periodicity Documentation

Release 1.0b4

Eduardo Nunes

Oct 19, 2021

USER GUIDE

`periodicity` is a toolkit of period determination methods. The latest development version can be found here.

CHAPTER ONE

INSTALLATION

`periodicity` requires `xarray` (for computation on labeled data), `celerite2` and `george` (for Gaussian Process models), `emcee` and `pymc3_ext` (for MCMC sampling Gaussian Processes), and `PyWavelets` (for Wavelet analysis).

Installing the most recent stable version of the package is as easy as:

```
pip install periodicity
```

CHAPTER

TWO

CHANGELOG

2.1 1.0 (2021-00-00)

Initial beta release

2.1.1 Signal objects

class periodicity.core.**FSeries**(*frequency=None*, *values=None*, *assume_sorted=False*)

Bases: periodicity.core.Signal

curvefit(*fun*, *use_period=False*, ***kwargs*)

property df

downsample(*df=None*, *dp=None*, *func=<function nanmean>*)

property dp

dropna()

fmax()

property frequency

from_xray(*xray*)

ifft(*nfft=None*)

property median_df

property median_dp

property period

property period_at_highest_peak

property period_at_highest_prominence

periods_at_half_max(*peak_order=1*, *use_prominence=False*)

plot(**args*, ***kwargs*)

pmax()

polyfit(*degree*, *use_period=False*)

psort_by_peak()

psort_by_prominence()

class `periodicity.core.`**`TFSeries`**(*time=None*, *frequency=None*, *values=None*)

Bases: `periodicity.core.Signal`

`contour`(**args*, ***kwargs*)

`contourf`(**args*, ***kwargs*)

`property df`

`downsample`(*dt=None*, *df=None*, *dp=None*, *func=<function nanmean>*)

`property dp`

`property dt`

`property frequency`

`from_xray`(*xray*)

`imshow`(**args*, ***kwargs*)

`property median_df`

`property median_dp`

`property median_dt`

`pcolormesh`(**args*, ***kwargs*)

`property period`

`surface`(**args*, ***kwargs*)

`property time`

class `periodicity.core.`**`TSeries`**(*time=None*, *values=None*, *assume_sorted=False*)

Bases: `periodicity.core.Signal`

`property TEO`

Teager Energy Operator (TEO)

J. F. Kaiser, "On Teager's energy algorithm and its generalization to continuous signals", Proc. 4th IEEE Signal Processing Workshop, 1990.

`acf`(*max_lag=None*, *unbias=False*)

Auto-Correlation Function implemented using IFFT of the power spectrum.

Parameters

- **`max_lag`** (`int or float, optional`) – Maximum lag to compute ACF. If given as a float, will be assumed to be a measure of time and the ACF will be computed for lags lower than or equal to *max_lag*.
- **`unbias`** (`bool, optional`) – Whether to correct for the "mask effect" (dividing Ryy by the ACF of a signal equal to 1 on the original domain of y and equal to 0 on the padding's domain).

Returns **acf** – ACF of input signal.

Return type *TSeries*

`acf_period_quality`(*p_min*, *p_max*)

Calculates the ACF quality of a band-pass filtered version of the signal.

Parameters

- **`p_min`** (`float, optional`) – Lower cutoff period to filter signal.

- **p_max** (`float, optional`) – Higher cutoff period to filter signal. Must be between *p_min* and half the baseline.

Returns

- **best_per** (*float*) – Highest peak (best period) for the ACF of the filtered signal.
- **height** (*float*) – Maximum height for the ACF of the filtered signal.
- **quality** (*float*) – Quality factor of the best period.

property baseline

butterworth(*fmin=None*, *fmax=None*, *order=5*)
Implements a IIR butterworth filter.

Parameters

- **fmin** (`float`) – Lower cutoff frequency.
- **fmax** (`float`) – Higher cutoff frequency.
- **order** (`int, optional`) – Order of the butterworth filter. Default is 5.

Returns **filt_signal** – Filtered signal.

Return type *TSeries*

corr(*other*)

cov(*other*)

curvefit(*fun*, ***kwargs*)

property derivative

downsample(*dt*, *func=<function nanmean>*)

drop(*index=None*)

dropna()

property dt

fft(*oversample=1.0*, *dt=None*)

fill_gaps(*dt=None*, ***kwargs*)

fold(*period*, *t0=0*)

from_xray(*xray*, *assume_sorted=False*)

get_envelope(*pad_width=0*, ***peak_kwargs*)
Interpolates maxima/minima with cubic splines into upper/lower envelopes.

Parameters

- **peak_kwargs** (`float, optional`) – Keyword arguments to use in *find_extrema*.
- **pad_width** (`int, optional`) – Number of extrema to repeat on either side of the signal.

Returns

- **upper** (*Signal*) – Upper envelope.
- **lower** (*Signal*) – Lower envelope.

interp(*new_time=None*, *method='linear'*, ***kwargs*)
Interpolation onto a new time grid.

Parameters

- **new_time** (*ndarray, optional*) – Sampling grid. If omitted, it will be uniform with period *median_dt*.
- **method** (*{'linear', 'spline', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic'}*) – Interpolation method to be used.
- **s** (*float, optional*) – Smoothing condition for the spline. Ignored unless method == "spline".

Returns **uniform_signal** – Interpolated signal.

Return type *Signal*

interpolate_na(*method='linear'*, ***kwargs*)

join(*other*, ***kwargs*)

property median_dt

pad(*pad_width*, ***kwargs*)

plot(**args*, ***kwargs*)

polyfit(*degree*)

psd(**args*, ***kwargs*)

split(*max_gap=None*)

property time

timescale(*alpha*)

timeshift(*t0*)

tmax()

2.1.2 Spectral methods

The basics: Fourier analysis

$$\mathcal{F}\{x(t)\} = \int_{-\infty}^{\infty} x(t) \exp(-j2\pi f t) dt$$

The Lomb-Scargle periodogram

class periodicity.spectral.**BGLST**
Bases: object

class periodicity.spectral.**GLS**(*fmin=None*, *fmax=None*, *n=5*, *psd=False*)
Bases: object

References

bootstrap(*n_bootstraps*, *random_seed=None*)

copy()

fal(*fap*)

fap(*power*)

fap_level: array-like, optional List of false alarm probabilities for which you want to calculate approximate levels. Can also be passed as a single scalar value.

model(*tf*, *f0*)

Compute the Lomb-Scargle model fit at a given frequency

Parameters

- **tf** (*float or array-like*) – The times at which the fit should be computed
- **f0** (*float*) – The frequency at which to compute the model

Returns **yf** – The model fit evaluated at each value of tf

Return type ndarray

window()

References and additional reading

Vanderplas 2018. *Understanding the Lomb-Scargle Periodogram.* ApJS, 236, 16.

2.1.3 Phase-folding methods

class periodicity.phase.**PDM**(*nb=5*, *nc=2*, *p_min=None*, *p_max=None*, *n_periods=1000*, *oversample=1*, *do_subharmonic=False*, *cores=None*)

Bases: object

Phase Dispersion Minimization[1].

Parameters

- **nb** (*int, optional*) – Number of phase bins (the default is 5).
- **nc** (*int, optional*) – Number of covers per bin (the default is 2).
- **p_min** (*float, optional*) – Minimum/maximum trial period.
- **p_max** (*float, optional*) – Minimum/maximum trial period.
- **n_periods** (*int, optional*) – Number of trial periods (the default is 1000).
- **oversample** (*scalar, optional*) – If *p_max* is omitted, the time baseline multiplied by *oversample* will be used instead.
- **do_subharmonic** (*bool, optional*) – Whether to perform subharmonic averaging. This option looks for a significant minimum in θ at both the main period and its double. For actual variations, both will be present. For a noise result, the double period signal will not be present.
- **cores** (*int, optional*) – The number of parallel cores to use. By default it will try to use all available CPU cores.

[1] R. F. Stellingwerf, "Period determination using phase dispersion minimization," ApJ, September 1978.

References

_pdm(*period*)
Calculates the PDM theta statistic for a single trial period.

class periodicity.phase.**StringLength**(*dphi=0.1*, *n_periods=1000*, *cores=None*)
Bases: `object`

String Length[2].

Parameters

- **dphi** (*float, optional*) – Factor to multiply (1 / baseline) in order to get the frequency separation (the default is 0.1).
- **n_periods** (*int, optional*) – Number of trial periods (the default is 1000).
- **cores** (*int, optional*) – The number of parallel cores to use. By default it will try to use all available CPU cores.

References

_stringlength(*period*)
Calculates the string length for a single trial period.

References and additional reading

2.1.4 Gaussian Processes

The determination of periodicities using Gaussian Processes has been getting somewhat trendy as of recent years. In the area of astrophysical time series analysis, it's been used with varying degrees of success to study stellar magnetic cycles (timescale of years) and stellar rotation (timescale of days), none of which are strictly periodic processes.

class periodicity.gp.**BrownianGP**(*signal*, *err*, *init_period=None*, *period_ppf=None*)
Bases: *periodicity.gp.CeleriteModeler*

prior_transform(*u*)

class periodicity.gp.**BrownianTheanoGP**(*signal*, *err*, *init_period=None*, *predict_at=None*, *psd_at=None*)
Bases: *periodicity.gp.TheanoModeler*

class periodicity.gp.**CeleriteModeler**(*signal*, *err*, *init_period=None*, *period_ppf=None*)
Bases: `object`

get_kernel(*tau*, *gp*)

get_prediction(*time*, *gp*)

get_psd(*frequency*, *gp*)

log_prob(*u*, *gp*)

mcmc(*n_walkers=50*, *n_steps=1000*, *burn=0*, *use_prior=False*, *random_seed=None*)
Samples the posterior probability distribution with a Markov Chain Monte Carlo simulation.

Parameters

- **n_walkers** (*int, optional*) – Number of walkers (the default is 50).

[2] R. F. Stellingwerf, "Period Determination of RR Lyrae Stars," RR Lyrae Stars, Metal-Poor Stars and the Galaxy, August 2011.

- **n_steps** (*int, optional*) – Number of steps taken by each walker (the default is 1000).
- **burn** (*int, optional*) – Number of burn-in samples to remove from the beginning of the simulation (the default is 0).
- **use_prior** (*bool, optional*) – Whether to start walkers by sampling from the prior distribution. The default is False, in which case a ball centered at the MLE hyperparameter vector is used.

Returns

- **trace** (*dict*) – Samples of the posterior hyperparameter distribution.
- **tau** (*ndarray*) – Estimated autocorrelation time of MCMC chain for each parameter.

minimize(*gp*, ***kwargs*)
Gradient-based optimization of the objective function within the unit hypercube.

nll(*u*, *gp*)
Objective function based on the Negative Log-Likelihood.

prior_transform(*u*)

set_params(*params*, *gp*)

class periodicity.gp.**GeorgeModeler**(*signal*, *err*, *init_period=None*, *period_prior=None*, *bounds=None*, *constraints=None*)
Bases: object

get_kernel(*tau*, *gp*)

get_prediction(*time*, *gp*)

grad_nll(*theta*, *gp*)

log_prior(*theta*)

log_prob(*theta*, *gp*)
Posterior distribution over the hyperparameters.

mcmc(*n_walkers=50*, *n_steps=1000*, *burn=0*, *random_seed=None*)
Samples the posterior probability distribution with a Markov Chain Monte Carlo simulation.

Parameters

- **n_walkers** (*int, optional*) – Number of walkers (the default is 50).
- **n_steps** (*int, optional*) – Number of steps taken by each walker (the default is 1000).
- **burn** (*int, optional*) – Number of burn-in samples to remove from the beginning of the simulation (the default is 0).
- **use_prior** (*bool, optional*) – Whether to start walkers by sampling from the prior distribution. The default is False, in which case a ball centered at the MLE hyperparameter vector is used.

Returns **samples** – Samples of the posterior hyperparameter distribution.

Return type ndarray[n_dim, n_walkers * (n_steps - burn)]

minimize(*gp*, *grad=False*, ***kwargs*)
Gradient-based optimization of the objective function within the unit hypercube.

nll(*theta*, *gp*)
Objective function based on the Negative Log-Likelihood.

set_params(*theta*, *gp*)

class periodicity.gp.**HarmonicGP**(*signal*, *err*, *init_period=None*, *period_ppf=None*)
Bases: `periodicity.gp.CeleriteModeler`

prior_transform(*u*)

class periodicity.gp.**HarmonicTheanoGP**(*signal*, *err*, *init_period=None*, *predict_at=None*, *psd_at=None*)
Bases: `periodicity.gp.TheanoModeler`

class periodicity.gp.**QuasiPeriodicGP**(*signal*, *err*, *init_period=None*, *period_prior=None*, *bounds=None*, *constraints=None*)
Bases: `periodicity.gp.GeorgeModeler`

log_prior(*theta*)

class periodicity.gp.**TheanoModeler**(*signal*, *err*, *init_period=None*)
Bases: `object`

mcmc(*n_walkers=1*, *n_steps=2000*, *burn=1000*, *cores=1*)

References and additional reading

2.1.5 Time-Frequency methods

class periodicity.timefrequency.**HHT**(*frequencies*, *emd=None*, *method='DQ'*, *norm_type='spline'*, *norm_iter=10*, *smooth_width=None*)
Bases: `object`

Hilbert-Huang Transform

_normalize(*mode*, *eps=1e-06*, *pad_width=2*)
Huang et al. (2009)

_spectrogram(*freq_grid*, *freq*, *amp*)

class periodicity.timefrequency.**WPS**(*periods*)
Bases: `object`

Wavelet Power Spectrum using Morlet wavelets.

Parameters **periods** (*array-like*) – Periods to consider, in the same time units as the input signal.

coi(*coi_samples=100*)

coi_samples: int, optional Number of samples of the Cone of Influence (COI). Used for plotting (default is 100).

gwps(*tmin=None*, *tmax=None*)

property mask_coi

masked_gwps(*tmin=None*, *tmax=None*)

masked_sav(*pmin=None*, *pmax=None*)

plot_coi(*coi_samples=100*, ***kwargs*)

sav(*pmin=None*, *pmax=None*)

References and additional reading

Hilbert-Huang Transform on Scholarpedia.

2.1.6 Decomposition methods

class periodicity.decomposition.**CEEMDAN**(*epsilon=0.2*, *ensemble_size=50*, *min_energy=0.0*, *random_seed=None*, *cores=None*, ***kwargs*)

Bases: object

_realization(*task*)

property c_orthogonality_matrix

property orthogonality_matrix

postprocessing()

class periodicity.decomposition.**EMD**(*max_iter=2000*, *pad_width=2*, *theta_1=0.05*, *theta_2=0.5*, *alpha=0.05*)

Bases: object

iter(*sig*)

sift(*sig*)

class periodicity.decomposition.**LMD**(*max_iter=10*, *pad_width=0*, *smooth_iter=12*, *eps=1e-06*)

Bases: object

iter(*sig*)

sift(*sig*)

class periodicity.decomposition.**VMD**

Bases: object

```
%matplotlib inline
%config InlineBackend.figure_format = "retina"
```

2.1.7 Stellar rotation with Gaussian Processes

This tutorial will introduce the basic features of Gaussian Process (GP) regression via `periodicity` by studying the `SpottedStar` photometric data from Kepler. The module `periodicity.gp` includes a few predefined classes using different GP kernels; their difference is better explained in the User Guide, but they all have one thing in common: a "period" hyperparameter. One can also define their very own `periodicity` compatible GP modeler, using either `george` or `celerite2` to define their kernel, but here we will stick with the prepackaged `HarmonicGP` class.

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

from periodicity.core import TSeries
from periodicity.data import SpottedStar
from periodicity.gp import HarmonicGP, make_gaussian_prior, make_ppf
```

The `HarmonicGP` is built upon the `celerite2.terms.RotationTerm`, which is the sum of two Stochastically-driven Harmonic Oscillator (SHO) terms; the secondary term"s period is half the primary one"s, and their relative contributions are controlled by a mixing ratio hyperparameter. More details can be found at `celerite2` excellent documentation.

More importantly, this kernel presents both positive and negative covariance values, which can be useful for some processes (this being one of them).

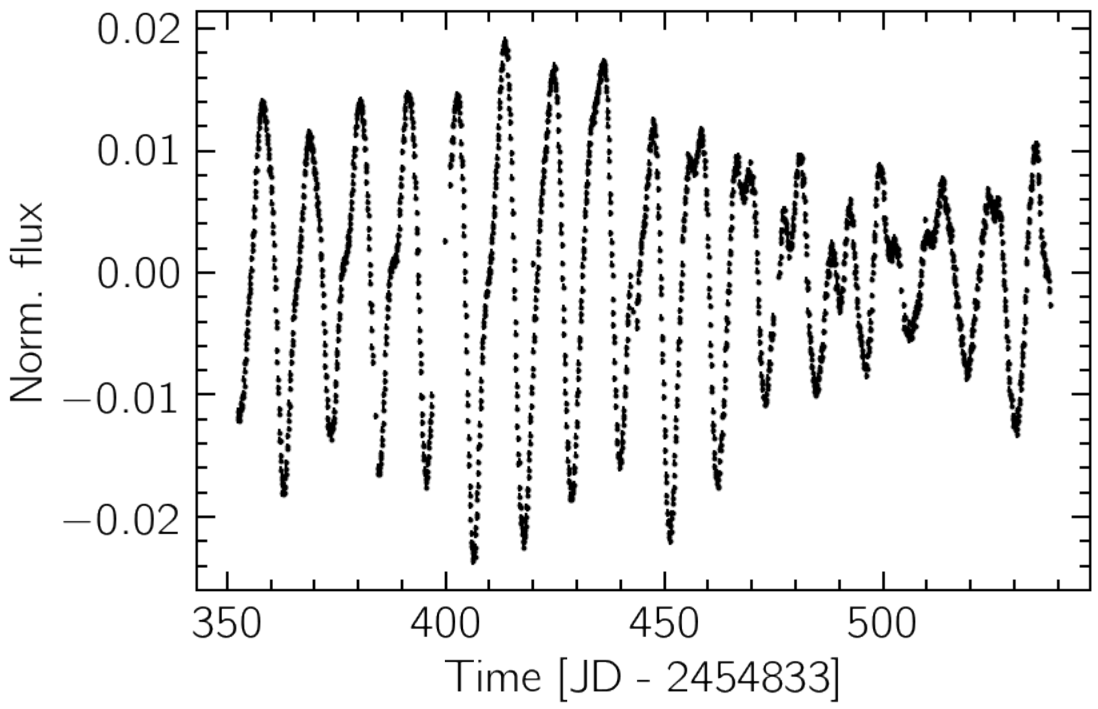
```
plt.rc("lines", linewidth=1.0, linestyle="-", color="black")
plt.rc("font", family="sans-serif", weight="normal", size=12.0)
plt.rc("text", color="black", usetex=True)
plt.rc("text.latex", preamble="\\usepackage{cmbright}")
plt.rc(
    "axes",
    edgecolor="black",
    facecolor="white",
    linewidth=1.0,
    grid=False,
    titlesize="x-large",
    labelsize="x-large",
    labelweight="normal",
    labelcolor="black",
)
plt.rc("axes.formatter", limits=(-4, 4))
plt.rc(("xtick", "ytick"), labelsize="x-large", direction="in")
plt.rc("xtick", top=True)
plt.rc("ytick", right=True)
plt.rc(("xtick.major", "ytick.major"), size=7, pad=6, width=1.0)
plt.rc(("xtick.minor", "ytick.minor"), size=4, pad=6, width=1.0, visible=True)
plt.rc("legend", numpoints=1, fontsize="x-large", shadow=False, frameon=False)
```

The creation of a GP object is pretty straightforward. Once you have a TSeries representing your data, the model can be initialized with the corresponding uncertainties as such:

```
time, flux, flux_err = SpottedStar()
signal = TSeries(time, flux)
model = HarmonicGP(signal, flux_err)
```

The data are taken from Kepler observations of the star KIC 9655172 from December 2009 to June 2010 (totalling six months or ~180 days). If we plot the data, we can visually detect a seasonality of ~10 days, which is clearly not strictly periodic. We also note the tiny size of the error bars, indicating a very high signal-to-noise ratio.

```
plt.errorbar(time, flux, flux_err, fmt="k.", ms=2)
plt.xlabel("Time [JD - 2454833]")
plt.ylabel("Norm. flux");
```



Maximum Likelihood Hyperparameters

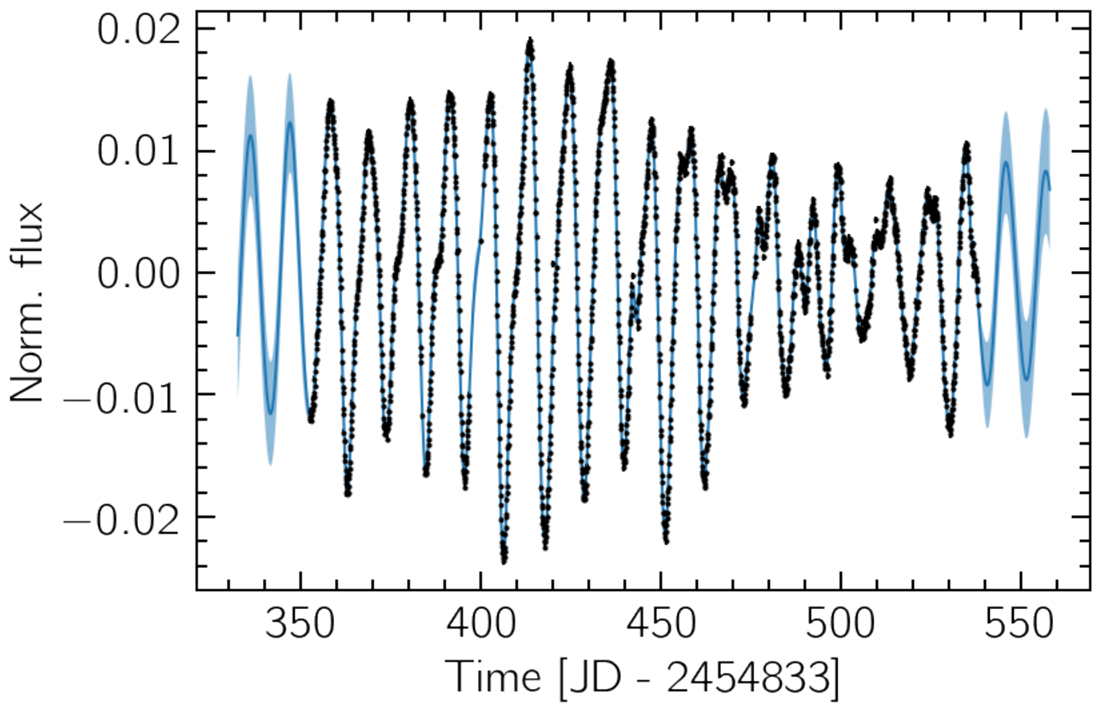
The model hyperparameters can be optimized to maximize the likelihood of the data. This is performed on the unit hypercube, and can be converted to real units using the `prior_transform` method.

```
soln, opt_gp = model.minimize(model.gp)
print(model.prior_transform(soln.x))
```

```
{'mean': -0.00016600149835243423, 'sigma': 0.00886947801744532, 'period': 11.
↪025243233068581, 'Q0': 5.056973979108306, 'dQ': 42.38472792648548, 'f': 0.
↪1432079772222711, 'jitter': 8.745275247612219e-08}
```

The resulting optimal kernel can then be used to make predictions on different time values. This is not only used to interpolate missing observations, but can also extrapolate to unobserved past of future times. The uncertainties here are essential to interpret the resulting predictions.

```
tpred = np.linspace(model.t[0] - 20, model.t[-1] + 20, 1000)
ypred, sd = model.get_prediction(tpred, opt_gp)

plt.plot(tpred, ypred)
plt.errorbar(time, flux, flux_err, fmt="k.", ms=2)
plt.fill_between(tpred, ypred - sd, ypred + sd, alpha=0.5)
plt.xlabel("Time [JD - 2454833]")
plt.ylabel("Norm. flux");
```

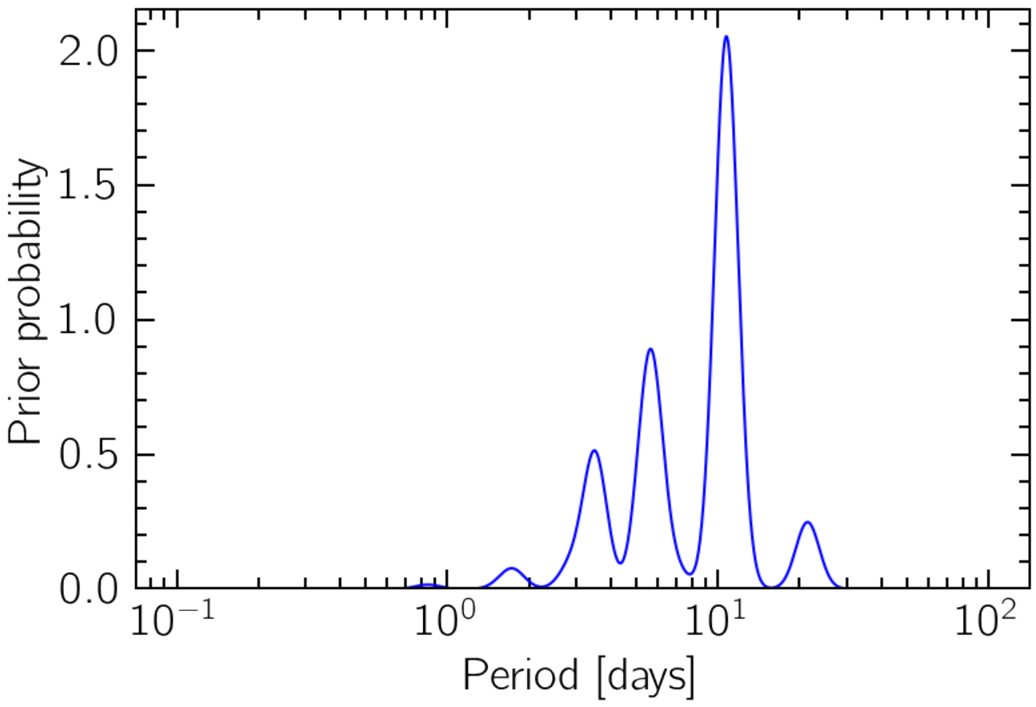


Gaussian Mixture as a Prior Probability Distribution

In order to better estimate the hyperparameter values, a Bayesian inference is usually recommended. Since we are mostly concerned with the period of the process, this is the only hyperparameter we will try and make as informative as possible.

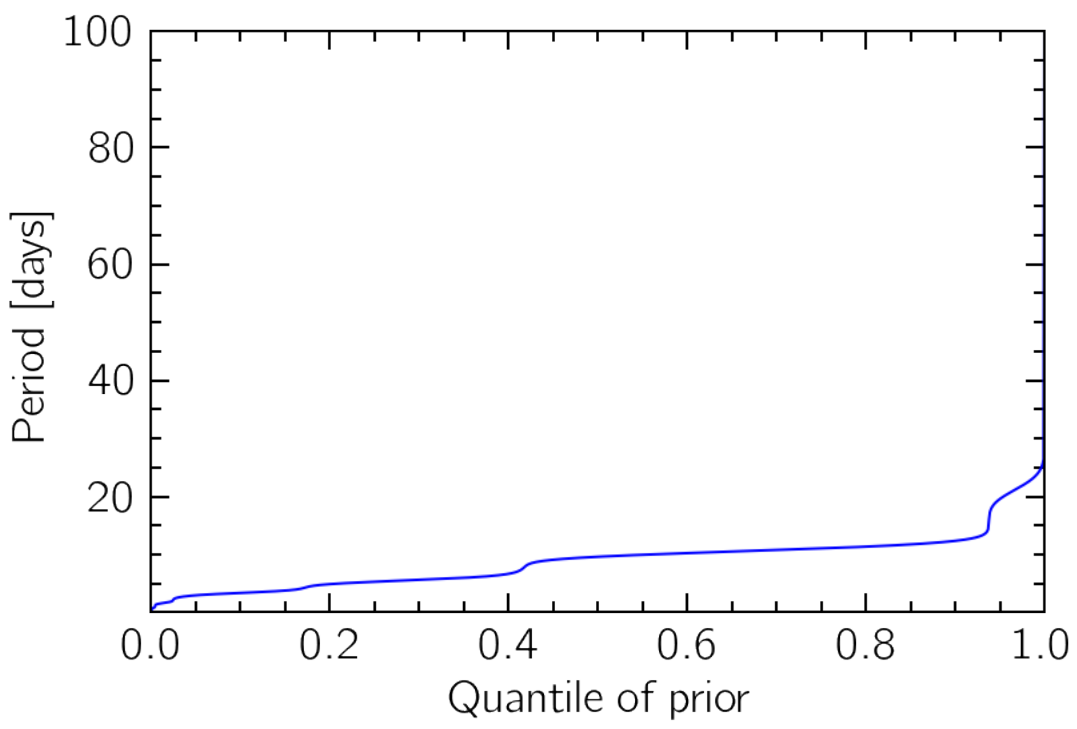
One option is creating a Gaussian Mixture distribution, where each Gaussian component is determined by the autocorrelation of different band-pass-filtered versions of the original data. This is based on the idea of Angus et al. (2018).

```
prior = make_gaussian_prior(signal)

periods = np.logspace(-1, 2, 1000)
prior_probs = prior(np.log(periods))
plt.plot(periods, prior_probs, "b")
plt.xlabel("Period [days]")
plt.ylabel("Prior probability")
plt.ylim(0)
plt.xscale("log");
```



The way these GP objects work, instead of a mapping from period-space into probability-space we will need to specify a reverse transformation, from the probability-space (unit line) into the corresponding quantiles of the distribution. This can be very hard to do analytically, depending on the exact distribution, so we use `make_ppf` to create a numeric interpolation that inverts the cumulative distribution of the PDF we just defined.

```
prior_ppf = make_ppf(periods, prior_probs)
model.period_prior = prior_ppf

quantiles = np.linspace(0, 1, 1000)
plt.plot(quantiles, prior_ppf(quantiles), "b")
plt.xlim(0, 1)
plt.ylim(0.1, 100)
plt.xlabel("Quantile of prior")
plt.ylabel("Period [days]");
```



Sampling the Posterior Probability Density

Now that our prior is defined, running a Markov Chain Monte Carlo (MCMC) simulation is everything we need to get a feel for the posterior distribution of the hyperparameters.

```
samples, tau = model.mcmc(n_walkers=32, n_steps=5000, burn=500, random_seed=42)
```

```
100%|| 5000/5000 [06:34<00:00, 12.67it/s]
The chain is shorter than 50 times the integrated autocorrelation time for 6␣
↪parameter(s). Use this estimate with caution and run a longer chain!
N/50 = 90;
tau: [ 99.72551401 154.98042982 103.92194232 147.91348714 186.18706053
 215.56281431  89.18027946]
```

You will notice the warning message telling us the chain is too short to accurately estimate the integrated autocorrelation time; since we are keeping things relatively simple and won"t run any convergence diagnostics, we can safely ignore this (for now).

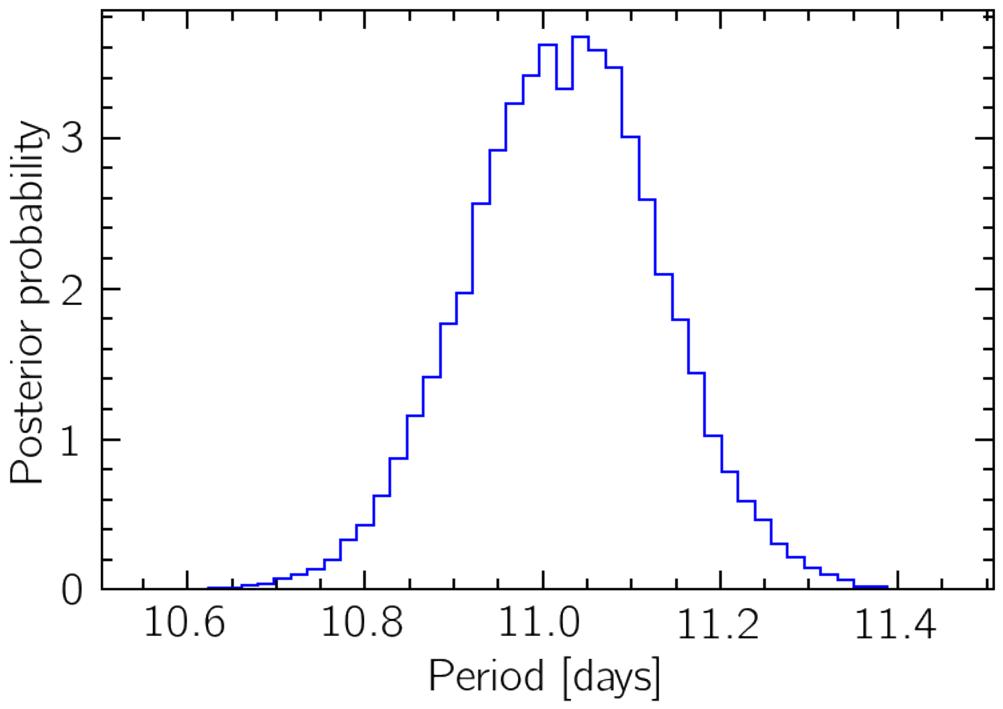
The samples we obtain are a dictionary, with a label for each hyperparameter. This can be useful for further analysis using a library like pandas, for example.

Our MAP period estimate can be taken from the median of the posterior distribution, and we can also estimate a confidence interval from the 16th and 84th percentiles:

```
print("Median period:", np.median(samples["period"]))
print("16th percentile:", np.percentile(samples["period"], 16))
print("84th percentile:", np.percentile(samples["period"], 84))
```

```
Median period: 11.02531261135498
16th percentile: 10.916270459499716
84th percentile: 11.130755005207826
```

```
bins = np.linspace(np.min(samples["period"]), np.max(samples["period"]), 50)
plt.hist(samples["period"], bins=bins, histtype="step", color="b", density=True)
plt.xlabel("Period [days]")
plt.ylabel("Posterior probability");
```



```
%matplotlib inline
%config InlineBackend.figure_format = "retina"
```

2.1.8 Wavelets & Cone of Influence

This tutorial will show the main steps involved in Wavelet analysis using `periodicity`. As with other tools from the `timefrequency` module, the aim is to localize the periodicities in time. That's specially useful for signals with additional information in their varying variance or fundamental frequency.

For a deeper theoretical discussion on Wavelet analysis, see the practical guide by Torrence and Compo (1998).

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

from periodicity.core import TSeries
from periodicity.data import SpottedStar
from periodicity.timefrequency import WPS
```

```
plt.rc("lines", linewidth=1.0, linestyle="-", color="black")
plt.rc("font", family="sans-serif", weight="normal", size=12.0)
```

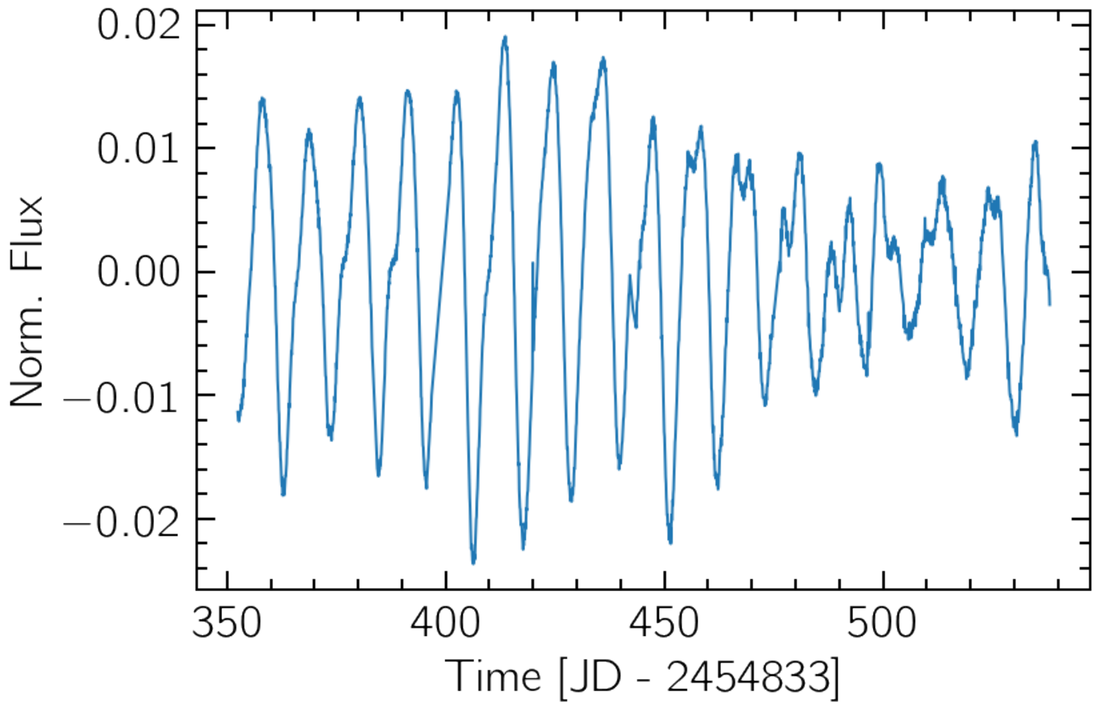
(continues on next page)

(continued from previous page)

```
plt.rc("text", color="black", usetex=True)
plt.rc("text.latex", preamble="\\usepackage{cmbright}")
plt.rc(
    "axes",
    edgecolor="black",
    facecolor="white",
    linewidth=1.0,
    grid=False,
    titlesize="x-large",
    labelsize="x-large",
    labelweight="normal",
    labelcolor="black",
)
plt.rc("axes.formatter", limits=(-4, 4))
plt.rc(("xtick", "ytick"), labelsize="x-large", direction="in")
plt.rc("xtick", top=True)
plt.rc("ytick", right=True)
plt.rc(("xtick.major", "ytick.major"), size=7, pad=6, width=1.0)
plt.rc(("xtick.minor", "ytick.minor"), size=4, pad=6, width=1.0, visible=True)
plt.rc("legend", numpoints=1, fontsize="x-large", shadow=False, frameon=False)
```

The data is taken from Kepler observations of the star KIC 9655172 from December 2009 to June 2010 (totaling six months or ~180 days). If we plot the data, we can visually detect a seasonality of ~10 days, with varying amplitude levels.

```
t, y, dy = SpottedStar()
sig = TSeries(t, y)

sig.plot()
plt.xlabel("Time [JD - 2454833]")
plt.ylabel("Norm. Flux");
```



Plotting the Wavelet Power Spectrum

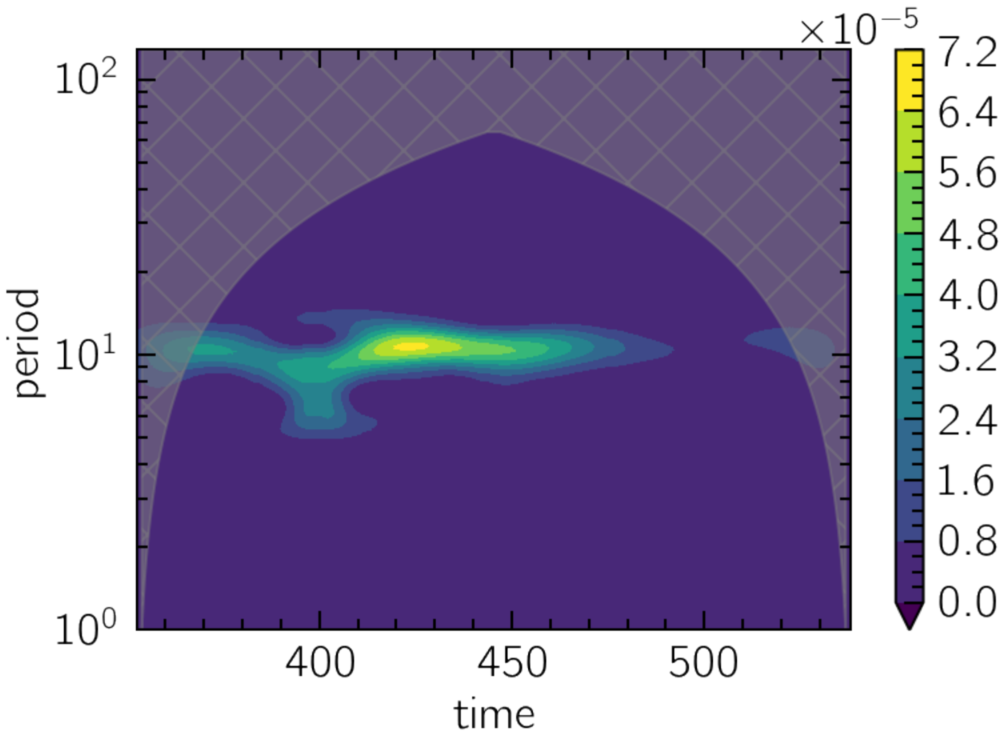
The creation of a new `WPS` object requires only a period grid on which to calculate the wavelet transform. A generally good advice is to use a log-uniform grid.

```
periods = np.logspace(0, 7, 1000, base=2)
wps = WPS(periods)
```

The created object can now be called with the input signal, returning a `TFSeries` which can be plotted and analysed with the usual tools.

More importantly, the `plot_coi` method will fill the area around the Cone of Influence, where edge effects become relevant.

```
spectrum = wps(sig)

spectrum.contourf(y="period", extend="min", levels=10)
wps.plot_coi(hatch="x", color="grey", alpha=0.5)
plt.yscale("log");
```

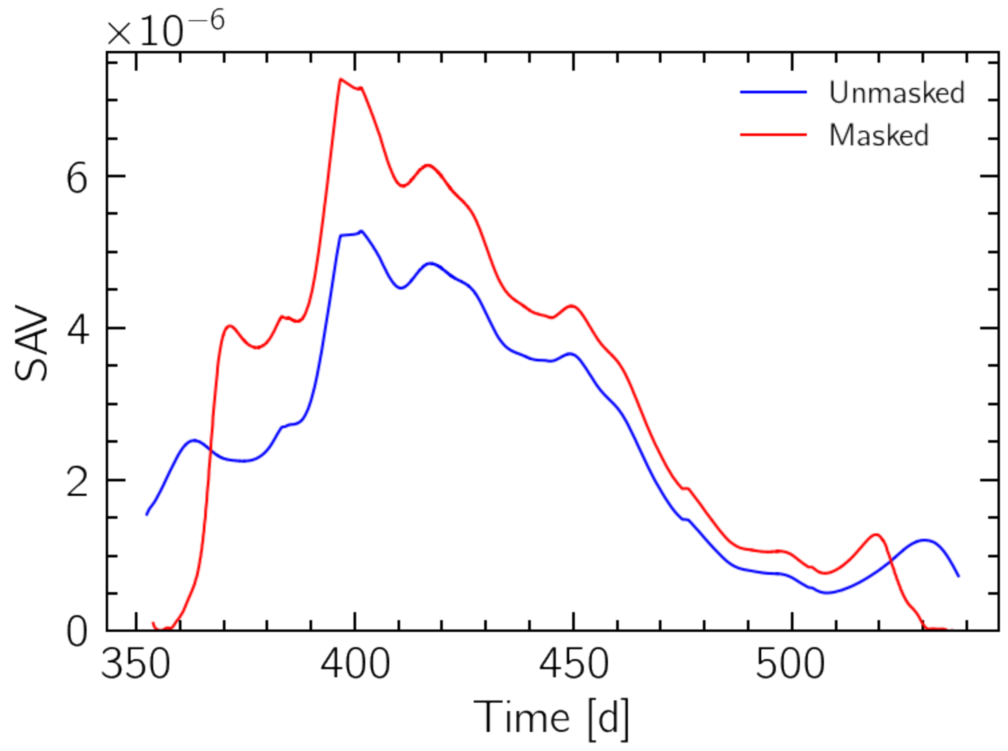


The Scale Average Variance (SAV)

The scale-average of the WPS, known as the SAV, is the projection of the representation on the time axis. It can give a good approximation on the variations of the signal variance with time.

The COI can also be used to mask out values on the edges.

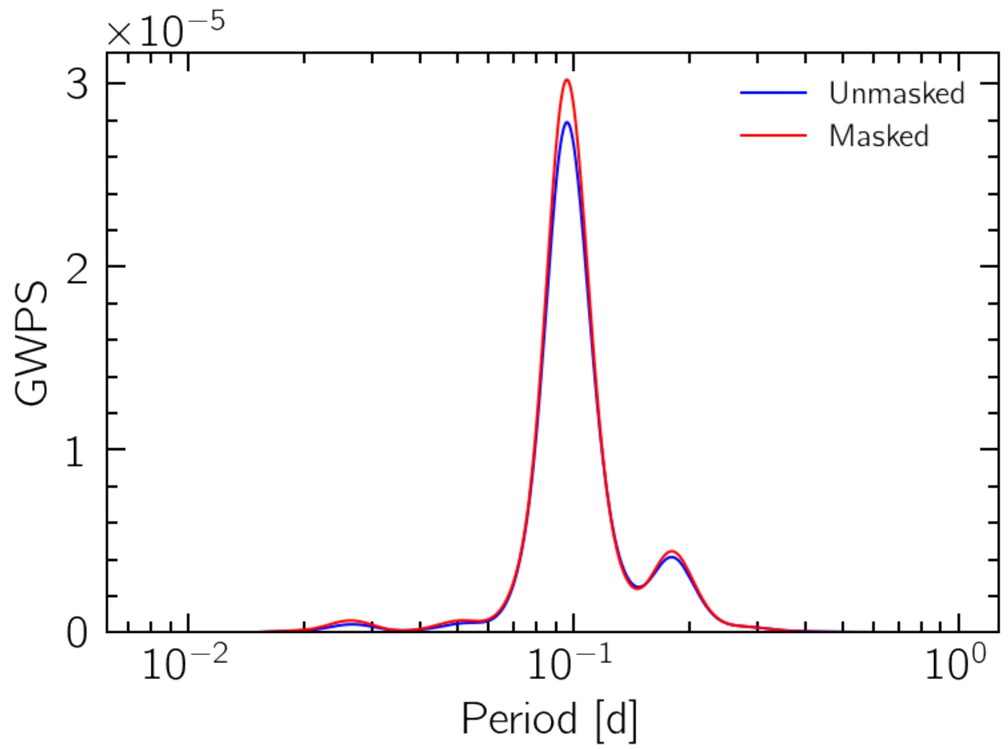
```
wps.sav().plot("b", label="Unmasked")
wps.masked_sav().plot("r", label="Masked")
plt.ylim(0)
plt.xlabel("Time [d]")
plt.ylabel("SAV")
plt.legend(fontsize=12);
```



The Global Wavelet Power Spectrum

Similarly, by performing the time-average of the WPS, a global power spectrum can be obtained. This can be useful to determine the main persistent periodicity on the signal. Once again, the COI can be used to mask-out edge effects.

```
wps.gwps().plot("b", label="Unmasked")
wps.masked_gwps().plot("r", label="Masked")
plt.ylim(0)
plt.xlabel("Period [d]")
plt.ylabel("GWPS")
plt.xscale("log")
plt.legend(fontsize=12);
```



```
print(wps.gwps().period_at_highest_peak, wps.masked_gwps().period_at_highest_peak)
```

```
10.391826740281603 10.391826740281603
```

```
%matplotlib inline
%config InlineBackend.figure_format = "retina"
```

2.1.9 Hilbert-Huang Analysis

This tutorial will cover some steps involved in decomposing and analysing non-stationary signals using the Hilbert-Huang Transform (HHT) using `periodicity`. While usually based on the traditional Empirical Mode Decomposition (EMD), the HHT can be applied to any decomposition of the signal in AM-FM modes; for this tutorial, we will illustrate how to combine ideas from recent improvements on the EMD in our custom HHT object.

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

from periodicity.core import TSeries
from periodicity.data import SunSpots
from periodicity.decomposition import CEEMDAN
from periodicity.timefrequency import HHT
```

```
plt.rc("lines", linewidth=1.0, linestyle="-", color="black")
plt.rc("font", family="sans-serif", weight="normal", size=12.0)
plt.rc("text", color="black", usetex=True)
plt.rc("text.latex", preamble="\\usepackage{cmbright}")
plt.rc(
```

(continues on next page)

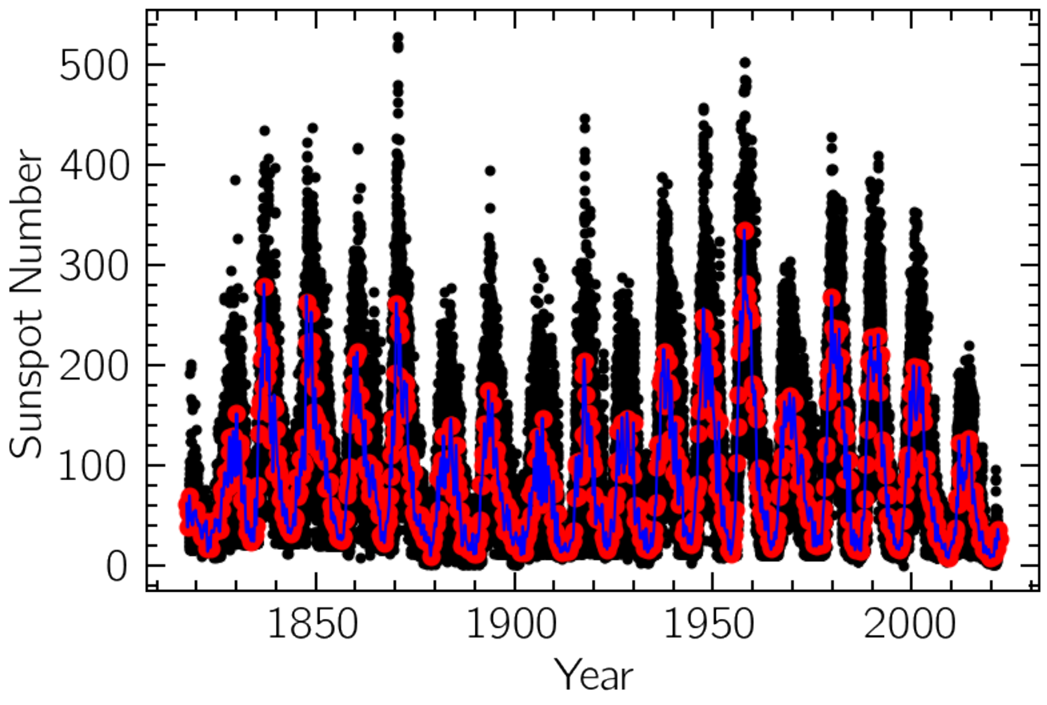
(continued from previous page)

```
    "axes",
    edgecolor="black",
    facecolor="white",
    linewidth=1.0,
    grid=False,
    titlesize="x-large",
    labelsize="x-large",
    labelweight="normal",
    labelcolor="black",
)
plt.rc("axes.formatter", limits=(-4, 4))
plt.rc(("xtick", "ytick"), labelsize="x-large", direction="in")
plt.rc("xtick", top=True)
plt.rc("ytick", right=True)
plt.rc(("xtick.major", "ytick.major"), size=7, pad=6, width=1.0)
plt.rc(("xtick.minor", "ytick.minor"), size=4, pad=6, width=1.0, visible=True)
plt.rc("legend", numpoints=1, fontsize="x-large", shadow=False, frameon=False)
```

Preprocessing

We will be analysing the Sunspot dataset, included in the `periodicity.data` module. These are daily measurements of the sunspot number from January 1818 up to June 2021, just over 200 years of observations. First we're going to perform some preprocessing, cleaning the bad measurements and resampling at a uniform 10-day cadence using a regularized cubic spline interpolation. This is important since our decomposition algorithm requires uniformly-sampled data.

```
t, y = SunSpots()
# bad measurements are marked with -1
sun = TSeries(t, y)[y > 0]
sun.plot("k.")
# bin to 90-day cadence to attenuate high-frequency noise
# pad in order to avoid edge effects when interpolating
sun = sun.downsample(dt=90/365).pad(2, mode="reflect", reflect_type="odd")
sun.plot("ro")
# resample at 10-day intervals using a noise estimate to regularize the spline␣
↪interpolation
new_time = np.arange(sun.time[0], sun.time[-1], 10/365)
sun = sun.interp(new_time, method="spline", s=sun.estimate_noise())
sun.plot("b")
plt.xlabel("Year")
plt.ylabel("Sunspot Number");
```



Modifying the HHT

To define a HHT object, we can use any custom-made function that accepts a `TSeries` and returns a list of modes. Here we define a Complete Ensemble EMD with Adaptive Noise (see Colominas et al. 2014) using the CEEMDAN object from `periodicity.decomposition` and combine it with the postprocessing algorithm from Wu and Huang (2009). This results in more well-behaved modes, with less mixing than with a standard EMD.

```
def my_emd(signal):
    emd = CEEMDAN(ensemble_size=50, random_seed=42, cores=4)
    emd(signal, progress=True)
    emd.postprocessing()
    return emd.c_modes

frequency = np.arange(0.004, 4.0, 0.001)
hht = HHT(frequency, emd=my_emd, method="NHT", norm_type="lmd", smooth_width=5)
```

Calling the resulting object on our solar time series performs the given decomposition and also calculates the instantaneous frequencies and amplitudes using a method defined by the `method` keyword (in this case, the Normalized Hilbert Transform).

```
spectrogram = hht(sun)
```

```
White noise: done
Mode #1: done
Mode #2: done
Mode #3: done
Mode #4: done
Mode #5: done
Mode #6: done
Mode #7: done
```

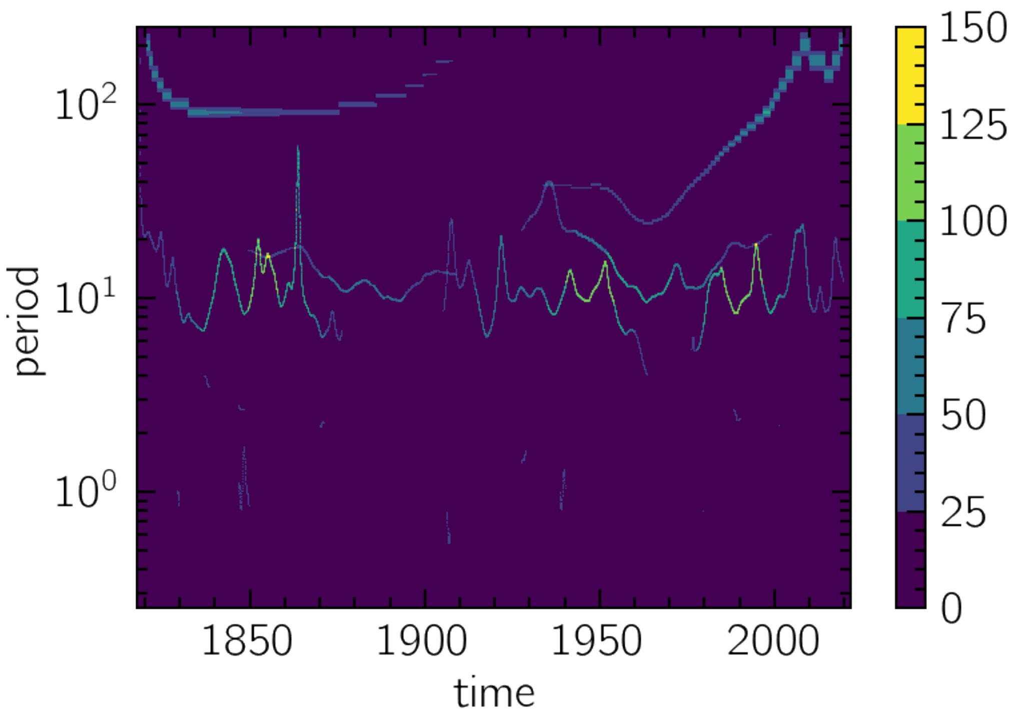
(continues on next page)

(continued from previous page)

```
Mode #8: done
Mode #9: done
```

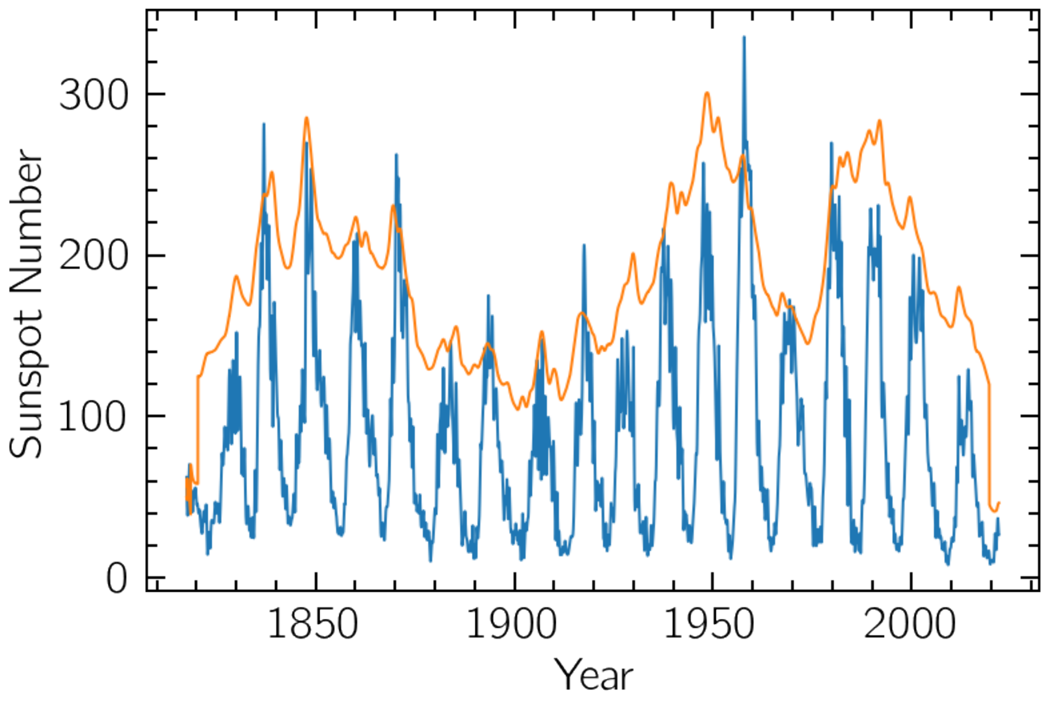
The resulting time-frequency representation can be visualized using standard plotting utilities built-in the `TFSeries` object. Notice how the frequency resolution can get arbitrarily small, circumventing the Heisenberg uncertainty principle that limits linear models (like the ones based on wavelets or on Fourier transforms).

```
spectrogram.contourf(y="period")
plt.yscale('log');
```



We can project the result on the time axis and get a decent approximation to the global amplitude variations of the signal:

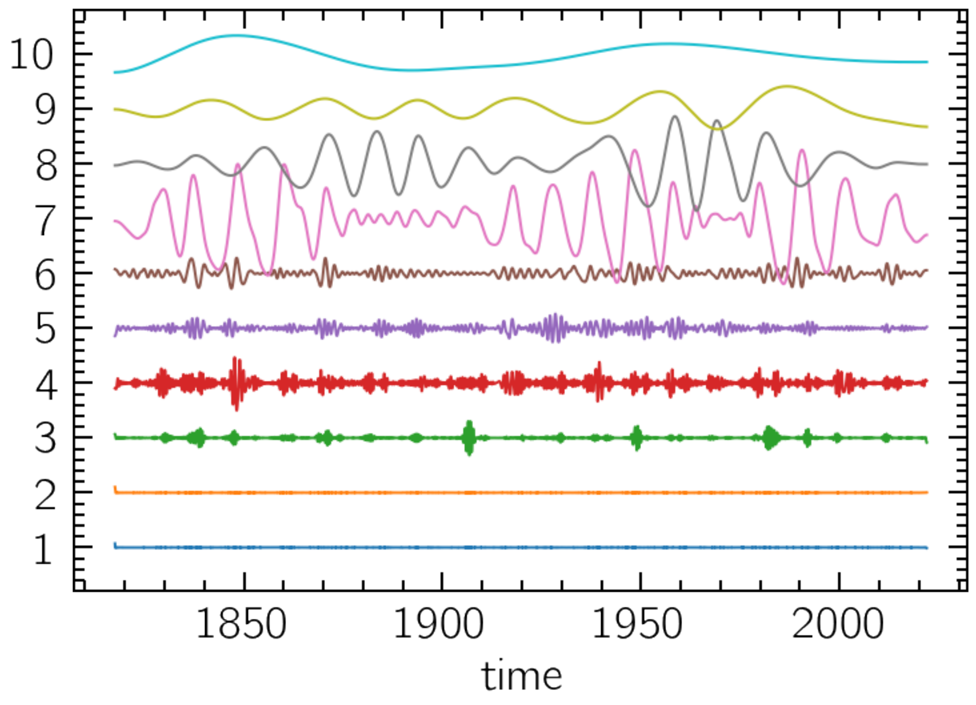
```
sun.plot()
spectrogram.sum("frequency").plot()
plt.xlabel("Year")
plt.ylabel("Sunspot Number");
```



Individual modes and the Hilbert Spectrum

The modes resulting from the decomposition can also be individually accessed, if any further processing is required. Here we'll visualize their relative contributions to the original series:

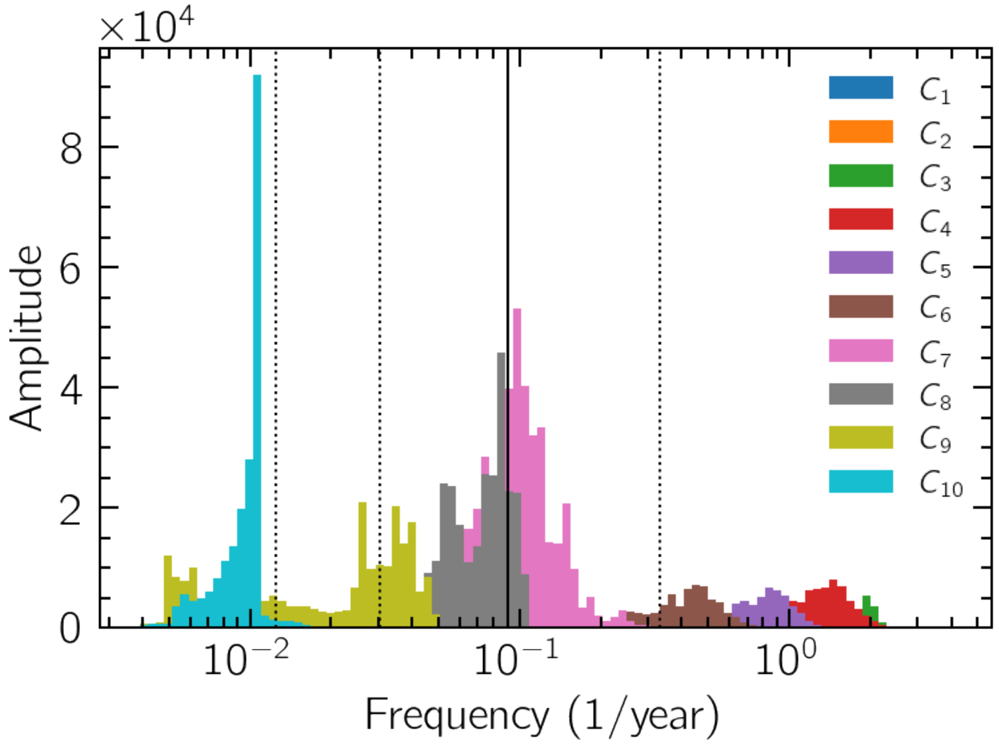
```
for i in range(len(hht.modes)):
    (100 * i + hht.modes[i]).plot()

plt.yticks(np.arange(0, 1000, 100), np.arange(1, 11))
plt.ylim(-80, 980);
```



Notice that, as expected, the noisy high-frequency modes are sifted first, and the long-term trends are last. This filter-bank-like behaviour of the decomposition can be better observed by determining each component's frequency distribution (weighed by their corresponding amplitudes) in a sort of Global Hilbert Spectrum.

We'll plot them as a collection of histograms from the `instant_fs` and `instant_as` attributes:

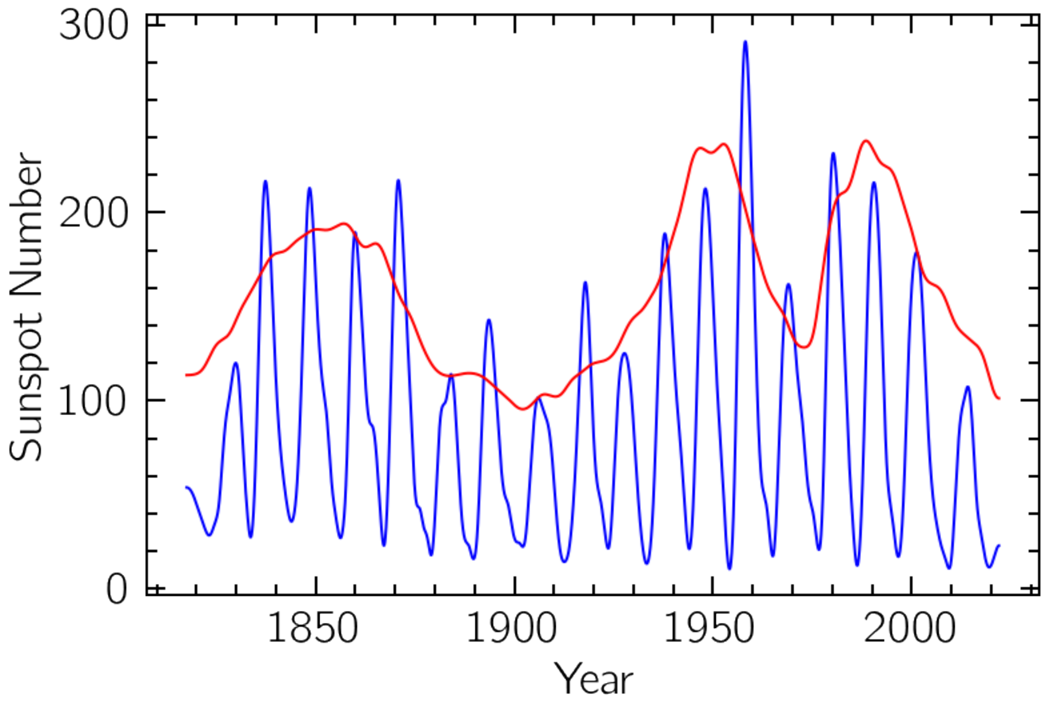
```
bins = np.logspace(-8, 2, 101, base=2)
for i in range(len(hht.instant_fs)):
    plt.hist(
        hht.instant_fs[i].values,
        weights=hht.instant_as[i].values,
        bins=bins,
        label=f"$C_{{{i+1}}}$",
    )
plt.xscale("log")
plt.axvline(1/3, ls=":")    # The quasi-biennial/quasi-triennial oscillations (C6)
plt.axvline(1/11)           # The Schwabe cycle (C7+C8)
plt.axvline(1/33, ls=":")   # Thrice the Schwabe cycle's period (C9)
plt.axvline(1/80, ls=":")   # The century-long Gleissberg cycle (C10)
plt.xlabel("Frequency (1/year)")
plt.ylabel("Amplitude")
plt.legend(fontsize=12);
```



In the figure above, the solid vertical line corresponds to a 11-year period, which is the main periodicity from the notorious Schwabe cycle of the magnetic activity of the Sun, and the one we were expecting to find in our data. We also highlight with dotted lines other periodicities known to occur in the Sun, especially the short biennial/triennial cycles (with very low relative contributions) and the ~80-year-long Gleissberg cycle.

We can use a subset of these modes to reconstruct a filtered version of the original signal. For example, we can discard the highest frequencies modes (more likely to be contaminated by noise) and focus on the 7th mode onwards:

```
residue = hht.signal - sum(hht.modes)  # monotonic residue corresponding to the␣
↪global mean/trend
(sum(hht.modes[6:10]) + residue).plot(color="b")  # low-pass filtered version of the␣
↪original data
sum(hht.instant_as[6:10]).plot(color="r")  # corresponding amplitude variations
plt.xlabel("Year")
plt.ylabel("Sunspot Number");
```



Notice how the amplitude variations of the data also seem to present a seasonality very similar to the Gleissberg cycle. Periodicity analysis on the amplitude of signals can sometimes be an interesting area of investigation, and the HHT can be used as a way to extract this information.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] Press W.H. and Rybicki, G.B, “Fast algorithm for spectral analysis of unevenly sampled data”. ApJ 1:338, p277, 1989

[2] M. Zechmeister and M. Kurster, A&A 496, 577-584 (2009)

[3] W. Press et al, Numerical Recipes in C (2002)

PYTHON MODULE INDEX

INDEX