wotan

Sep 24, 2021

Contents

CHAPTER 1

Installation

To install the released version, type

```
$ pip install wotan
```

which automatically installs *numpy*, *numba* and *scipy* if not present. Depending on the algorithm, additional dependencies exist:

- *huber*, *ramsay* and *hampel* depend on *statsmodels*
- *hspline* and *gp* depend on *sklearn*
- *pspline* depends on *pygam*
- *supersmoother* depends on *supersmoother*

To install all additional dependencies, type `$ pip install statsmodels sklearn supersmoother pygam`.

A known incompatibility exists between versions 1.3 of *scipy* and 0.9 of *statsmodels*, as the latest version of scipy deprecated the import for *factorial* from *scipy.misc*. This should be fixed again in a future version of *statsmodels*. Until then, I recommend to *pip install scipy==1.2* (or *conda install scipy==1.2*, if you use conda.).

CHAPTER 2

Wotan Interface

2.1 Detrending with the `flatten` module

Usage example:

```
import numpy as np
from astropy.io import fits

def load_file(filename):
    """Loads a TESS *spoc* FITS file and returns TIME, PDCSAP_FLUX"""
    hdu = fits.open(filename)
    time = hdu[1].data['TIME']
    flux = hdu[1].data['PDCSAP_FLUX']
    flux[flux == 0] = np.nan
    return time, flux

print('Loading TESS data from archive.stsci.edu...')
path = 'https://archive.stsci.edu/hlsps/tess-data-alerts/'
filename = "hlsp_tess-data-alerts_tess_phot_00062483237-s01_tess_v1_lc.fits"
time, flux = load_file(path + filename)

# Use wotan to detrend
from wotan import flatten
flatten_lc1, trend_lc1 = flatten(time, flux, window_length=0.75, return_trend=True,␣
↪method='mean')
flatten_lc2, trend_lc2 = flatten(time, flux, window_length=0.75, return_trend=True,␣
↪method='biweight')

# Plot the result
import matplotlib.pyplot as plt
plt.scatter(time, flux, s=1, color='black')
plt.plot(time, trend_lc1, linewidth=2, color='red')
plt.plot(time, trend_lc2, linewidth=2, color='blue')
plt.xlim(min(time), 1365)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

2.2 Choosing the right window size

Shorter windows (or knot distances, smaller kernels...) remove stellar variability more effectively, but suffer a larger risk of removing the desired signal (the transit) as well. What is the right window size?

For the time-windowed sliders, the window should be 2-3 times longer than the transit duration (for details, read [the paper](www). The transit duration is

$T_{14,\max} = (R_\mathrm{s} + R_\mathrm{p}) \left(\frac{4P}{\pi G M_\mathrm{s}}\right)^{1/3}$

for a central transit on a circular orbit. If you have a prior on the stellar mass and radius, and a (perhaps maximum) planetary period, `wotan` offers a convenience function to calculate $T_{14,\max}$:

As an example, we can calculate the duration of an Earth-Sun transit:

```
from wotan import t14
tdur = t14(R_s=1, M_s=1, P=365, small_planet=True)
print(tdur)
```

This should print ~0.54 (days), or about 13 hours. To protect a transit that long, it is reasonable to choose a window size of 3x as long, or about 1.62 days. With the `biweight` time-windowed slider, we would detrend with these settings:

```
from wotan import t14, flatten
tdur = t14(R_s=1, M_s=1, P=365, small_planet=True)
flatten_lc = flatten(time, flux, window_length=3 * tdur)
```

2.3 Removing outliers before the detrending

Despite robust detrending methods, it is sometimes preferable to remove outliers. Wotan offers a sliding time-windowed function to do this. The user can define an upper and lower threshold, in multiples of the *standard deviation* or the *median absolute deviation*. The middle point in each window can be calculated with the *mean* or the *median*. Outliers are replaced with `NaN` values.

Sliding time-windowed outlier clipper.

Parameters

- **time** (*array-like*) – Time values
- **flux** (*array-like*) – Flux values for every time point
- **window_length** (*float*) – The length of the filter window in units of `time` (usually days)
- **low** (*float or int*) – Lower bound factor of clipping. Default is 3.
- **high** (*float or int*) – Lower bound factor of clipping. Default is 3.
- **method** (`mad` (median absolute deviation; default) or `std` (standard deviation)) – Outliers more than `low` and `high` times the `mad` (or the `std`) from the middle point are clipped
- **center** (`median` (default) or `mean`) – Method to determine the middle point

returns **clipped** (*array-like*) – Input array with clipped elements replaced by `NaN` values.

Example:

```
from wotan import slide_clip
clipped_flux = slide_clip(
    time,
    flux,
    window_length=0.5,
    low=3,
    high=2,
    method='mad',  # mad or std
    center='median'  # median or mean
    )
```

2.4 Removing outliers after the detrending

With robust detrending methods, the trend line (and thus the detrended data) may be unaffected by outliers. In the actual data, however, outliers are still present after detrending. For many purposes, it is acceptable to clip this:

```
from astropy.stats import sigma_clip
flux = sigma_clip(flux, sigma_upper=3, sigma_lower=20)
```

2.5 Masking transits during detrending

If transits have already been discovered, it is best practice to mask them while detrending. This way, the in-transit data points can not influence the detrending.

The current version supports this feature in the `cosine` and `lowess` methods. It is implemented but experimental in most other methods (give it a try...)

Example:

Example:

```
from wotan import transit_mask, flatten
mask = transit_mask(
    time=time_array,
    period=1.234,
    duration=0.1,
    T0=1234.123)
flatten_lc, trend_lc = flatten(
    time,
    flux,
    method='cosine',
    window_length=0.5,
    return_trend=True,
    robust=True,
    mask=mask
    )
```

CHAPTER 3

Usage examples

As follows are usage example for all detrending methods offered by wotan. In all examples, the following synthetic data are used:

```
import numpy as np
from wotan import flatten

points = 1000
time = np.linspace(0, 30, points)
flux = 1 + ((np.sin(time) + time / 10 + time**1.5 / 100) / 1000)
noise = np.random.normal(0, 0.0001, points)
flux += noise
for i in range(points):
    if i % 75 == 0:
        flux[i:i+5] -= 0.0004  # Add some transits
        flux[i+50:i+52] += 0.0002  # and flares
flux[300:400] = np.nan
```

3.1 Robust estimators with tuning constant

Some robust estimators can be tuned: `biweight`, `andrewsinewave`, `welsch`, `huber`, `huber_psi`, `hampel`, `hampelfilt`, `tau`. The `hodges` can not be tuned.

Example usage:

```
flatten_lc, trend_lc = flatten(
    time,                 # Array of time values
    flux,                 # Array of flux values
    method='biweight',
    window_length=0.5,    # The length of the filter window in units of ``time``
    edge_cutoff=0.5,      # length (in units of time) to be cut off each edge.
    break_tolerance=0.5,  # Split into segments at breaks longer than that
    return_trend=True,    # Return trend and flattened light curve
```

(continues on next page)

(continued from previous page)

```
    cval=5.0                # Tuning parameter for the robust estimators
    )
```

Which we can plot as follows:

```
import matplotlib.pyplot as plt
plt.scatter(time, flux, s=1, color='black')
plt.plot(time, trend_lc, color='red', linewidth=2)
plt.show()

plt.close()
plt.scatter(time, flatten_lc, s=1, color='black')
plt.show()
```

Note: Tuning constants `cval` are defined as multiples in units of median absolute deviation from the central location. Defaults are usually chosen to achieve high efficiency for Gaussian distributions. For example, for the `biweight` a `cval` of 6 includes data up to 4 standard deviations (6 median absolute deviations) from the central location and has an efficiency of 98%. Another typical value for the `biweight` is 4.685 with 95% efficiency. Larger values for make the estimate more efficient but less robust. The default for the `biweight` in wotan is 5, as it has shown the best results in the transit injection retrieval experiment. The other defaults are from the literature.

- `biweight` 5
- `andrewsinewave` 1.339
- `welsch` 2.11
- `huber` 1.5
- `huber_psi` 1.28
- `hampel` (1.7, 3.4, 8.5)
- `hampelfilt` 3
- `ramsay` 0.3
- `tau`: 4.5

The `hampel` has a 3-part descending function, known also as (a,b,c). Its tuning constant `cval` must be given as a tuple of 3 values. Typical values are (1.7, 3.4, 8.5) called "17A"; and (2.5, 4.5, 9.5) called "25A". With values given as multiples of the median absolute deviation, the 25A can be stated equivalently: a' = 1.686, b' = 3.035 c' = 6.408 as multiples of the standard deviation.

3.2 Trimmed methods

There are 3 methods which first focus on outlier treatment, followed by taking the mean in a second stage: `trim_mean`, `winsorize` and `hampelfilt`.

- The `hampelfilt` was already discussed in the previous section because its threshold is defined as `cval` times the median absolute deviation, beyond which it replaces values with the median.
- The `trim_mean` deletes the fraction `proportiontocut` from both sides of the distribution.
- The `winsorize` replaces the fraction `proportiontocut` from both sides of the distribution with the remaining values at the edges.

Example usage:

```
flatten_lc, trend_lc = flatten(
    time,                  # Array of time values
    flux,                  # Array of flux values
    method='trim_mean',
    window_length=0.5,     # The length of the filter window in units of ``time``
    edge_cutoff=0.5,       # length (in units of time) to be cut off each edge.
    break_tolerance=0.5,   # Split into segments at breaks longer than that
    return_trend=True,     # Return trend and flattened light curve
    proportiontocut=0.1    # Cut 10% off both ends
    )
```

3.3 median, mean

These methods ignore the parameters `proportiontocut` and `cval`

Example usage:

```
flatten_lc, trend_lc = flatten(
    time,                  # Array of time values
    flux,                  # Array of flux values
    method='median',
    window_length=0.5,     # The length of the filter window in units of ``time``
    edge_cutoff=0.5,       # length (in units of time) to be cut off each edge.
    break_tolerance=0.5,   # Split into segments at breaks longer than that
    return_trend=True,     # Return trend and flattened light curve
    )
```

3.4 medfilt

This method is cadence-based. Included to compare to the time-windowed `median`. The parameter `window_length` is now in units of cadence (i.e., array data points). It ignores the parameters `edge_cutoff` and `break_tolerance`.

Example usage:

```
flatten_lc, trend_lc = flatten(
    time,                  # Array of time values
    flux,                  # Array of flux values
    method='medfilt',
    window_length=31 ,     # The length of the filter window in cadences
    return_trend=True,     # Return trend and flattened light curve
    )
```

3.5 Spline: robust rspline

Spline with iterative sigma-clipping. It does not provide `edge_cutoff`, but benefits greatly from using a sensible `break_tolerance`. Example usage:

```
flatten_lc, trend_lc = flatten(
    time,                   # Array of time values
    flux,                   # Array of flux values
    method='rspline',
    window_length=0.5,      # The knot distance in units of ``time``
    break_tolerance=0.5,    # Split into segments at breaks longer than that
    return_trend=True,      # Return trend and flattened light curve
    )
```

3.6 Spline: robust hspline

Spline with robust Huber-estimator (linear and quadratic loss). It does not provide `edge_cutoff`, but benefits greatly from using a sensible `break_tolerance`. Example usage:

```
flatten_lc, trend_lc = flatten(
    time,                   # Array of time values
    flux,                   # Array of flux values
    method='hspline',
    window_length=0.5,      # The knot distance in units of ``time``
    break_tolerance=0.5,    # Split into segments at breaks longer than that
    return_trend=True,      # Return trend and flattened light curve
    )
```

3.7 Spline: robust penalized pspline

Major update with version 1.6.

Robust spline through iterative sigma-clipping. The iterations (as printed during the runtime) make the spline fit robust against outliers. In each iteration, data points more than `PSPLINES_STDEV_CUT=2` standard deviations away from the fit are removed. Remaining data are fit again. The iteration cycle stops when zero outliers remain, or (at the latest) after `PSPLINES_MAXITER=10` iterations are completed.

In each iteration, `PyGAM` is used to determine the optimal number of splines (with equidistantly spaced knots). It tests `n=[1,..max_splines]` knots. In each test, the sum of the squared residuals is noted. Afterwards, a "penalty calculation" is performed. More knots make a smoother fit, i.e. smaller residuals. But more knots are "bad" due to the risk of overfitting. Both measures are weighted against each other, i.e. the number of knots is penalized. Per default, the L2 norm (ridge smoothing) is used.

The `edge_cutoff` functionality is provided. The penalized spline method benefits greatly from using a sensible `break_tolerance`. Example usage:

```
flatten_lc, trend_lc, nsplines = flatten(
    time,                   # Array of time values
    flux,                   # Array of flux values
    method='pspline',
    max_splines=100,        # The maximum number of knots to be tested
    edge_cutoff=0.5,        # Remove edges
    stdev_cut=2             # Larger outliers are removed in each iteration
    return_trend=True,      # Return trend and flattened light curve
    return_nsplines=True,   # Return chosen number of knots
    verbose=False           # If true, prints status during runtime
    )
```

which returns the usual flattened light curve and the actual trend. In addition, when choosing `return_nsplines=True`, the chosen spline value (number of knots) is returned. This is done separately for each segment, in case `break_tolerance>0` resulted in segmentation. Check this with:

```
print('lightcurve was split into', len(nsplines), 'segments')
print('chosen number of splines', nsplines)
```

which returns something like:

```
lightcurve was split into 2 segments
nsplines [19. 26.]
```

To determine the distance between the knots in each segment, you can run

```
from wotan.gaps import get_gaps_indexes
segs = get_gaps_indexes(time, break_tolerance=break_tolerance)
segs[-1] -= 2  # remove endpoint padding
durations = []
for seg in range(len(segs)-1):
    start = time[segs[seg]]
    stop = time[segs[seg+1]-1]
    duration = stop - start
    durations.append(duration)
    print(start, stop, duration)
print('Segment durations', durations)
print('Time between knots', durations / nsplines)
```

which prints something like:

```
Segment durations [7.54, 12.25, 12.14, 12.33]
Time between knots [0.25 0.27 0.21 0.16]
```

3.8 Lowess / Loess

Locally weighted scatterplot smoothing (Cleveland 1979). Offers segmentation (`break_tolerance`), but no edge clipping (`edge_cutoff`). For similar results compared to other spline-based methods or sliders, use a `window_length` about twice as long. Example usage:

```
flatten_lc, trend_lc = flatten(
    time,                 # Array of time values
    flux,                 # Array of flux values
    method='lowess',
    window_length=1,      # The length of the filter window in units of ``time``
    break_tolerance=0.5,  # Split into segments at breaks longer than that
    return_trend=True,    # Return trend and flattened light curve
    )
```

3.9 CoFiAM

Cosine Filtering with Autocorrelation Minimization. Does not provide `edge_cutoff`, but benefits greatly from using a sensible `break_tolerance`. Example usage:

```
flatten_lc, trend_lc = flatten(
    time,                  # Array of time values
    flux,                  # Array of flux values
    method='cofiam',
    window_length=0.5,     # Protected window span in units of ``time``
    break_tolerance=0.5,   # Split into segments at breaks longer than that
    return_trend=True,     # Return trend and flattened light curve
    )
```

3.10 Fitting of sines and cosines

Fits a sum of sines and cosines, where the highest order is determined by the protected window span `window_length` in units of `time`. A robustification (iterative sigma-clipping of 2-sigma outliers until convergence) is available by setting the parameter `robust=True`. Example usage:

```
flatten_lc, trend_lc = flatten(
    time,                  # Array of time values
    flux,                  # Array of flux values
    method='cosine',
    robust='True',         # iterative sigma-clipping of 2-sigma outliers until␣
↪convergence
    window_length=0.5,     # Protected window span in units of ``time``
    break_tolerance=0.5,   # Split into segments at breaks longer than that
    return_trend=True,     # Return trend and flattened light curve
    )
```

3.11 SuperSmoother

Friedman's (1984) Super-Smoother, a local linear regression with adaptive bandwidth. Does not provide `edge_cutoff`, but benefits greatly from using a sensible `break_tolerance`. Example usage:

```
flatten_lc, trend_lc = flatten(
    time,                  # Array of time values
    flux,                  # Array of flux values
    method='supersmoother',
    window_length=0.5,     # The knot distance in units of ``time``
    break_tolerance=0.5,   # Split into segments at breaks longer than that
    return_trend=True,     # Return trend and flattened light curve
    cval=None              # Bass enhancement (smoothness)
    )
```

Note: `cval` determines the bass enhancement (smoothness) and can be *None* or in the range $0 <$ `cval` < 10. Smaller values make the trend more flexible to fit out small variations.

3.12 Savitzky-Golay savgol

Sliding segments are fit with polynomials (Savitzky & Golay 1964). This filter is cadence-based (not time-windowed), so that `window_length` must be an integer value. If an even integer is provided, it is made uneven (a requirement)

by adding 1. The polyorder is set by `cval` (default: 2 - the best value from our experiments). Does not provide `edge_cutoff`, but benefits from using a sensible `break_tolerance`.

Example usage:

```
flatten_lc, trend_lc = flatten(
    time,                # Array of time values
    flux,                # Array of flux values
    method='savgol',
    cval=2,              # Defines polyorder
    window_length=51,    # The window length in cadences
    break_tolerance=0.5, # Split into segments at breaks longer than that
    return_trend=True,   # Return trend and flattened light curve
    )
```

3.13 Gaussian Processes

Available kernels are :

- `squared_exp` Squared-exponential kernel, with option for iterative sigma-clipping
- `matern` Matern 3/2 kernel, with option for iterative sigma-clipping
- `periodic` Periodic kernel informed by a user-specified period
- `periodic_auto` Periodic kernel informed by a Lomb-Scargle periodogram pre-search

GPs do not provide `edge_cutoff`, but benefit from using a sensible `break_tolerance`.

Example usage:

```
flatten_lc, trend_lc = flatten(
    time,                # Array of time values
    flux,                # Array of flux values
    method='gp',
    kernel='squared_exp', # GP kernel choice
    kernel_size=10,       # GP kernel length
    break_tolerance=0.5, # Split into segments at breaks longer than that
    return_trend=True,   # Return trend and flattened light curve
    )
```

Note: The sensible `kernel_size` varies between kernels.

A robustification (iterative sigma-clipping of 2-sigma outliers until convergence) is available by setting the parameter `robust=True`:

```
flatten_lc, trend_lc = flatten(
    time,                # Array of time values
    flux,                # Array of flux values
    method='gp',
    kernel='squared_exp', # GP kernel choice
    kernel_size=10,       # GP kernel length
    break_tolerance=0.5, # Split into segments at breaks longer than that
    robust=True,         # Robustification using iterative sigma clipping
    return_trend=True,   # Return trend and flattened light curve
    )
```

Here we can simply swap `kernel='squared_exp'` for `kernel='matern'` and play with `kernel_size` to get a very similar result.

In the presence of strong periodicity, we can also use the periodic kernel. This version does not support robustification. If we know the period, we can do this.

```
flatten_lc2, trend_lc2 = flatten(
    time,                   # Array of time values
    flux,                   # Array of flux values
    method='gp',
    kernel='periodic',      # GP kernel choice
    kernel_period=2*3.14,   # GP kernel period
    kernel_size=10,         # GP kernel length
    break_tolerance=0.5,    # Split into segments at breaks longer than that
    return_trend=True,      # Return trend and flattened light curve
    )
```

Usually, however, it is better to let wotan detect the period. We can do this by setting `kernel='periodic_auto'`. Then, a Lomb-Scargle periodogram is calculated, and the strongest peak is used as the period. In addition, a Matern kernel is added to consume the remaining non-periodic variation. This version does not support robustification. Example:

```
flatten_lc2, trend_lc2 = flatten(
    time,                     # Array of time values
    flux,                     # Array of flux values
    method='gp',
    kernel='periodic_auto',   # GP kernel choice
    kernel_size=10,           # GP kernel length
    break_tolerance=0.5,      # Split into segments at breaks longer than that
    return_trend=True,        # Return trend and flattened light curve
    )
```

Python Module Index

h

s

Index

H

S