stingray Documentation

Release 0.1.dev408

Stingray Developers

January 19, 2017

Contents

Contents:

Part I

Stingray API

Library of Time Series Methods For Astronomical X-ray Data.

class `stingray.AveragedCrossspectrum`(*lc1*, *lc2*, *segment_size*, *norm='none'*)

Make an averaged cross spectrum from a light curve by segmenting two light curves, Fourier-transforming each segment and then averaging the resulting cross spectra.

Parameters

lc1: lightcurve.Lightcurve object OR

iterable of lightcurve.Lightcurve objects One light curve data to be Fourier-transformed. This is the band of interest or channel of interest.

lc2: lightcurve.Lightcurve object OR

iterable of lightcurve.Lightcurve objects Second light curve data to be Fourier-transformed. This is the reference band.

segment_size: float

The size of each segment to average. Note that if the total duration of each Lightcurve object in lc1 or lc2 is not an integer multiple of the segment_size, then any fraction left-over at the end of the time series will be lost. Otherwise you introduce artefacts.

norm: {'frac', 'abs', 'leahy', 'none'}, default 'none'

The normalization of the (real part of the) cross spectrum.

Attributes

freq: numpy.ndarray	The array of mid-bin frequencies that the Fourier transform samples
power: numpy.ndarray	The array of cross spectra
df: float	The frequency resolution
m: int	The number of averaged cross spectra
n: int	The number of time bins per segment of light curve?
nphots1: float	The total number of photons in the first (interest) light curve
nphots2: float	The total number of photons in the second (reference) light curve

`coherence()`

Compute an averaged Coherence function of cross spectrum by computing coherence function of each segment and averaging them. The return type is a tuple with first element as the coherence function and the second element as the corresponding uncertainty[1] associated with it.

Note

[The uncertainty in coherence function is strictly valid for] Gaussian statistics only.

Returns

tuple : tuple of np.ndarray

Tuple of coherence function and uncertainty.

References

[R1]

class `stingray.AveragedPowerspectrum`(*lc*, *segment_size*, *norm='frac'*)

Make an averaged periodogram from a light curve by segmenting the light curve, Fourier-transforming each segment and then averaging the resulting periodograms.

Parameters

lc: lightcurve.Lightcurve object OR

iterable of lightcurve.Lightcurve objects The light curve data to be Fourier-transformed.

segment_size: float

The size of each segment to average. Note that if the total duration of each Lightcurve object in lc is not an integer multiple of the segment_size, then any fraction left-over at the end of the time series will be lost.

norm: {"leahy" | "rms"}, optional, default "rms"

The normaliation of the periodogram to be used. Options are "leahy" or "rms", default is "rms".

Attributes

norm: {"leahy" | "rms"}	the normalization of the periodogram
freq: numpy.ndarray	The array of mid-bin frequencies that the Fourier transform samples
power: numpy.ndarray	The array of normalized squared absolute values of Fourier amplitudes
df: float	The frequency resolution
m: int	The number of averaged periodograms
n: int	The number of data points in the light curve
nphots: float	The total number of photons in the light curve

class `stingray.`**`Crossspectrum`**(*lc1=None*, *lc2=None*, *norm='none'*)

Make a cross spectrum from a (binned) light curve. You can also make an empty Crossspectrum object to populate with your own fourier-transformed data (this can sometimes be useful when making binned periodograms).

Parameters

lc1: lightcurve.Lightcurve object, optional, default None

The first light curve data for the channel/band of interest.

lc2: lightcurve.Lightcurve object, optional, default None

The light curve data for the reference band.

norm: {'frac', 'abs', 'leahy', 'none'}, default 'none'

The normalization of the (real part of the) cross spectrum.

Attributes

freq: numpy.ndarray	The array of mid-bin frequencies that the Fourier transform samples
power: numpy.ndarray	The array of cross spectra (complex numbers)
df: float	The frequency resolution
m: int	The number of averaged cross-spectra amplitudes in each bin.
n: int	The number of data points/time bins in one segment of the light curves.
nphots1: float	The total number of photons in light curve 1
nphots2: float	The total number of photons in light curve 2

`coherence()`

Compute Coherence function of the cross spectrum. Coherence is a Fourier frequency dependent measure of the linear correlation between time series measured simultaneously in two energy channels.

Returns

coh : numpy.ndarray

Coherence function

References

[R2]

rebin(*df*, *method='mean'*)

Rebin the cross spectrum to a new frequency resolution df.

Parameters

df: float

The new frequency resolution

Returns

bin_cs = Crossspectrum object

The newly binned cross spectrum

rebin_log(*f=0.01*)

Logarithmic rebin of the periodogram. The new frequency depends on the previous frequency modified by a factor f:

dnu_j = dnu_{j-1}*(1+f)

Parameters

f: float, optional, default 0.01

parameter that steers the frequency resolution

Returns

binfreq: numpy.ndarray

the binned frequencies

binpower: numpy.ndarray

the binned powers

nsamples: numpy.ndarray

the samples of the original periodogramincluded in each frequency bin

time_lag()

Calculate the fourier time lag of the cross spectrum. The time lag is calculate using the center of the frequency bins.

class `stingray.`**Lightcurve**(*time*, *counts*, *input_counts=True*)

Make a light curve object from an array of time stamps and an array of counts.

Parameters

time: iterable

A list or array of time stamps for a light curve

counts: iterable, optional, default None

A list or array of the counts in each bin corresponding to the bins defined in `time` (note: **not** the count rate, i.e. counts/second, but the counts/bin).

input_counts: bool, optional, default True

If True, the code assumes that the input data in 'counts' is in units of counts/bin. If False, it assumes the data in 'counts' is in counts/second.

Attributes

time: numpy.ndarray	The array of midpoints of time bins
counts: numpy.ndarray	The counts per bin corresponding to the bins in `time`.
countrate: numpy.ndarray	The counts per second in each of the bins defined in `time`.
ncounts: int	The number of data points in the light curve.
dt: float	The time resolution of the light curve.
tseg: float	The total duration of the light curve.
tstart: float	The start time of the light curve.

`join`(*other*)

Join two lightcurves into a single object.

The new Lightcurve object will contain time stamps from both the objects. The count per bin in the resulting object will be the individual count per bin, or the average in case of overlapping time arrays of both lightcurve objects.

Note : Time array of both lightcurves should not overlap each other.

Parameters

other : Lightcurve object

The other Lightcurve object which is supposed to be joined with.

Returns

lc_new : Lightcurve object

The resulting lightcurve object.

static **`make_lightcurve`**(*toa*, *dt*, *tseg=None*, *tstart=None*)

Make a light curve out of photon arrival times.

Parameters

toa: iterable

list of photon arrival times

dt: float

time resolution of the light curve (the bin width)

tseg: float, optional, default None

The total duration of the light curve. If this is `None`, then the total duration of the light curve will be the interval between the arrival between the first and the last photon in `toa`.

Note: If tseg is not divisible by dt (i.e. if tseg/dt is not an integer number), then the last fractional bin will be dropped!

tstart: float, optional, default None

The start time of the light curve. If this is None, the arrival time of the first photon will be used as the start time of the light curve.

Returns

lc: `Lightcurve` object

A light curve object with the binned light curve

`plot`(*labels=None*, *axis=None*, *title=None*, *marker='-'*, *save=False*, *filename=None*)
Plot the Lightcurve using Matplotlib.

Plot the Lightcurve object on a graph `self.time` on x-axis and `self.counts` on y-axis.

Parameters
labels : iterable, default None

A list of tuple with xlabel and ylabel as strings.

axis : list, tuple, string, default None

Parameter to set axis properties of Matplotlib figure. For example it can be a list like `[xmin, xmax, ymin, ymax]` or any other acceptable argument for `matplotlib.pyplot.axis()` function.

title : str, default None

The title of the plot.

marker : str, default '-'

Line style and color of the plot. Line styles and colors are combined in a single format string, as in `'bo'` for blue circles. See `matplotlib.pyplot.plot` for more options.

save : boolean, optional (default=False)

If True, save the figure with specified filename.

filename : str

File name of the image to save. Depends on the boolean `save`.

`read`(*filename*, *format_='pickle'*)
Imports LightCurve object.

Parameters
filename: str

Name of the LightCurve object to be read.

format_: str

Available options are 'pickle', 'hdf5', 'ascii'

Returns
If **format_** is 'ascii': astropy.table is returned.

If **format_** is 'hdf5': dictionary with key-value pairs is returned.

If **format_** is 'pickle': class object is set.

`rebin_lightcurve`(*dt_new*, *method='sum'*)
Rebin the light curve to a new time resolution. While the new resolution need not be an integer multiple of the previous time resolution, be aware that if it is not, the last bin will be cut off by the fraction left over by the integer division.

Parameters
dt_new: float

The new time resolution of the light curve. Must be larger than the time resolution of the old light curve!

method: {"sum" | "mean" | "average"}, optional, default "sum"

This keyword argument sets whether the counts in the new bins should be summed or averaged.

Returns

lc_new: `Lightcurve` object

The `Lightcurve` object with the new, binned light curve.

sort(*reverse=False*)

Sort a Lightcurve object in accordance with its counts array.

A Lightcurve can be sorted in either increasing or decreasing order using this method. The counts array gets sorted and the time array is changed accordingly.

Parameters

reverse : boolean, default False

If True then the object is sorted in reverse order.

Returns

lc_new: `Lightcurve` object

The `Lightcurve` object with truncated time and counts arrays.

truncate(*start=0*, *stop=None*, *method='index'*)

Truncate a Lightcurve object from points on the time array.

This method allows the truncation of a Lightcurve object and returns a new light curve.

Parameters

start : int, default 0

Index of the starting point of the truncation.

stop : int, default None

Index of the ending point (exclusive) of the truncation. If no value of stop is set, then points including the last point in the counts array are taken in count.

method : {"index" | "time"}, optional, default "index"

Type of the start and stop values. If set to "index" then the values are treated as indices of the counts array, or if set to "time", the values are treated as actual time values.

Returns

lc_new: `Lightcurve` object

The `Lightcurve` object with truncated time and counts arrays.

write(*filename*, *format_='pickle'*, ***kwargs*)

Exports LightCurve object.

Parameters

filename: str

Name of the LightCurve object to be created.

format_: str

Available options are 'pickle', 'hdf5', 'ascii'

class stingray.**Powerspectrum**(*lc=None*, *norm='frac'*)

Make a Periodogram (power spectrum) from a (binned) light curve. Periodograms can be Leahy normalized or fractional rms normalized. You can also make an empty Periodogram object to populate with your own fourier-transformed data (this can sometimes be useful when making binned periodograms).

Parameters

lc: lightcurve.Lightcurve object, optional, default None

The light curve data to be Fourier-transformed.

norm: {"leahy" | "rms"}, optional, default "rms"

The normaliation of the periodogram to be used. Options are "leahy" or "rms", default is "rms".

Attributes

norm: {"leahy" | "rms"}	the normalization of the periodogram
freq: numpy.ndarray	The array of mid-bin frequencies that the Fourier transform samples
power: numpy.ndarray	The array of normalized squared absolute values of Fourier amplitudes
df: float	The frequency resolution
m: int	The number of averaged powers in each bin
n: int	The number of data points in the light curve
nphots: float	The total number of photons in the light curve

`classical_significances`(*threshold=1*, *trial_correction=False*)

Compute the classical significances for the powers in the power spectrum, assuming an underlying noise distribution that follows a chi-square distributions with 2M degrees of freedom, where M is the number of powers averaged in each bin.

Note that this function will *only* produce correct results when the following underlying assumptions are fulfilled: (1) The power spectrum is Leahy-normalized (2) There is no source of variability in the data other than the periodic signal to be determined with this method. This is important! If there are other sources of (aperiodic) variability in the data, this method will *not* produce correct results, but instead produce a large number of spurious false positive detections! (3) There are no significant instrumental effects changing the statistical distribution of the powers (e.g. pile-up or dead time)

By default, the method produces (index,p-values) for all powers in the power spectrum, where index is the numerical index of the power in question. If a `threshold` is set, then only powers with p-values *below* that threshold with their respective indices. If `trial_correction` is set to True, then the threshold will be corrected for the number of trials (frequencies) in the power spectrum before being used.

Parameters

threshold : float

The threshold to be used when reporting p-values of potentially significant powers. Must be between 0 and 1. Default is 1 (all p-values will be reported).

trial_correction : bool

A Boolean flag that sets whether the `threshold` will be correted by the number of frequencies before being applied. This decreases the threshold (p-values need to be lower to count as significant). Default is False (report all powers) though for any application where `threshold` is set to something meaningful, this should also be applied!

Returns

pvals : iterable

A list of (index, p-value) tuples for all powers that have p-values lower than the threshold specified in `threshold`.

`compute_rms`(*min_freq*, *max_freq*)

Compute the fractional rms amplitude in the periodgram between two frequencies.

Parameters

min_freq: float

The lower frequency bound for the calculation

max_freq: float

The upper frequency bound for the calculation

Returns

rms: float

The fractional rms amplitude contained between min_freq and max_freq

`rebin`(*df*, *method='mean'*)

exception `stingray.UnrecognizedMethod`

`stingray.assign_value_if_none`(*value*, *default*)

`stingray.coherence`(*lc1*, *lc2*)

Estimate coherence function of two light curves.

Parameters

lc1: lightcurve.Lightcurve object

The first light curve data for the channel of interest.

lc2: lightcurve.Lightcurve object

The light curve data for reference band

Returns

coh : np.ndarray

Coherence function

`stingray.contiguous_regions`(*condition*)

Find contiguous True regions of the boolean array "condition".

Return a 2D array where the first column is the start index of the region and the second column is the end index.

Parameters

condition : boolean array

Returns

idx : [[i0_0, i0_1], [i1_0, i1_1], ...]

A list of integer couples, with the start and end of each True blocks in the original array

Notes

From http://stackoverflow.com/questions/4494404/find-large-number-of-consecutive-values-fulfilling-condition-in-a-numpy-array

`stingray.is_iterable`(*stuff*)

Test if stuff is an iterable.

`stingray.is_string`(*s*)

Portable function to answer this question.

`stingray.jit`(*fun*)

`stingray.look_for_array_in_array`(*array1*, *array2*) `stingray.`**`optimal_bin_time`**(*fftlen*, *tbin*)

Vary slightly the bin time to have a power of two number of bins.

Given an FFT length and a proposed bin time, return a bin time slightly shorter than the original, that will produce a power-of-two number of FFT bins.

`stingray.`**`order_list_of_arrays`**(*data*, *order*)

`stingray.`**`rebin_data`**(*x*, *y*, *dx_new*, *method=u'sum'*)

Rebin some data to an arbitrary new data resolution. Either sum the data points in the new bins or average them.

Parameters

x: iterable

The dependent variable with some resolution dx_old = x[1]-x[0]

y: iterable

The independent variable to be binned

dx_new: float

The new resolution of the dependent variable x

method: {"sum" | "average" | "mean"}, optional, default "sum"

The method to be used in binning. Either sum the samples y in each new bin of x, or take the arithmetic mean.

Returns

xbin: numpy.ndarray

The midpoints of the new bins in x

ybin: numpy.ndarray

The binned quantity y

`stingray.`**`simon`**(*message*, ***kwargs*)

The Statistical Interpretation MONitor.

A warning system designed to always remind the user that Simon is watching him/her.

Parameters

message : string

The message that is thrown

kwargs : dict

The rest of the arguments that are passed to warnings.warn

`stingray.`**`test`**(*package=None*, *test_path=None*, *args=None*, *plugins=None*, *verbose=False*, *pastebin=None*, *remote_data=False*, *pep8=False*, *pdb=False*, *coverage=False*, *open_files=False*, ***kwargs*)

Run the tests using py.test. A proper set of arguments is constructed and passed to pytest.main.

Parameters

package : str, optional

The name of a specific package to test, e.g. 'io.fits' or 'utils'. If nothing is specified all default tests are run.

test_path : str, optional

Specify location to test by path. May be a single file or directory. Must be specified absolutely or relative to the calling directory.

args : str, optional

Additional arguments to be passed to pytest.main in the `args` keyword argument.

plugins : list, optional

Plugins to be passed to pytest.main in the `plugins` keyword argument.

verbose : bool, optional

Convenience option to turn on verbose output from py.test. Passing True is the same as specifying `'-v'` in `args`.

pastebin : {'failed','all',None}, optional

Convenience option for turning on py.test pastebin output. Set to `'failed'` to upload info for failed tests, or `'all'` to upload info for all tests.

remote_data : bool, optional

Controls whether to run tests marked with @remote_data. These tests use online data and are not run by default. Set to True to run these tests.

pep8 : bool, optional

Turn on PEP8 checking via the pytest-pep8 plugin and disable normal tests. Same as specifying `'--pep8 -k pep8'` in `args`.

pdb : bool, optional

Turn on PDB post-mortem analysis for failing tests. Same as specifying `'--pdb'` in `args`.

coverage : bool, optional

Generate a test coverage report. The result will be placed in the directory htmlcov.

open_files : bool, optional

Fail when any tests leave files open. Off by default, because this adds extra run time to the test suite. Requires the psutil package.

parallel : int, optional

When provided, run the tests in parallel on the specified number of CPUs. If parallel is negative, it will use the all the cores on the machine. Requires the pytest-xdist plugin installed. Only available when using Astropy 0.3 or later.

kwargs

Any additional keywords passed into this function will be passed on to the astropy test runner. This allows use of test-related functionality implemented in later versions of astropy without explicitly updating the package template.

Part II

Indices and tables

- genindex
- modindex
- search

Bibliography

[R1] http://iopscience.iop.org/article/10.1086/310430/pdf

[R2] http://iopscience.iop.org/article/10.1086/310430/pdf

Python Module Index