PyKE: Open source Python data analysis for NASA's Kepler, K2, and TESS missions

PyKE Contributors

January 20, 2018

Abstract

1 Introduction

2 Lightcurve Basics

2.1 The `LightCurve` class

A `LightCurve` object can be instantiated by passing a `time` array, a `flux` array, and optionally a `flux_err` array which accounts for uncertainties in the `flux` measurements, i.e.,

```
from pyke.lightcurve import LightCurve
lc = LightCurve(time, flux)
```

`LightCurve` object provides methods described in Table (1)

The `KeplerLightCurve` class extends `LightCurve` by adding attributes to store metadata information such as channel number, quality flags, campaign or quarter number, kepler id, etc.

Additionally, `KeplerLightCurve` can be corrected for motion-dependent correlated noise using the `correct` method which will be discussed in Section 4.3.

2.2 The `KeplerLightCurveFile` class

The `KeplerLightCurveFile` class defines a structure to deal with lightcurve files from both NASA's Kepler and K2 missions.

To instantiate a `KeplerLightCurveFile` object, it is necessary to pass a `path` which represents the address (url or local path) of a lightcurve file in the fits (or compressed) format, and a `quality_bitmask` string which specifies quality flags of cadences that should be ignored.

Table 1: A subset of methods provided by the `LightCurve` class

Method signature	Short description
`stitch`	appends the attributes `flux`, `time`, and `flux_err` of other given `LighCurve` objects.
`flatten`	applies a Savitzky-Golay filter to capture low frequency flux variations which can be then removed in order to aid transit detection algorithms.
`fold`	folds a lightcurve at a given period and phase.
`bin`	bins a lightcurve using a block mean or median.
`cdpp`	computes the Combined Differential Photometric Precision (CDPP) metric, which is a proxy for the amount of scatter in the lightcurve signal.
`plot`	displays a lightcurve.

One crucial method of the `KeplerLightCurveFile` class is `get_lightcurve` which returns a `KeplerLightCurve` object with the metadata provided by the corresponding `KeplerLightCurveFile`.

Therefore, one can, for example, perform the following series of operations in order to fold a lightcurve from the MAST archive

```
>>> lc_file = KeplerLightCurveFile("https://archive.stsci.edu/missions/kepler/"
... "lightcurves/0119/011904151/kplr011904151-2009350155506_llc.fits")
>>> klc = lc_file.get_lightcurve("PDCSAP_FLUX").fold(period=10.8234)
>>> klc.plot()
```

3 Target Pixel File Basics

3.1 The `KeplerTargetPixelFile` class

A `KeplerTargetPixelFile` object can be instantiated by passing a `path` (URL or local) of a target pixel file. Optionally, the user the select to throw away frames that contain a specific flag by using the `quality_bitmask` argument.

`KeplerTargetPixelFile` offers a number of methods that range from getting raw aperture photometry lightcurves to data visualization.

For instance, the method `plot` can be used to visualize a given frame, which are depicted in Fig. **??**

```
import numpy as np
from pyke import KeplerTargetPixelFile
tpf = KeplerTargetPixelFile('kplr008462852-2011073133259_lpd-targ.fits.gz')
tpf.plot()
tpf.plot(aperture_mask=tpf.flux[0] > np.nanmean(tpf.flux[0]))
```

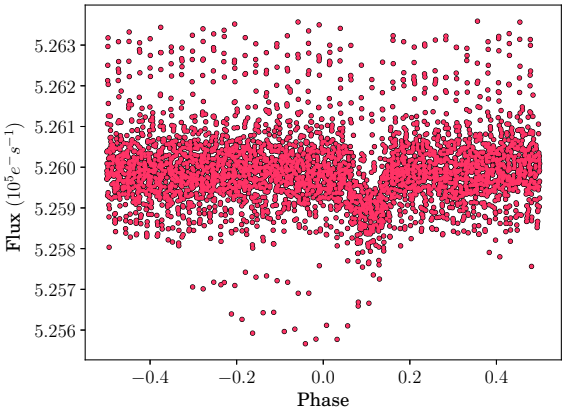

Figure 1: Folded lightcurve of target `KIC011904151` quarter 3, showing the transit signal of Kepler-10b.

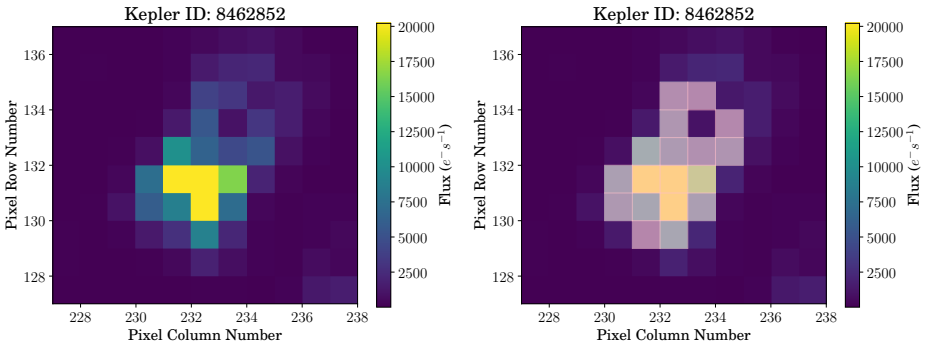

Figure 2:

4 Tools

4.1 Cotrending basis vectors

Cotrending basis vectors (CBVs) correction consists in removing global correlated systematics that occurs in a given channel [add reference]. The procedure of designing the CBVs is discussed in [add reference].

Briefly, given a set of n CBVs, one is interested in finding a set of n coefficients $\boldsymbol{\theta} = (\theta_1, \theta_2, ..., \theta_n)$ which minimizes some cost function between the SAP flux and the set of CBVs.

The following cost function is often used

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}\in\Theta} \sum_t |f_{SAP}(t) - \sum_{j=1}^{n} \theta_j v_j(t)|^p, p > 0, p \in \mathbb{R}, \tag{1}$$

in which f_{SAP} is the SAP flux and v_j is the j-th CBV.

Then, the cbv-corrected flux can be computed as

$$f_{CBV} = f_{SAP} - \sum_{j=1}^{n} \theta_j^* v_j(t). \tag{2}$$

A runnable example of SAP flux correction for target `KOI 8462852` can be written as follows

```
from pyke.lightcurve import KeplerCBVCorrector
cbv = KeplerCBVCorrector("https://archive.stsci.edu/missions/kepler/lightcurves/"
                         "0084/008462852/kplr008462852-2011073133259_llc.fits")
cbv_lc = cbv.correct(cbvs=[1,2])
```

As a result, Fig 3 illustrates the correction.

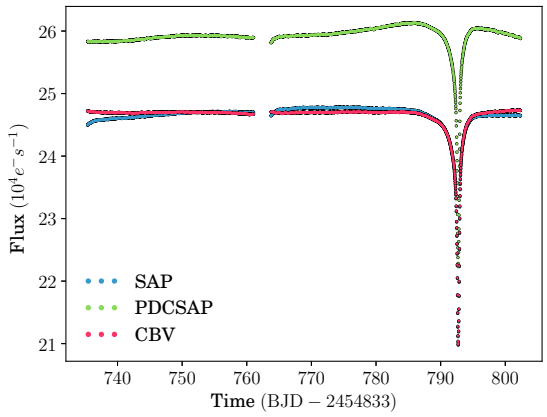

Figure 3: CBV correction applied on `KOI 8462852`

In this setting, the user is left to choose the cost function and the number of CBVs. It is often the case where $p = 1$ yields robust (non-overfitted) results. In fact, $p = 1$ defines the ℓ_1-norm which is robust against outliers [add reference].

The number of CBVs will directly contribute to overfitting effects. One way to identify a reasonable number of CBVs is to perform a grid search as suggested in Fig (4).

In the same fashion, we can apply cotrending basis vector correction to *K2* lightcurves.

For example, Fig. 5 shows the result after estimating the first nine coefficients for CBV correciton on target `EPIC 201543306`. The selection of the number of CBVs is also done empirically by inspecting the grid search curve.

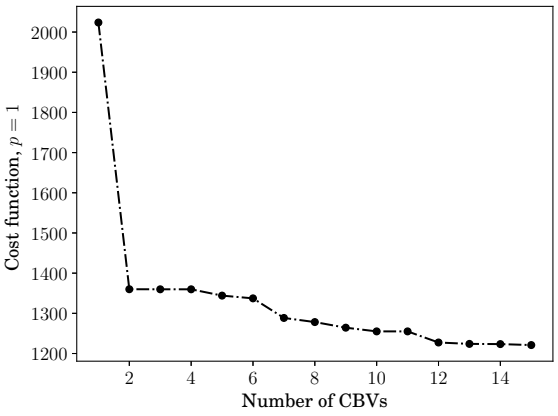

Figure 4: Grid search on the number of CBVs

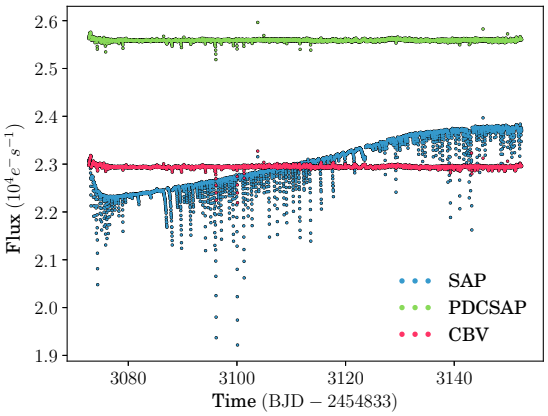

Figure 5: CBV correction applied on `EPIC 201543306`.

4.2 Point spread function photometry

PyKE contains routines to perform PSF photometry in TPFs which are implemented in the `psf` module.

Briefly, the PSF photometry problem that PyKE solves can be formulated as follows. Given an image $\boldsymbol{y}$, with n pixels and m stars, and a PSF model $\lambda(\boldsymbol{\theta}) = \sum_{j=1}^{m} \lambda(\theta_j)$, find the best parameter vector $\boldsymbol{\theta}^* = (\theta_1^*, \theta_2^*, ..., \theta_m^*)$ that minimizes some cost (or loss) function $R(\lambda(\boldsymbol{\theta}), \boldsymbol{y})$ of assigning $\boldsymbol{\theta} = \boldsymbol{\theta}^*$.

A runnable code to perform PSF photometry in `EPIC 246199087` (Trappist-1), can be written as follows:

```
from pyke import KeplerTargetPixelFile
from pyke.psf import PRFPhotometry, SceneModel
```

```
from oktopus import UniformPrior

tpf = KeplerTargetPixelFile("ktwo246199087-c12_lpd-targ.fits.gz")
prf = tpf.get_prf_model()
prior = UniformPrior(lb=[4e3, 990, 25, 1], ub=[2e4, 996, 30, 2e3])
scene = SceneModel(prf=[prfs])

phot = PRFPhotometry(scene_model=scene, prior=prior)
results = phot.fit(tpf.flux + tpf.flux_bkg)
```

The photometric results are stored in the $c \times 4$ matrix, where c is the number of frames (cadences).

From a probabilistic point of view, one is often interested in minimizing the expected cost with respect to some probability distribution assigned to the data $\boldsymbol{y}$ and to the parameter vector $\boldsymbol{\theta}$, from which the cost function R naturally arises. The default assumption on the data is that it follows a Poisson probability distribution; whereas the probability distribution on the parameter vector has to be assigned by the user using the `prior` argument.

Another important aspect is the PSF model...

4.3 Motion-dependent Correlated Noise

5 Acknowledgments

PyKE is build on top of numerous packages that constitute the Python scientific stack, more precisely, `numpy`, `scipy`, `matplotlib`, and `astropy`.