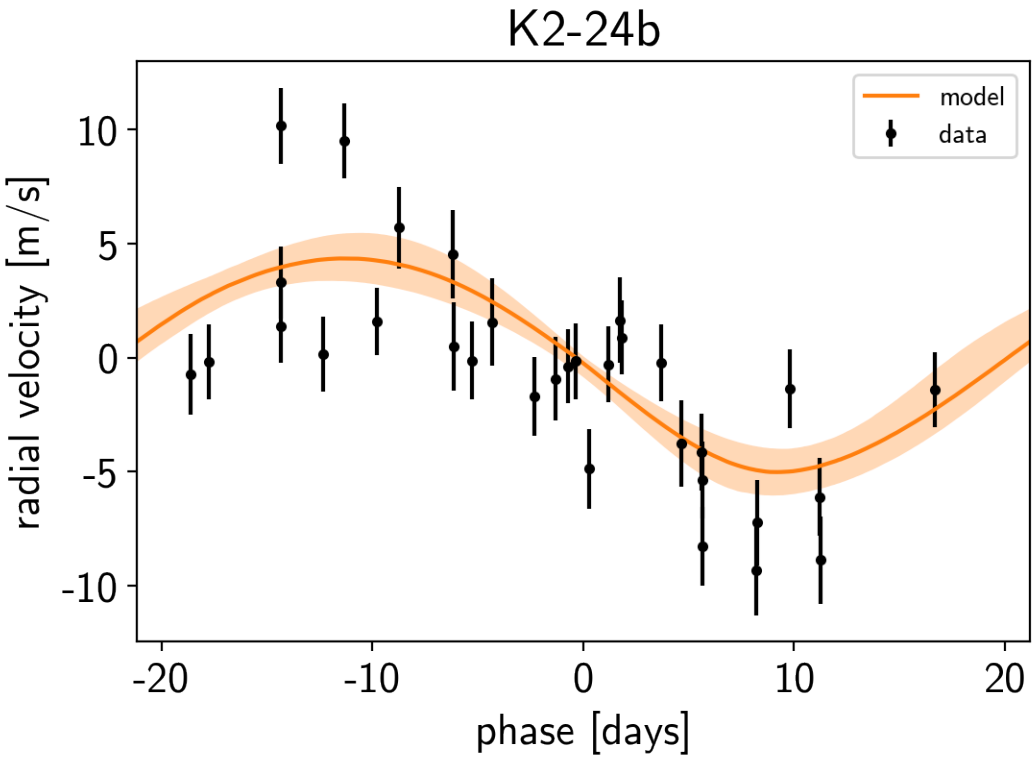


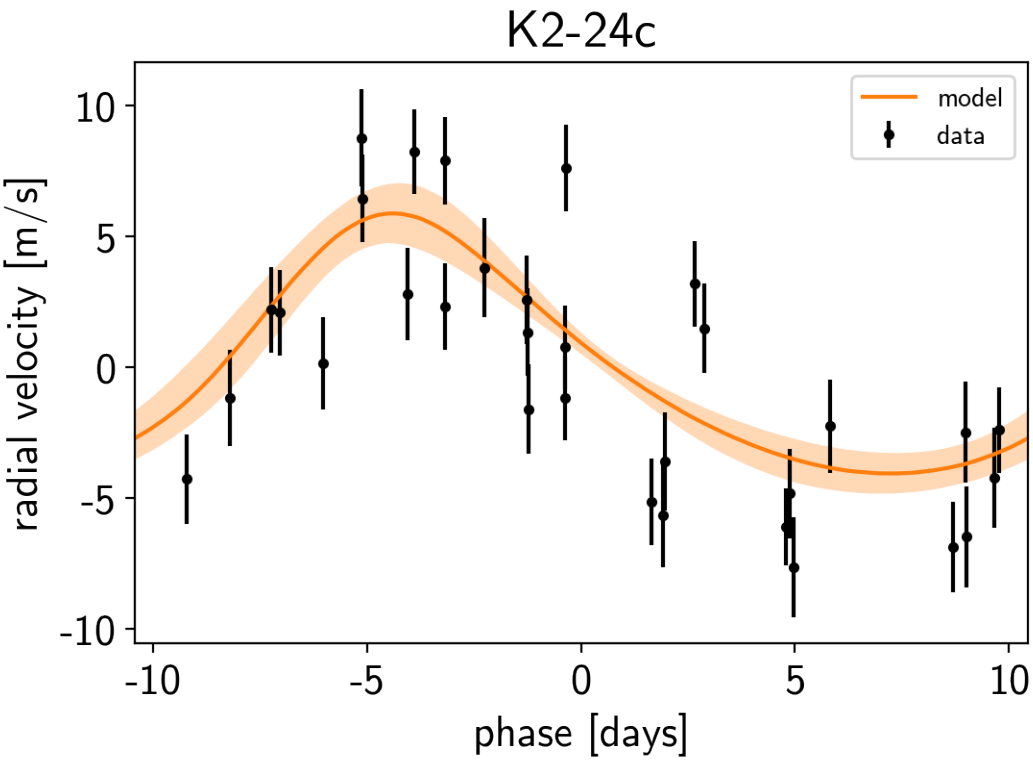

```
for n, letter in enumerate("bc"):
    plt.figure()
```

(continues on next page)

(continued from previous page)

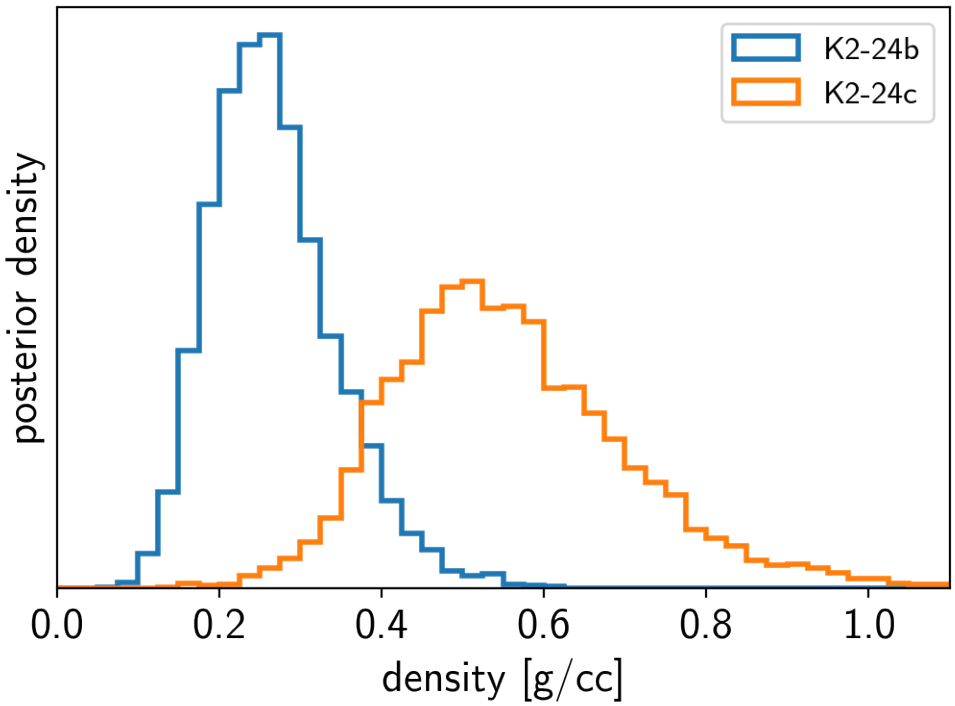
```
    # Get the posterior median orbital parameters
    p = np.median(trace["period"][:, n])
    t0 = np.median(trace["t0"][:, n])

    # Compute the median of posterior estimate of the background RV
    # and the contribution from the other planet. Then we can remove
    # this from the data to plot just the planet we care about.
    other = np.median(trace["vrad"][:, :, (n + 1) % 2], axis=0)
    other += np.median(trace["bkg"], axis=0)

    # Plot the folded data
    x_fold = (x_rv - t0 + 0.5*p) % p - 0.5*p
    plt.errorbar(x_fold, y_rv - other, yerr=yerr_rv, fmt=".k", label="data")

    # Compute the posterior prediction for the folded RV model for this
    # planet
    t_fold = (t_rv - t0 + 0.5*p) % p - 0.5*p
    inds = np.argsort(t_fold)
    pred = np.percentile(trace["vrad_pred"][:, inds, n], [16, 50, 84], axis=0)
    plt.plot(t_fold[inds], pred[1], color="C1", label="model")
    art = plt.fill_between(t_fold[inds], pred[0], pred[2], color="C1", alpha=0.3)
    art.set_edgecolor("none")

    plt.legend(fontsize=10)
    plt.xlim(-0.5*p, 0.5*p)
    plt.xlabel("phase [days]")
    plt.ylabel("radial velocity [m/s]")
    plt.title("K2-24{0}".format(letter));
```





We can also compute the posterior constraints on the planet densities.

```
volume = 4/3*np.pi*trace["r_pl"]**3
density = u.Quantity(trace["m_pl"] / volume, unit=u.M_earth / u.R_sun**3)
density = density.to(u.g / u.cm**3).value

bins = np.linspace(0, 1.1, 45)
for n, letter in enumerate("bc"):
    plt.hist(density[:, n], bins, histtype="step", lw=2,
             label="K2-24{0}".format(letter), density=True)
plt.yticks([])
plt.legend(fontsize=12)
plt.xlim(bins[0], bins[-1])
plt.xlabel("density [g/cc]")
plt.ylabel("posterior density");
```



2.7.6 Citations

As described in the citation tutorial, we can use *exoplanet.citations.get_citations_for_model()* to construct an acknowledgement and BibTeX listing that includes the relevant citations for this model.

```
with model:
    txt, bib = xo.citations.get_citations_for_model()
print(txt)
```

```
This research made use of textsf{exoplanet} citep{exoplanet} and its
dependencies citep{exoplanet:astropy13, exoplanet:astropy18,
exoplanet:exoplanet, exoplanet:foremanmackey17, exoplanet:foremanmackey18,
exoplanet:kipping13, exoplanet:luger18, exoplanet:pymc3, exoplanet:theano}.
```

```
print("\n".join(bib.splitlines()[:10]) + "\n...")
```

```
@misc{exoplanet:exoplanet,
  author = {Dan Foreman-Mackey and
            Geert Barentsen and
            Tom Barclay},
   title = {dfm/exoplanet: exoplanet v0.1.4},
   month = feb,
    year = 2019,
     doi = {10.5281/zenodo.2561395},
     url = {https://doi.org/10.5281/zenodo.2561395}
...
```

Note: This tutorial was generated from an IPython notebook that can be downloaded here.

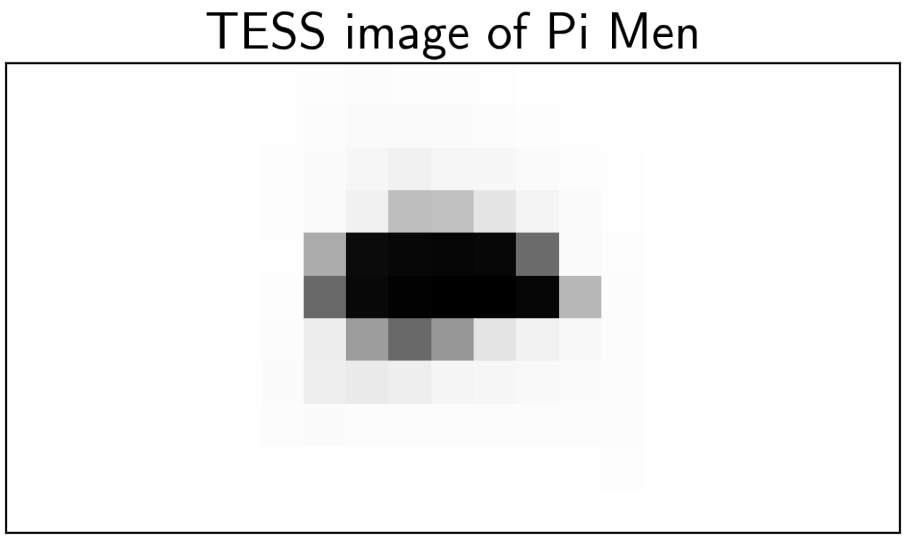
```
theano version: 1.0.4
pymc3 version: 3.6
exoplanet version: 0.1.5
```

2.8 Fitting TESS data

In this tutorial, we will reproduce the fits to the transiting planet in the Pi Mensae system discovered by Huang et al. (2018). The data processing and model are similar to the together tutorial, but with a few extra bits like aperture selection and de-trending.

To start, we need to download the target pixel file:

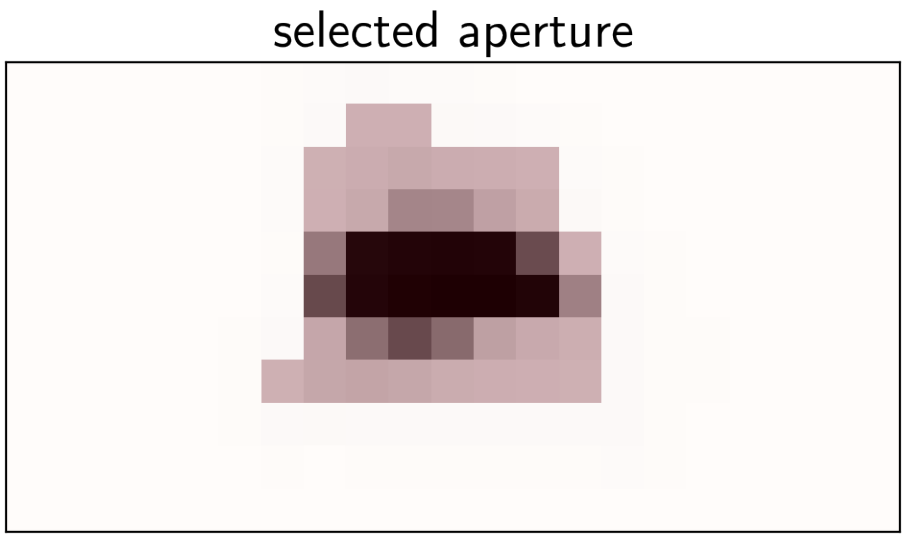
```
import numpy as np
from astropy.io import fits
import matplotlib.pyplot as plt

tpf_url = "https://archive.stsci.edu/missions/tess/tid/s0001/0000/0002/6113/6679/
↪tess2018206045859-s0001-0000000261136679-0120-s_tp.fits"
with fits.open(tpf_url) as hdus:
    tpf = hdus[1].data
    tpf_hdr = hdus[1].header

texp = tpf_hdr["FRAMETIM"] * tpf_hdr["NUM_FRM"]
texp /= 60.0 * 60.0 * 24.0
time = tpf["TIME"]
flux = tpf["FLUX"]
m = np.any(np.isfinite(flux), axis=(1, 2)) & (tpf["QUALITY"] == 0)
ref_time = 0.5 * (np.min(time[m])+np.max(time[m]))
time = np.ascontiguousarray(time[m] - ref_time, dtype=np.float64)
flux = np.ascontiguousarray(flux[m], dtype=np.float64)

mean_img = np.median(flux, axis=0)
plt.imshow(mean_img.T, cmap="gray_r")
plt.title("TESS image of Pi Men")
plt.xticks([])
plt.yticks([]);
```



2.8.1 Aperture selection

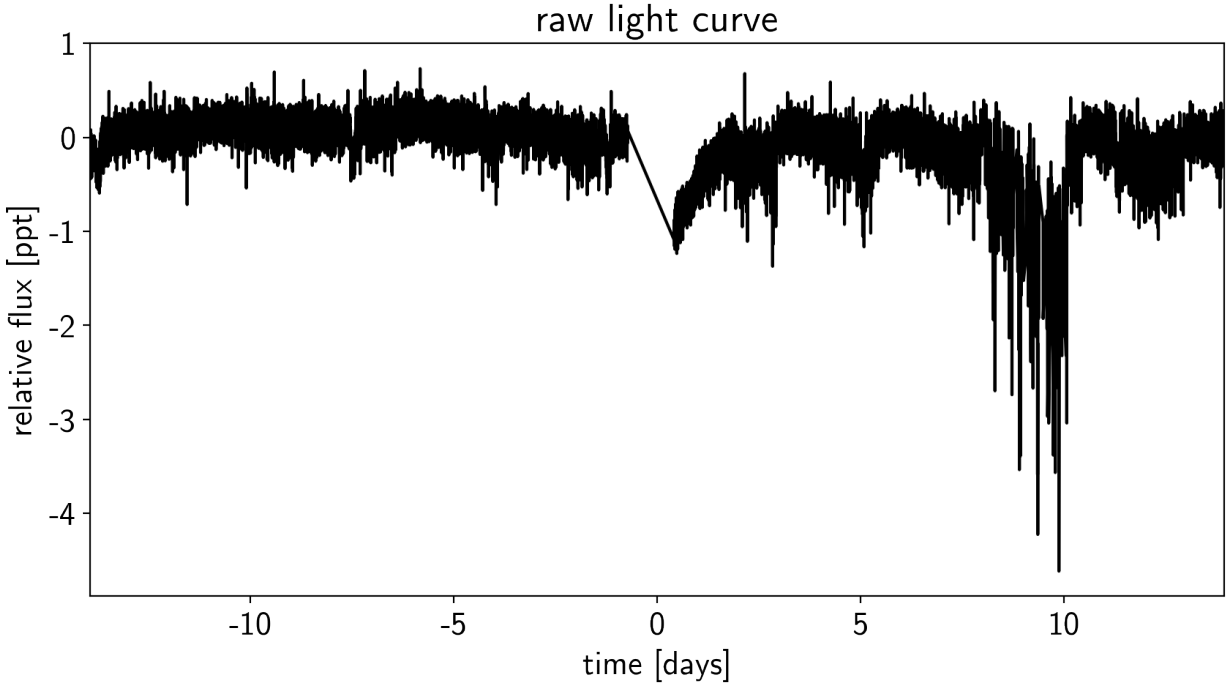
Next, we'll select an aperture using a hacky method that tries to minimizes the windowed scatter in the lightcurve (something like the CDPP).

```
from scipy.signal import savgol_filter

# Sort the pixels by median brightness
order = np.argsort(mean_img.flatten())[::-1]

# A function to estimate the windowed scatter in a lightcurve
def estimate_scatter_with_mask(mask):
    f = np.sum(flux[:, mask], axis=-1)
    smooth = savgol_filter(f, 1001, polyorder=5)
    return 1e6 * np.sqrt(np.median((f / smooth - 1)**2))

# Loop over pixels ordered by brightness and add them one-by-one
# to the aperture
masks, scatters = [], []
for i in range(10, 100):
    msk = np.zeros_like(mean_img, dtype=bool)
    msk[np.unravel_index(order[:i], mean_img.shape)] = True
    scatter = estimate_scatter_with_mask(msk)
    masks.append(msk)
    scatters.append(scatter)

# Choose the aperture that minimizes the scatter
pix_mask = masks[np.argmin(scatters)]

# Plot the selected aperture
plt.imshow(mean_img.T, cmap="gray_r")
plt.imshow(pix_mask.T, cmap="Reds", alpha=0.3)
plt.title("selected aperture")
plt.xticks([])
plt.yticks([]);
```



This aperture produces the following light curve:

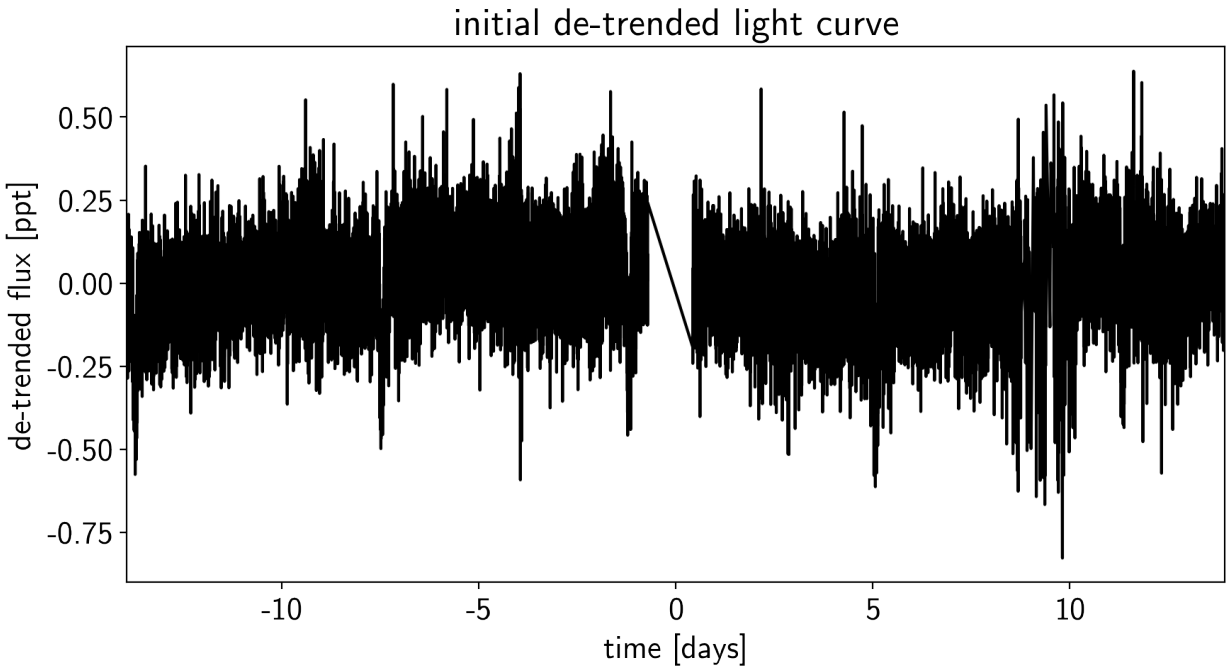
```
plt.figure(figsize=(10, 5))
sap_flux = np.sum(flux[:, pix_mask], axis=-1)
sap_flux = (sap_flux / np.median(sap_flux) - 1) * 1e3
plt.plot(time, sap_flux, "k")
plt.xlabel("time [days]")
plt.ylabel("relative flux [ppt]")
plt.title("raw light curve")
plt.xlim(time.min(), time.max());
```



2.8.2 De-trending

This doesn't look terrible, but we're still going to want to de-trend it a little bit. We'll use "pixel-level deconvolution" (PLD) to de-trend following the method used by Everest. Specifically, we'll use first order PLD plus the top few PCA components of the second order PLD basis.

```
# Build the first order PLD basis
X_pld = np.reshape(flux[:, pix_mask], (len(flux), -1))
X_pld = X_pld / np.sum(flux[:, pix_mask], axis=-1)[:, None]

# Build the second order PLD basis and run PCA to reduce the number of dimensions
X2_pld = np.reshape(X_pld[:, None, :] * X_pld[:, :, None], (len(flux), -1))
U, _, _ = np.linalg.svd(X2_pld, full_matrices=False)
X2_pld = U[:, :X_pld.shape[1]]

# Construct the design matrix and fit for the PLD model
X_pld = np.concatenate((np.ones((len(flux), 1)), X_pld, X2_pld), axis=-1)
XTX = np.dot(X_pld.T, X_pld)
w_pld = np.linalg.solve(XTX, np.dot(X_pld.T, sap_flux))
pld_flux = np.dot(X_pld, w_pld)

# Plot the de-trended light curve
plt.figure(figsize=(10, 5))
plt.plot(time, sap_flux-pld_flux, "k")
plt.xlabel("time [days]")
plt.ylabel("de-trended flux [ppt]")
plt.title("initial de-trended light curve")
plt.xlim(time.min(), time.max());
```

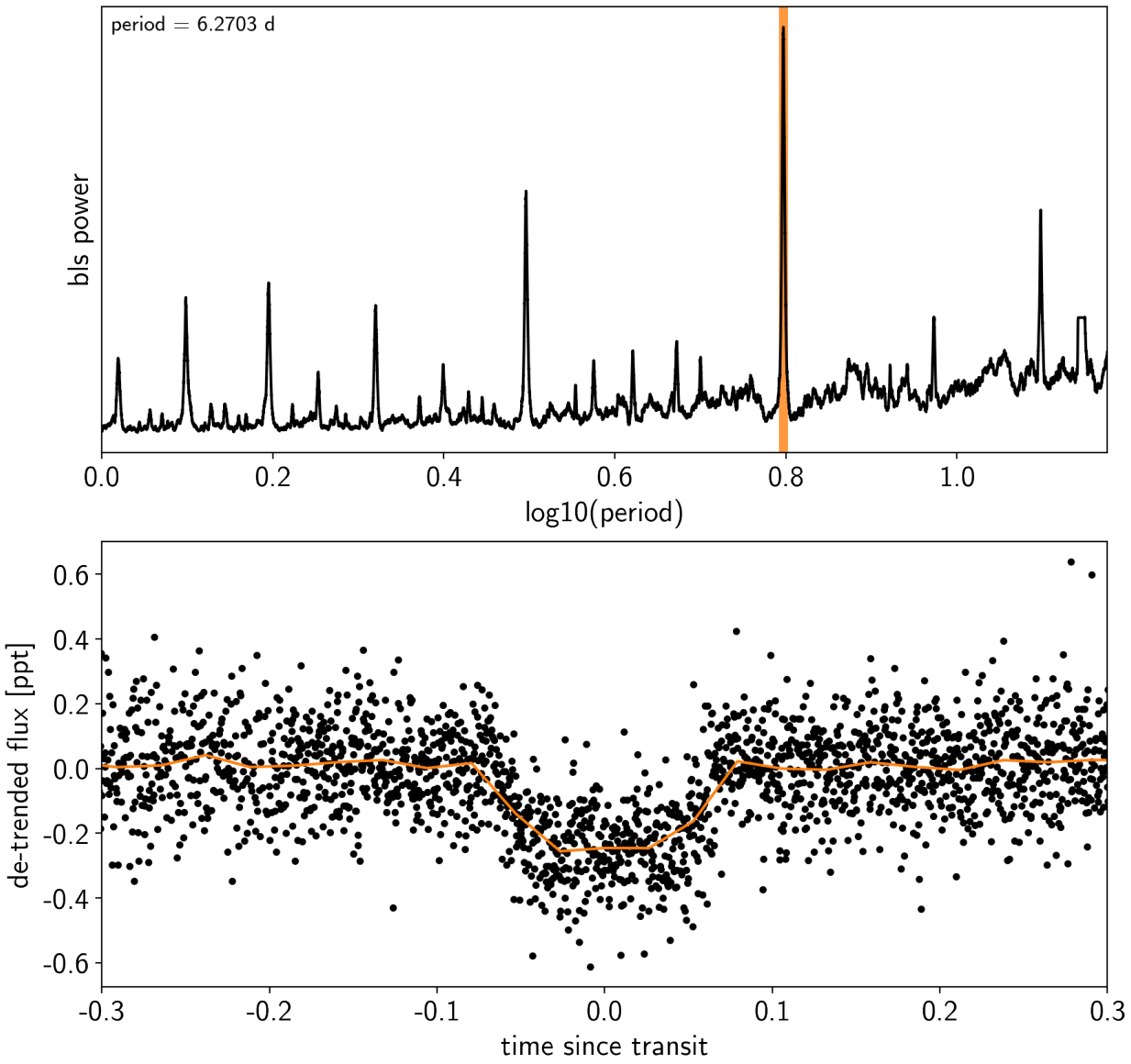


That looks better.

2.8.3 Transit search

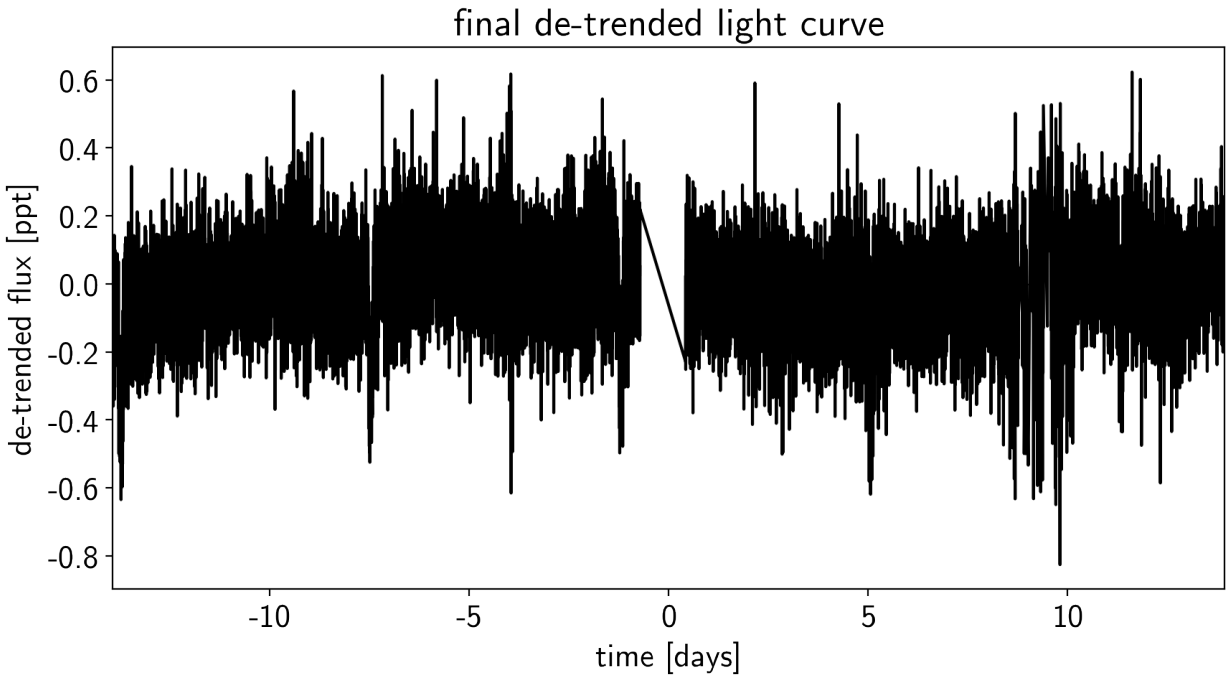
Now, let's use the box least squares periodogram from AstroPy (Note: you'll need AstroPy v3.1 or more recent to use this feature) to estimate the period, phase, and depth of the transit.

```
from astropy.stats import BoxLeastSquares

period_grid = np.exp(np.linspace(np.log(1), np.log(15), 50000))

bls = BoxLeastSquares(time, sap_flux - pld_flux)
bls_power = bls.power(period_grid, 0.1, oversample=20)

# Save the highest peak as the planet candidate
index = np.argmax(bls_power.power)
bls_period = bls_power.period[index]
bls_t0 = bls_power.transit_time[index]
bls_depth = bls_power.depth[index]
transit_mask = bls.transit_mask(time, bls_period, 0.2, bls_t0)

fig, axes = plt.subplots(2, 1, figsize=(10, 10))

# Plot the periodogram
ax = axes[0]
ax.axvline(np.log10(bls_period), color="C1", lw=5, alpha=0.8)
ax.plot(np.log10(bls_power.period), bls_power.power, "k")
ax.annotate("period = {0:.4f} d".format(bls_period),
            (0, 1), xycoords="axes fraction",
            xytext=(5, -5), textcoords="offset points",
            va="top", ha="left", fontsize=12)
ax.set_ylabel("bls power")
ax.set_yticks([])
ax.set_xlim(np.log10(period_grid.min()), np.log10(period_grid.max()))
ax.set_xlabel("log10(period)")

# Plot the folded transit
ax = axes[1]
x_fold = (time - bls_t0 + 0.5*bls_period)%bls_period - 0.5*bls_period
m = np.abs(x_fold) < 0.4
ax.plot(x_fold[m], sap_flux[m] - pld_flux[m], ".k")

# Overplot the phase binned light curve
bins = np.linspace(-0.41, 0.41, 32)
denom, _ = np.histogram(x_fold, bins)
num, _ = np.histogram(x_fold, bins, weights=sap_flux - pld_flux)
denom[num == 0] = 1.0
ax.plot(0.5*(bins[1:] + bins[:-1]), num / denom, color="C1")

ax.set_xlim(-0.3, 0.3)
ax.set_ylabel("de-trended flux [ppt]")
ax.set_xlabel("time since transit");
```



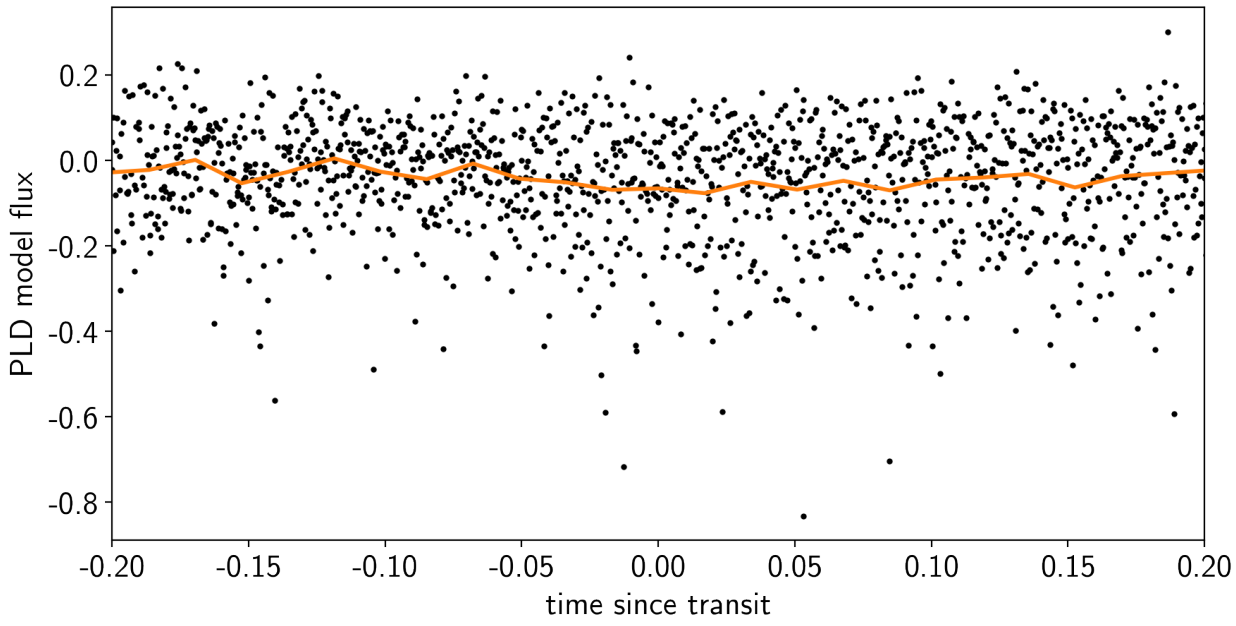
Now that we know where the transits are, it's generally good practice to de-trend the data one more time with the transits masked so that the de-trending doesn't overfit the transits. Let's do that.

```
m = ~transit_mask
XTX = np.dot(X_pld[m].T, X_pld[m])
w_pld = np.linalg.solve(XTX, np.dot(X_pld[m].T, sap_flux[m]))
pld_flux = np.dot(X_pld, w_pld)

x = np.ascontiguousarray(time, dtype=np.float64)
y = np.ascontiguousarray(sap_flux-pld_flux, dtype=np.float64)

plt.figure(figsize=(10, 5))
plt.plot(time, y, "k")
plt.xlabel("time [days]")
plt.ylabel("de-trended flux [ppt]")
plt.title("final de-trended light curve")
plt.xlim(time.min(), time.max());
```



To confirm that we didn't overfit the transit, we can look at the folded light curve for the PLD model near trasit. This shouldn't have any residual transit signal, and that looks correct here:

```
plt.figure(figsize=(10, 5))

x_fold = (x - bls_t0 + 0.5*bls_period) % bls_period - 0.5*bls_period
m = np.abs(x_fold) < 0.3
plt.plot(x_fold[m], pld_flux[m], ".k", ms=4)

bins = np.linspace(-0.5, 0.5, 60)
denom, _ = np.histogram(x_fold, bins)
num, _ = np.histogram(x_fold, bins, weights=pld_flux)
denom[num == 0] = 1.0
plt.plot(0.5*(bins[1:] + bins[:-1]), num / denom, color="C1", lw=2)
plt.xlim(-0.2, 0.2)
plt.xlabel("time since transit")
plt.ylabel("PLD model flux");
```



2.8.4 The transit model in PyMC3

The transit model, initialization, and sampling are all nearly the same as the one in together, but we'll use a more informative prior on eccentricity.

```
import exoplanet as xo
import pymc3 as pm
import theano.tensor as tt

def build_model(mask=None, start=None):
    if mask is None:
        mask = np.ones(len(x), dtype=bool)
    with pm.Model() as model:

        # Parameters for the stellar properties
        mean = pm.Normal("mean", mu=0.0, sd=10.0)
        u_star = xo.distributions.QuadLimbDark("u_star")

        # Stellar parameters from Huang et al (2018)
        M_star_huang = 1.094, 0.039
        R_star_huang = 1.10, 0.023
        m_star = pm.Normal("m_star", mu=M_star_huang[0], sd=M_star_huang[1])
        r_star = pm.Normal("r_star", mu=R_star_huang[0], sd=R_star_huang[1])

        # Prior to require physical parameters
        pm.Potential("m_star_prior", tt.switch(m_star > 0, 0, -np.inf))
        pm.Potential("r_star_prior", tt.switch(r_star > 0, 0, -np.inf))

        # Orbital parameters for the planets
        logP = pm.Normal("logP", mu=np.log(bls_period), sd=1)
        t0 = pm.Normal("t0", mu=bls_t0, sd=1)
        b = pm.Uniform("b", lower=0, upper=1, testval=0.5)
        logr = pm.Normal("logr", sd=1.0,
                         mu=0.5*np.log(1e-3*np.array(bls_depth))+np.log(R_star_
↪huang[0]))
```

(continues on next page)

(continued from previous page)

```
        r_pl = pm.Deterministic("r_pl", tt.exp(logr))
        ror = pm.Deterministic("ror", r_pl / r_star)

        # This is the eccentricity prior from Kipping (2013):
        # https://arxiv.org/abs/1306.4982
        ecc = pm.Beta("ecc", alpha=0.867, beta=3.03, testval=0.1)
        omega = xo.distributions.Angle("omega")

        # Transit jitter & GP parameters
        logs2 = pm.Normal("logs2", mu=np.log(np.var(y[mask])), sd=10)
        logw0_guess = np.log(2*np.pi/10)
        logw0 = pm.Normal("logw0", mu=logw0_guess, sd=10)

        # We'll parameterize using the maximum power (S_0 * w_0^4) instead of
        # S_0 directly because this removes some of the degeneracies between
        # S_0 and omega_0
        logpower = pm.Normal("logpower",
                             mu=np.log(np.var(y[mask]))+4*logw0_guess,
                             sd=10)
        logS0 = pm.Deterministic("logS0", logpower - 4 * logw0)

        # Tracking planet parameters
        period = pm.Deterministic("period", tt.exp(logP))

        # Orbit model
        orbit = xo.orbits.KeplerianOrbit(
            r_star=r_star, m_star=m_star,
            period=period, t0=t0, b=b,
            ecc=ecc, omega=omega)

        # Compute the model light curve using starry
        light_curves = xo.StarryLightCurve(u_star).get_light_curve(
            orbit=orbit, r=r_pl, t=x[mask], texp=texp)*1e3
        light_curve = pm.math.sum(light_curves, axis=-1) + mean
        pm.Deterministic("light_curves", light_curves)

        # GP model for the light curve
        kernel = xo.gp.terms.SHOTerm(log_S0=logS0, log_w0=logw0, Q=1/np.sqrt(2))
        gp = xo.gp.GP(kernel, x[mask], tt.exp(logs2) + tt.zeros(mask.sum()), J=2)
        pm.Potential("transit_obs", gp.log_likelihood(y[mask] - light_curve))
        pm.Deterministic("gp_pred", gp.predict())

        # Fit for the maximum a posteriori parameters, I've found that I can get
        # a better solution by trying different combinations of parameters in turn
        if start is None:
            start = model.test_point
        map_soln = xo.optimize(start=start, vars=[logs2, logpower, logw0])
        map_soln = xo.optimize(start=map_soln, vars=[logr])
        map_soln = xo.optimize(start=map_soln, vars=[b])
        map_soln = xo.optimize(start=map_soln, vars=[logP, t0])
        map_soln = xo.optimize(start=map_soln, vars=[u_star])
        map_soln = xo.optimize(start=map_soln, vars=[logr])
        map_soln = xo.optimize(start=map_soln, vars=[b])
        map_soln = xo.optimize(start=map_soln, vars=[ecc, omega])
        map_soln = xo.optimize(start=map_soln, vars=[mean])
        map_soln = xo.optimize(start=map_soln, vars=[logs2, logpower, logw0])
        map_soln = xo.optimize(start=map_soln)
```

(continues on next page)

(continued from previous page)

```
    return model, map_soln

model0, map_soln0 = build_model()
```

```
optimizing logp for variables: ['logw0', 'logpower', 'logs2']
message: Optimization terminated successfully.
logp: 12740.765561069971 -> 13007.915373837846
optimizing logp for variables: ['logr']
message: Optimization terminated successfully.
logp: 13007.915373837846 -> 13009.375296936547
optimizing logp for variables: ['b_interval__']
message: Optimization terminated successfully.
logp: 13009.37529693655 -> 13011.964145594648
optimizing logp for variables: ['t0', 'logP']
message: Optimization terminated successfully.
logp: 13011.964145594648 -> 13025.549814395878
optimizing logp for variables: ['u_star_quadlimbdark__']
message: Optimization terminated successfully.
logp: 13025.54981439587 -> 13028.711073192928
optimizing logp for variables: ['logr']
message: Desired error not necessarily achieved due to precision loss.
logp: 13028.711073192928 -> 13028.922206186133
optimizing logp for variables: ['b_interval__']
message: Optimization terminated successfully.
logp: 13028.922206186126 -> 13029.259565820235
optimizing logp for variables: ['omega_angle__', 'ecc_logodds__']
message: Optimization terminated successfully.
logp: 13029.259565820235 -> 13049.868148593856
optimizing logp for variables: ['mean']
message: Optimization terminated successfully.
logp: 13049.86814859386 -> 13049.889405526703
optimizing logp for variables: ['logw0', 'logpower', 'logs2']
message: Optimization terminated successfully.
logp: 13049.889405526703 -> 13049.918411594184
optimizing logp for variables: ['logpower', 'logw0', 'logs2', 'omega_angle__', 'ecc_
↪logodds__', 'logr', 'b_interval__', 't0', 'logP', 'r_star', 'm_star', 'u_star_
↪quadlimbdark__', 'mean']
message: Desired error not necessarily achieved due to precision loss.
logp: 13049.918411594203 -> 13052.459974919158
```

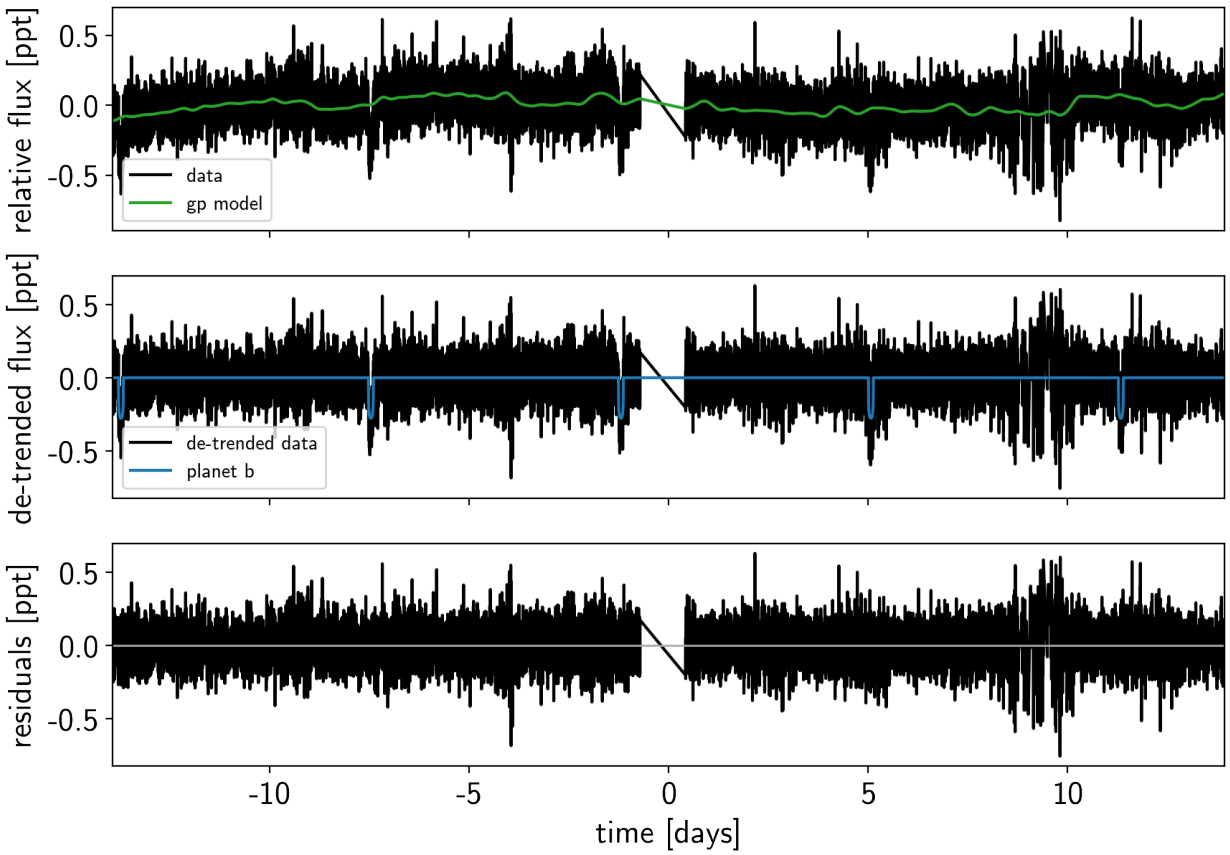
Here's how we plot the initial light curve model:

```
def plot_light_curve(soln, mask=None):
    if mask is None:
        mask = np.ones(len(x), dtype=bool)

    fig, axes = plt.subplots(3, 1, figsize=(10, 7), sharex=True)

    ax = axes[0]
    ax.plot(x[mask], y[mask], "k", label="data")
    gp_mod = soln["gp_pred"] + soln["mean"]
    ax.plot(x[mask], gp_mod, color="C2", label="gp model")
    ax.legend(fontsize=10)
    ax.set_ylabel("relative flux [ppt]")
```

(continues on next page)

(continued from previous page)

```
    ax = axes[1]
    ax.plot(x[mask], y[mask] - gp_mod, "k", label="de-trended data")
    for i, l in enumerate("b"):
        mod = soln["light_curves"][:, i]
        ax.plot(x[mask], mod, label="planet {0}".format(l))
    ax.legend(fontsize=10, loc=3)
    ax.set_ylabel("de-trended flux [ppt]")

    ax = axes[2]
    mod = gp_mod + np.sum(soln["light_curves"], axis=-1)
    ax.plot(x[mask], y[mask] - mod, "k")
    ax.axhline(0, color="#aaaaaa", lw=1)
    ax.set_ylabel("residuals [ppt]")
    ax.set_xlim(x[mask].min(), x[mask].max())
    ax.set_xlabel("time [days]")

    return fig

plot_light_curve(map_soln0);
```



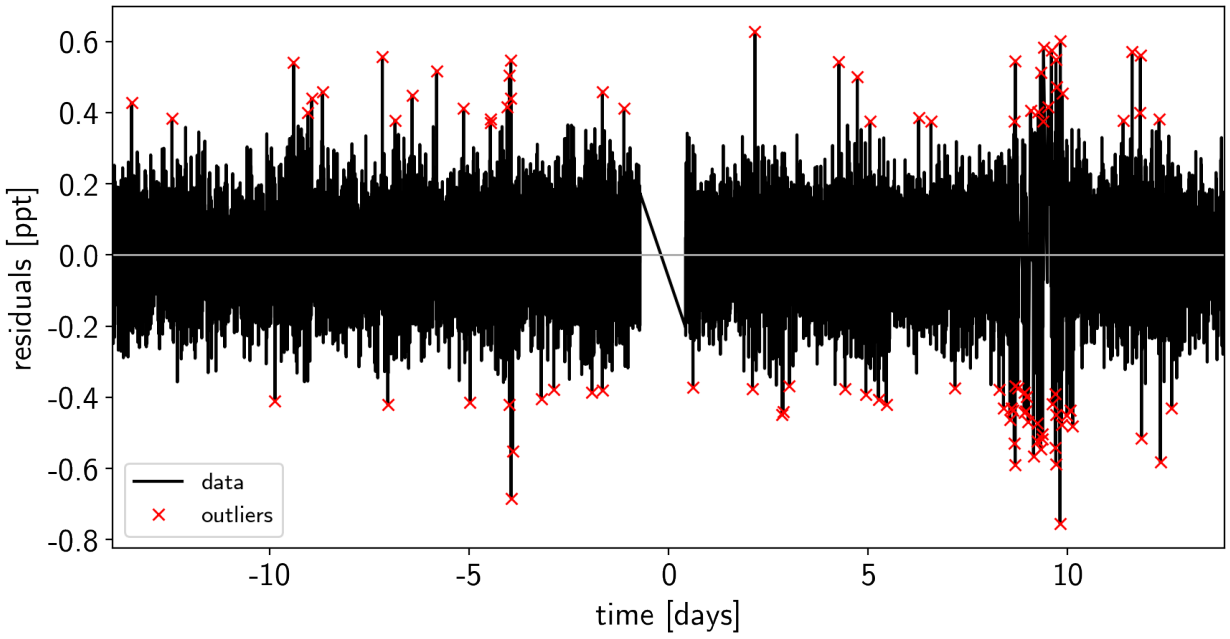
As in the together tutorial, we can do some sigma clipping to remove significant outliers.

```
mod = map_soln0["gp_pred"] + map_soln0["mean"] + np.sum(map_soln0["light_curves"],␣
↪axis=-1)
resid = y - mod
rms = np.sqrt(np.median(resid**2))
```

(continues on next page)

(continued from previous page)

```
mask = np.abs(resid) < 5 * rms

plt.figure(figsize=(10, 5))
plt.plot(x, resid, "k", label="data")
plt.plot(x[~mask], resid[~mask], "xr", label="outliers")
plt.axhline(0, color="#aaaaaa", lw=1)
plt.ylabel("residuals [ppt]")
plt.xlabel("time [days]")
plt.legend(fontsize=12, loc=3)
plt.xlim(x.min(), x.max());
```



And then we re-build the model using the data without outliers.

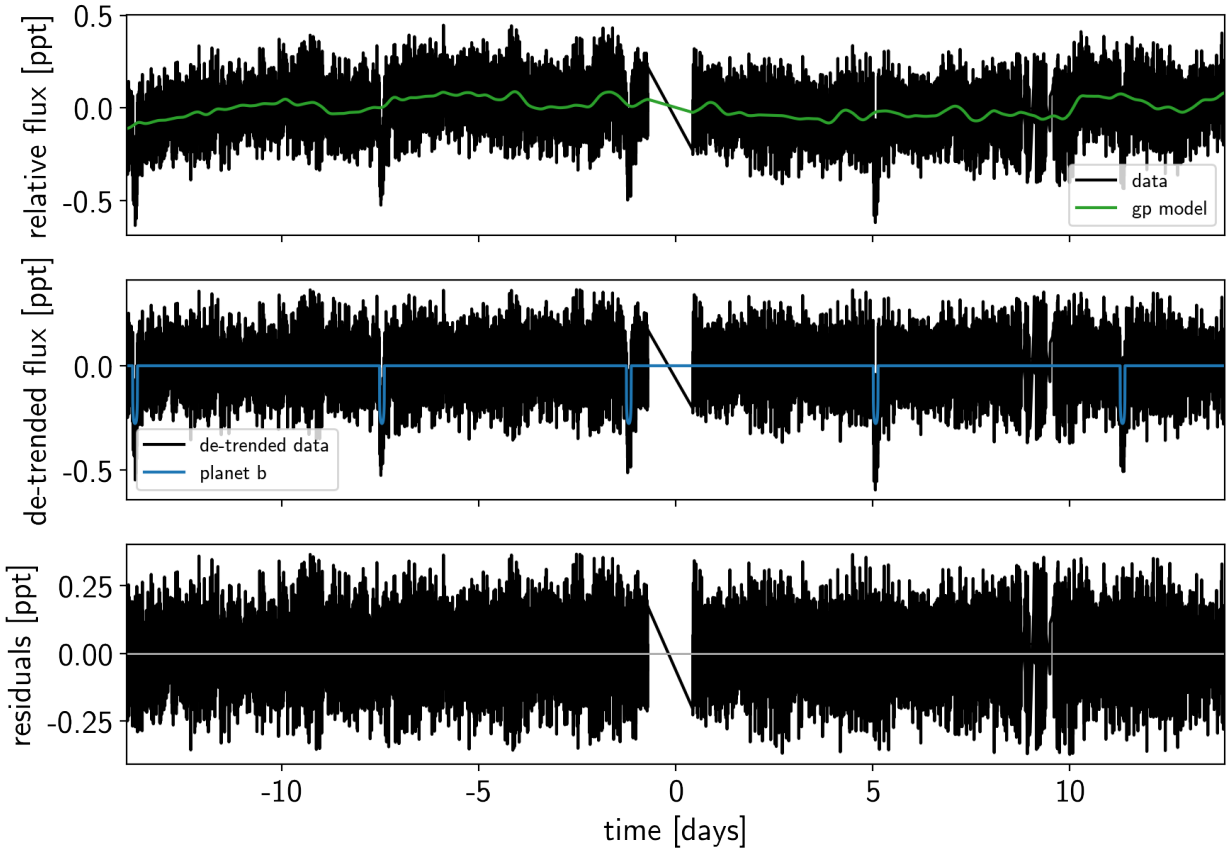
```
model, map_soln = build_model(mask, map_soln0)
plot_light_curve(map_soln, mask);
```

```
optimizing logp for variables: ['logw0', 'logpower', 'logs2']
message: Optimization terminated successfully.
logp: 13706.965171929465 -> 13737.924647263992
optimizing logp for variables: ['logr']
message: Optimization terminated successfully.
logp: 13737.924647263992 -> 13737.944799031835
optimizing logp for variables: ['b_interval__']
message: Optimization terminated successfully.
logp: 13737.944799031839 -> 13737.945672047628
optimizing logp for variables: ['t0', 'logP']
message: Desired error not necessarily achieved due to precision loss.
logp: 13737.945672047628 -> 13737.954508551973
optimizing logp for variables: ['u_star_quadlimbdark__']
message: Optimization terminated successfully.
logp: 13737.954508551977 -> 13737.978356987767
optimizing logp for variables: ['logr']
message: Optimization terminated successfully.
```

(continues on next page)

(continued from previous page)

```
logp: 13737.978356987767 -> 13737.981517082586
optimizing logp for variables: ['b_interval__']
message: Optimization terminated successfully.
logp: 13737.981517082586 -> 13737.990354096222
optimizing logp for variables: ['omega_angle__', 'ecc_logodds__']
message: Optimization terminated successfully.
logp: 13737.990354096222 -> 13737.990393687394
optimizing logp for variables: ['mean']
message: Optimization terminated successfully.
logp: 13737.990393687402 -> 13737.993544839675
optimizing logp for variables: ['logw0', 'logpower', 'logs2']
message: Optimization terminated successfully.
logp: 13737.993544839675 -> 13737.993547696959
optimizing logp for variables: ['logpower', 'logw0', 'logs2', 'omega_angle__', 'ecc_
↪logodds__', 'logr', 'b_interval__', 't0', 'logP', 'r_star', 'm_star', 'u_star_
↪quadlimbdark__', 'mean']
message: Desired error not necessarily achieved due to precision loss.
logp: 13737.993547696966 -> 13738.00534645991
```



Now that we have the model, we can sample it using a *exoplanet.PyMC3Sampler*:

```
np.random.seed(42)
sampler = xo.PyMC3Sampler(window=100, start=200, finish=300)
with model:
    burnin = sampler.tune(tune=3500, start=map_soln,
                          step_kwargs=dict(target_accept=0.9),
```

(continues on next page)

(continued from previous page)

```
                            chains=4)
```

```
Sampling 4 chains: 100%|| 808/808 [06:48<00:00,  2.37s/draws]
Sampling 4 chains: 100%|| 408/408 [03:10<00:00,  1.43s/draws]
Sampling 4 chains: 100%|| 808/808 [06:46<00:00,  4.98s/draws]
Sampling 4 chains: 100%|| 1608/1608 [19:31<00:00,  1.35s/draws]
Sampling 4 chains: 100%|| 3208/3208 [46:32<00:00,  2.57s/draws]
Sampling 4 chains: 100%|| 7208/7208 [1:31:00<00:00,  1.29s/draws]
Sampling 4 chains: 100%|| 1208/1208 [11:13<00:00,  5.46s/draws]
```

```
with model:
    trace = sampler.sample(draws=2000, chains=4)
```

```
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [logpower, logw0, logs2, omega, ecc, logr, b, t0, logP, r_star, m_star,␣
↪u_star, mean]
Sampling 4 chains: 100%|| 8000/8000 [1:06:42<00:00,  1.94s/draws]
There were 2 divergences after tuning. Increase target_accept or␣
↪reparameterize.
There was 1 divergence after tuning. Increase target_accept or reparameterize.
There were 2 divergences after tuning. Increase target_accept or␣
↪reparameterize.
The number of effective samples is smaller than 25% for some parameters.
```

```
pm.summary(trace, varnames=["logw0", "logpower", "logs2", "omega", "ecc", "r_pl", "b",
↪ "t0", "logP", "r_star", "m_star", "u_star", "mean"])
```

2.8.5 Results

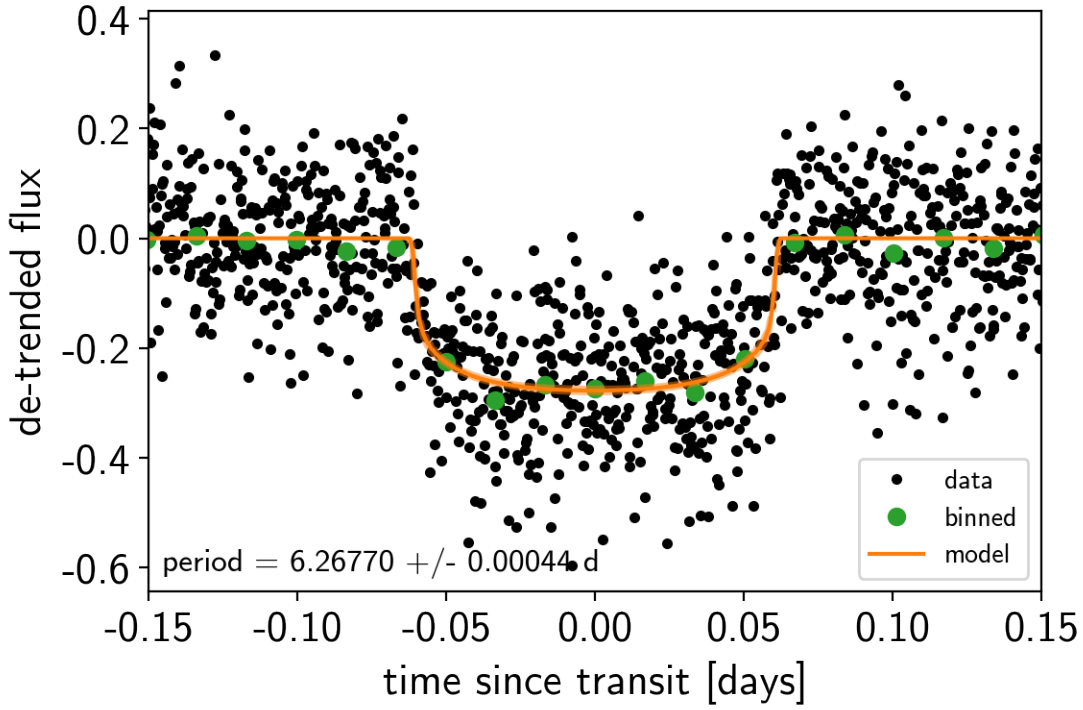
After sampling, we can make the usual plots. First, let's look at the folded light curve plot:

```
# Compute the GP prediction
gp_mod = np.median(trace["gp_pred"] + trace["mean"][:, None], axis=0)

# Get the posterior median orbital parameters
p = np.median(trace["period"])
t0 = np.median(trace["t0"])

# Plot the folded data
x_fold = (x[mask] - t0 + 0.5*p) % p - 0.5*p
plt.plot(x_fold, y[mask] - gp_mod, ".k", label="data", zorder=-1000)

# Overplot the phase binned light curve
bins = np.linspace(-0.41, 0.41, 50)
denom, _ = np.histogram(x_fold, bins)
num, _ = np.histogram(x_fold, bins, weights=y[mask])
denom[num == 0] = 1.0
plt.plot(0.5*(bins[1:] + bins[:-1]), num / denom, "o", color="C2",
         label="binned")

# Plot the folded model
inds = np.argsort(x_fold)
inds = inds[np.abs(x_fold)[inds] < 0.3]
```

(continues on next page)

(continued from previous page)

```
pred = trace["light_curves"][:, inds, 0]
pred = np.percentile(pred, [16, 50, 84], axis=0)
plt.plot(x_fold[inds], pred[1], color="C1", label="model")
art = plt.fill_between(x_fold[inds], pred[0], pred[2], color="C1", alpha=0.5,
                       zorder=1000)
art.set_edgecolor("none")

# Annotate the plot with the planet's period
txt = "period = {0:.5f} +/- {1:.5f} d".format(
    np.mean(trace["period"]), np.std(trace["period"]))
plt.annotate(txt, (0, 0), xycoords="axes fraction",
             xytext=(5, 5), textcoords="offset points",
             ha="left", va="bottom", fontsize=12)

plt.legend(fontsize=10, loc=4)
plt.xlim(-0.5*p, 0.5*p)
plt.xlabel("time since transit [days]")
plt.ylabel("de-trended flux")
plt.xlim(-0.15, 0.15);
```



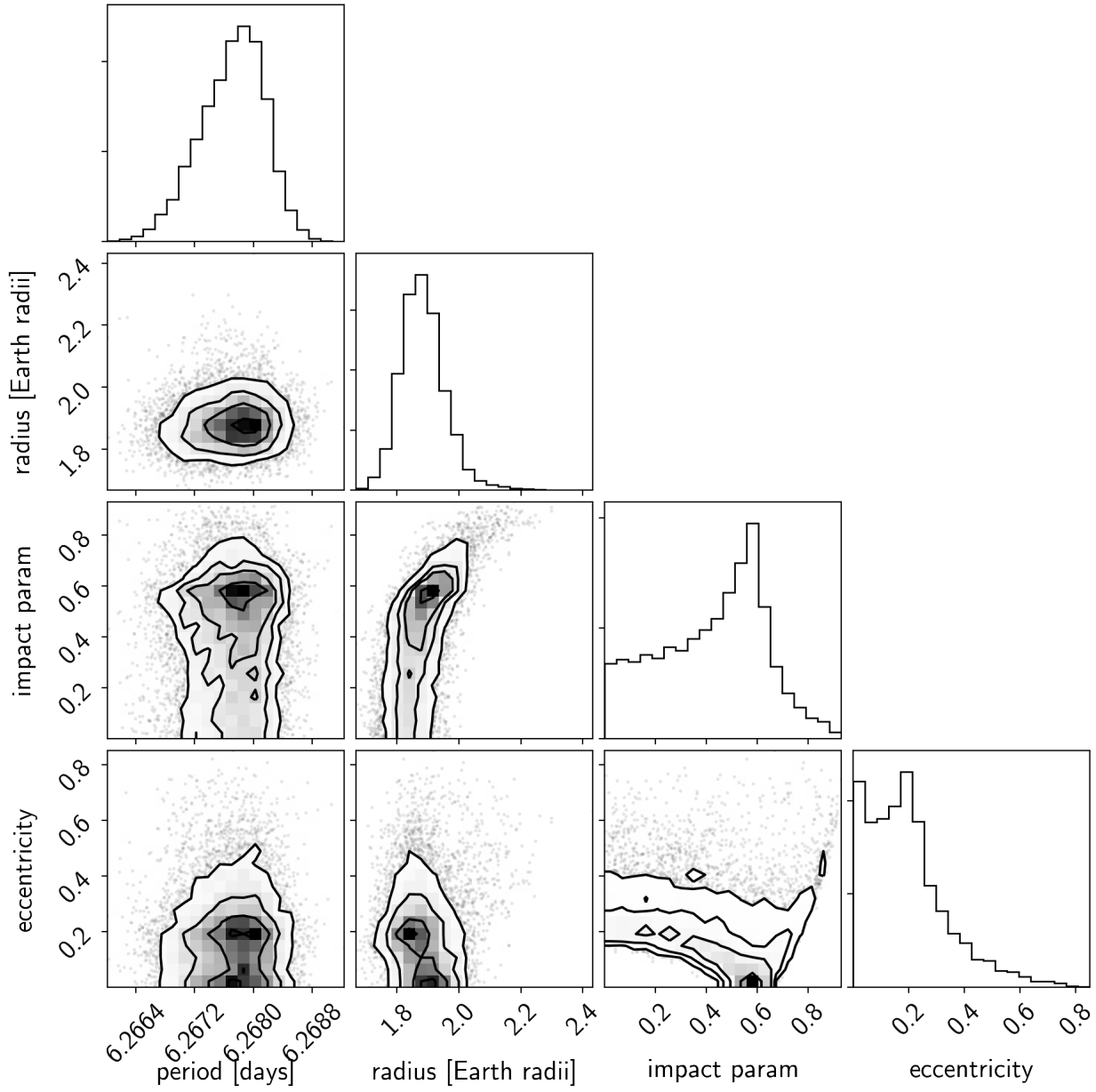
And a corner plot of some of the key parameters:

```
import corner
import astropy.units as u
varnames = ["period", "b", "ecc", "r_pl"]
samples = pm.trace_to_dataframe(trace, varnames=varnames)

# Convert the radius to Earth radii
samples["r_pl"] = (np.array(samples["r_pl"]) * u.R_sun).to(u.R_earth).value

corner.corner(
```

(continues on next page)

(continued from previous page)

```
    samples[["period", "r_pl", "b", "ecc"]],
    labels=["period [days]", "radius [Earth radii]", "impact param", "eccentricity"]);
```



These all seem consistent with the previously published values and an earlier inconsistency between this radius measurement and the literature has been resolved by fixing a bug in *exoplanet*.

CHAPTER 3

License & attribution

If you make use of this code, please cite this package and its dependencies. You can find more information about how and what to cite in the citation documentation.

CHAPTER 4

Changelog

4.1 0.1.5 (2019-03-07)

- Improves contact point solver using companion matrix to solve quadratic
- Improves reliability of `Angle` distribution when the value of the angle is well constrained

4.2 0.1.4 (2019-02-10)

- Improves the reliability of the PyMC3Sampler
- Adds a new `optimize` function since the `find_MAP` method in PyMC3 is deprecated
- Adds cronjob script for automatically updating tutorials.

4.3 0.1.3 (2019-01-09)

- Adds a more robust and faster Kepler solver (ref)
- Fixes minor behavioral bugs in PyMC3 sampler wrapper

4.4 0.1.2 (2018-12-13)

- Adds regular grid interpolation Op for Theano
- Fixes major bug in handling of the stellar radius for transits
- Fixes small bugs in packaging and installation
- Fixes handling of diagonal covariances in `PyMC3Sampler`

4.5 0.1.1 (IPO; 2018-12-06)

- Initial public release

Python Module Index

Index

R

S

T

U

W