# nicerlab Documentation

*Release 0.1*

**Peter Bult**

**Sep 24, 2017**

# Contents

The nicerlab package offers x-ray timing analysis tools for python. The goal of nicerlab is not to present a comprehensive timing library, but rather to give a framework of efficient processing methods and convenient data classes. These tools are intended to be used as building blocks for a customized python-based pipeline.

To achieve flexibility, the data objects of nicerlab are derived from numpy's array object. This allows, for instance, a *Light curve* object to be reshaped and binned as though it is simply a multi-dimensional `ndarray`.

The I/O operations of nicerlab use astropy to interface directly with fits formatted data.

To give a taste of what a nicerlab implementation might look like, the following code block reads a standard event list and computes a power spectrum for each good time interval.

```
import nicerlab as ni

filename = "my_file.fits"

events, gti = ni.read_events_and_gti(filename)

light_curves = [
    ni.make_light_curve(events, dt=1, tstart=t0, tstop=t1) for t0,t1 in gti
]

power_spectra = [
    ni.make_power_spectrum(lc, tseg=64, collect='avg') for lc in light_curves
]

for i,pds in enumerate(power_spectra):
    ni.io.write_pds(pds, "pds{}.fits".format(i))
```

**Warning:** While most of the current implementation will work as expected, nicerlab is still in development. Some

CHAPTER 1

# User Documentation

## Getting Started

### Obtaining nicerlab

The nicerlab package can be obtained through github. To obtain the source code use:

```
git clone https://github.com/peterbult/nicerlab.git
```

### Installing nicerlab

The recommended way to install nicerlab is to use pip. Navigate to the project folder, and from the command line:

```
make install
```

This will use pip to install the package, as well as all required dependencies. Alternatively, one can also use pip directly to install as:

```
pip install -r requirements.txt
pip install .
```

### Testing

Most nicerlab modules have associated test functions. These tests can be run using:

```
make test
```

or alternatively, using pytest directly:

```
pytest test/
```

## License

The nicerlab package is licensed using the MIT license.



### Contributing

Contributions through github are welcome.

# Data Structures

## Event list

An `Eventlist` represents a list of photon arrival times. This object is a `ndarray` subclass with the following added attributes:

| | |
|---|---|
| `tstart` | The start time of the observation |
| `tstop` | The stop time of the observation |
| `MJD` | The observation start time in MJD |

If a `tstart` or `tstop` time is not explicitly defined, the respective smallest and largest value in the array of arrival times will be used instead. The `MJD` of an `Eventlist` is zero unless explicitly defined.

Because an `Eventlist` is just an `ndarray` standard numpy methods can be used to manipulate the data. Consider the following example where we generate some random data and proceed to sort the `Eventlist` on time:

```
>>> import numpy as np
>>> import nicerlab as ni
>>> data = np.random.uniform(10,20,500)
>>> evt = ni.Eventlist(data, tstart=10, tstop=20)
>>> evt = np.sort(evt)
>>> evt.info()
Eventlist:
> counts......: 500
> exposure....: 10 seconds
> start/stop..: 10 / 20
> MJD.........: 0
```

**Note:** Event list data *should* generally be time ordered, however, this is not a strict requirement for nicerlab. Data contained in an Eventlist object is allowed to be out-of-order, and routines operating on an `Eventlist` will return the correct ouput either way.

## Light curve

A *Lightcurve* object is an array of event counts in equidistant time bins. This object is a *ndarray* subclass with the following added attributes:

| | |
|---|---|
| `dt` | Time bin width in seconds |
| `tstart` | The start time of the observation |
| `tstop` | The stop time of the observation |
| `MJD` | The observation start time in MJD |

The bin width will default to 1 if no value is provided. Like for the `Eventlist`, if the `tstart` or `tstop` time is not explicitly defined, the respective smallest and largest value in the array of arrival times will be used instead. The `MJD` of an `Eventlist` is zero unless explicitly defined.

In addition to the extra attributes, the `Lightcurve` object also has special member functions:

| | |
|---|---|
| `timespace` | Generate an array of times |
| `mjdspace` | Generate an array of MJD times |

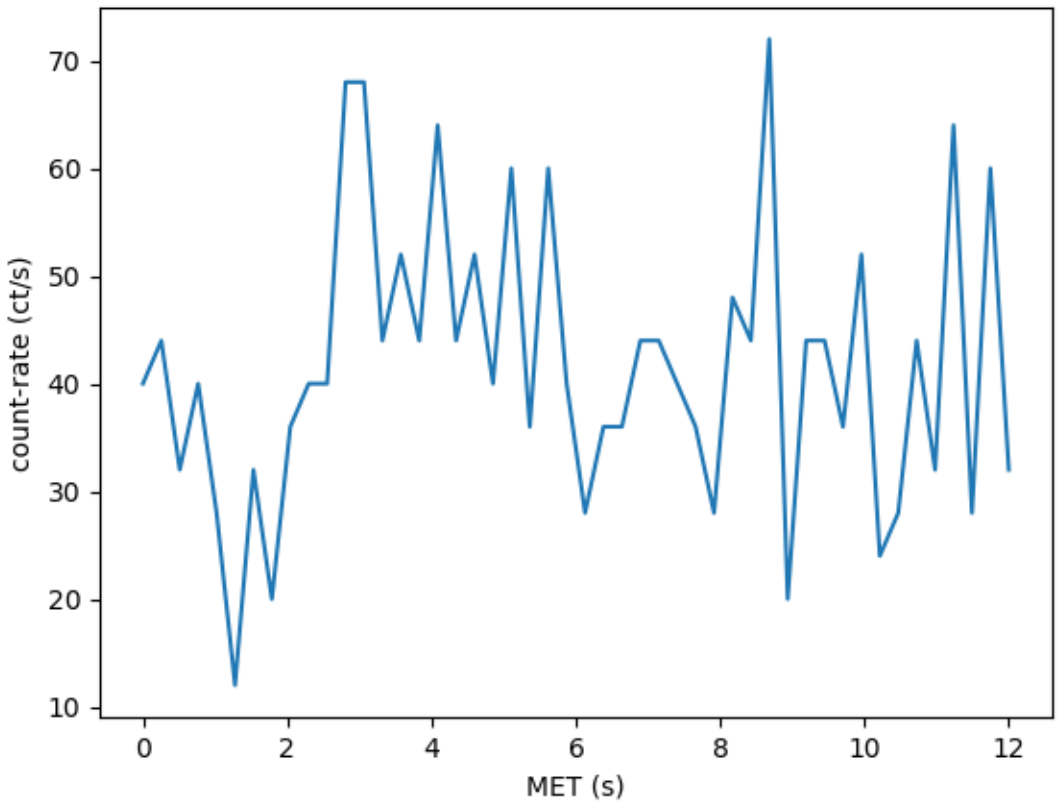
A `Lightcurve` can be constructed manually from any iterable list of counts as:

```
>>> from nicerlab.lightcurve import Lightcurve
>>> counts = [11, 12, 10, 8, 15, 12]
>>> lc = Lightcurve(counts, dt=2, tstart=0, mjd=55750)
>>> lc
Lightcurve([11, 12, 10, 8, 15, 12])
>>> lc.timespace()
array([ 0., 2., 4., 6., 8.])
>>> np.sum(lc.reshape(-1,2), axis=1)
Lightcurve([23, 18, 27])
```

**Warning:** The `Lightcurve` constructor will *not* check if the combination dt/tstart/tstop are self consistent. The array-size/bin-width always take precedence in determining the time axis of the light curve.

An example:

```
>>> import numpy as np
>>> import nicerlab as ni
>>> import matplotlib.pyplot as plt
>>> data = np.sort(np.random.uniform(0,12,500))
>>> evt = ni.Eventlist(data, tstart=0, tstop=12)
>>> lc = ni.make_light_curve(evt, dt=0.25) / 0.25
>>> plt.plot(lc.timespace(), lc);
>>> plt.xlabel('MET (s)');
>>> plt.ylabel('count-rate (ct/s)');
>>> plt.show()
```




## Power density spectrum

A *Powerspectrum* object is an array of powers densities. Note that the object does not care about what normalization is used. By default the *make_power_spectrum()* will use the Leahy normalization. The following attributes have been added to the object

| | |
|---|---|
| `df` | Frequency bin width in Hertz |
| `tstart` | The start time of the observation |
| `tstop` | The stop time of the observation |
| `MJD` | The observation start time in MJD |
| `stack` | The number of FFT segments in the spectrum |

Additionally it has a special member function

| | |
|---|---|
| `freqspace` | Generate an array of frequencies |

## Spectrum

A spectrum is an PI spectrum object.

# I/O

## Read

### Reading a generic fits file

Example:

```
>>> import nicerlab as ni
>>> filename = 'data_cl.evt'
>>> table, keys = ni.io.read_from_fits(
...                 filename, ext='EVENTS', cols=['TIME', 'PI'],
...                 keys=['TSTART', 'TSTOP', 'OBJECT'],
...                 as_table=True)
>>> keys
{'TSTART': 523327317.6109142, 'TSTOP': 523345136.3998488, 'OBJECT': 'TOO'}
>>> table
array([[  5.23327318e+08,   6.11000000e+02],
       [  5.23327319e+08,   6.23000000e+02],
       [  5.23327320e+08,   3.02000000e+02],
       ...,
       [  5.23345135e+08,   5.99000000e+02],
       [  5.23345136e+08,   2.89000000e+02],
       [  5.23345136e+08,   2.83000000e+02]])
```

Alternatively we could have also just used astropy as:

```
>>> from astropy.table import Table
>>> tb = Table.read(filename, format='fits', hdu=1)['TIME', 'PI']
>>> tb
<Table length=5946>
     TIME       PI
   float64    int32
------------- -----
523327317.829   611
523327319.446   623
523327320.199   302
          ...   ...
523345135.417   599
523345135.787   289
523345136.275   283
>>> keys = {k: tb.meta[k] for k in ['TSTART', 'TSTOP', 'OBJECT']}
>>> keys
{'TSTART': 523327317.6109142, 'TSTOP': 523345136.3998488, 'OBJECT': 'TOO'}
```

However, this will load all columns from the fits table into memory, which is sometimes unwieldy. In any case the `read_from_fits()` function is just a building block used to construct a set of higher level convience functions, and should seldomly be used directly.

### Reading event data

### Events only

If only the event arrival times are of interest these can be obtained by invoking the fits interface as:

```
>>> table = ni.io.read_from_fits(filename, ext='EVENTS', cols=['TIME'])
```

However, the `table` will be a 2-dimensional numpy array with only one defined column, hence:

```
>>> events = table[:,0]
```

What's more, we will actually want to construct an *Event list* object with the proper attributes set. To that end a convience function has been implemented that does all this for you:

```
>>> evt = ni.io.read_events(filename)
>>> evt.info()
Eventlist:
> counts......: 5946
> exposure....: 17818.788934648037 seconds
> start/stop..: 523327317.6109142 / 523345136.3998488
> MJD.........: 57967.02988188558
```

### Events and ...

For convience a number of combination functions are implemented that read event arrival times *and* some other datum from the fits table:

```
>>> evt, gti = ni.io.read_events_and_gti(filename)
>>> gti
array([[  5.23327318e+08,   5.23329114e+08],
       [  5.23333245e+08,   5.23334856e+08],
       [  5.23338946e+08,   5.23340281e+08],
       [  5.23344887e+08,   5.23345136e+08]])
>>>
>>> tb = ni.io.read_events_and_pi(filename)
>>> tb
array([[  5.23327318e+08,   6.11000000e+02],
       [  5.23327319e+08,   6.23000000e+02],
       [  5.23327320e+08,   3.02000000e+02],
       ...,
       [  5.23345135e+08,   5.99000000e+02],
       [  5.23345136e+08,   2.89000000e+02],
       [  5.23345136e+08,   2.83000000e+02]])
```

### Reading good time intervals

You can also read the GTI table only:

```
>>> table = ni.io.read_from_fits(filename, ext='GTI', cols=['START', 'STOP'])
```

which has its own convience function:

```
>>> gti = ni.io.read_gti(filename)
>>> gti
array([[  5.23327318e+08,   5.23329114e+08],
       [  5.23333245e+08,   5.23334856e+08],
       [  5.23338946e+08,   5.23340281e+08],
       [  5.23344887e+08,   5.23345136e+08]])
```

### Write

#### Write power spectrum

Write a single *Power density spectrum* object to a fits file

```
ni.io.write_pds(pds, "pds.fits")
```

If the pds has multiple rows, then this works too.

#### Write spectrum

Write a *Spectrum* object as an OGIP compatible fits file.

```
ni.io.write_spectrum(spec, "pi_spectrum.fits")
```

## Utilities

### Utils

The utils module implements a set of convenience functions that are used througout the library.

- find_first_of: This function will find the index of the first element in an array that exceeds a given threshold value. The function is used a number of times to dissect light curve or event data into discrete blocks.
- truncate: This function will

### GTI tools

The GTI Tools subpackages offers a set of functions that manage good time intervals. Available tools are

- durations – computes the GTI exposures
- trunctate – truncate a list of GTIs to a lower and/or upper boundary in time.
- good_to_bad – convert between good time and bad time
- bad_to_good – convert between good time and bad time
- merge – merge two GTI lists using and/or logic

### Ftools

The Ftools subpackage wraps around the heasoft FTOOLS and allows the user to call a number of operations on a fits file. These ftool operations are performed in a subproccess outside the python environment, but can be useful in preparing or cleaning the data for analysis.

# Scripts

## ni-lightcurve

The *ni-lightcurve* script is a simple nicerlab implementation for constructing light curves from event files. Basic usage is

```
ni-lightcurve evt.meta
```

By default *ni-lightcurve* will output count-rates as a function of MJD using a 16 second time resolution. All data is written to a single ASCII file named *lc.dat*, with a continuous block for each good time interval. Good time intervals are read from the fits file.

| | |
|---|---|
| `--dt` | Set the time resolution |
| `--clobber` | Overwrite the outputfile is it exists |
| `--met` | Format the time axis as mission-elapsed time |
| `--output` | Set the output file name |
| `--help` | Display usage instructions |

## gti-select

The *gti-select* tool gives an interactive method for shaping an event file's good time interval table. Basic usage is as

```
gti-select evt.meta --dt 1
```

| | |
|---|---|
| `--dt` | Set the time resolution |
| `--clobber` | Overwrite the eventfile internal GTI table |
| `--help` | Display usage instructions |

After initialization *gti-select* will iterate through the event files listed in the metafile. For each event file it will construct a light curve at the requested resolution and produce a plot with the current GTI's overlayed. One can then use the mouse to shape the good time intervals, and the terminal to move to the next event file. The available controls are

| | |
|---|---|
| left-mouse | Add bad time |
| right-mouse | Add good time |
| type 'undo' | Reset the current event file |
| type 'quit' | Quit without saving current file |
| type <enter> | Save and go to next file |

CHAPTER 2

# Index

- genindex
- modindex
- search