tlpipe Documentation

Release 0.1

Shifan Zuo

Dec 20, 2018

Contents

CHAPTER 1

Introduction

1.1 Introduction

- This is a Python project for the Tianlai data processing pipeline.
- This software can simply run as a single process on a single compute node, but for higher performance, it can also use the Message Passing Interface (MPI) to run data processing tasks distributed and parallelly on multiple computie nodes / supercomputers.
- It mainly focuses on the Tianlai cylinder array's data processing, though its basic framework and many tasks also work for Tianlai dish array's data, it does not specifically tuned for that currently.
- It can fulfill data processing tasks from reading data from raw observation data files, to RFI flagging, to relative and absolute calibration, to map-making, etc. It also provides some plotting tasks for data visualization.
- Currrently, foreground subtractiong and power spectrum estimation have not been implemented, but they will come maybe in the near future.

CHAPTER 2

Installation

2.1 Installation

2.1.1 Python version

The package works only with python 2 with version >= 2.75. Python 3 does not supported currently.

2.1.2 Prerequisites

For the installation and proper work of `tlpipe`, the following packages are required:

- h5py, Pythonic interface to the HDF5 binary data format;
- healpy, Healpix tools package for Python;
- pyephem, Basic astronomical computations for the Python;
- numpy, Base N-dimensional array package for Python;
- scipy, The fundamental package for scientific computing with Python;
- matplotlib, A python 2D plotting library;
- caput, Cluster Astronomical Python Utilities;
- cora, A package for simulating skies for 21cm Intensity Mapping;
- aipy, Astronomical Interferometry in PYthon;
- cython, An static compiler for Python, *optional;
- mpi4py, MPI for Python, *optional*.

Note: `tlpipe` can work without MPI support, in which case, only a single process is invoked, but in order to process large amounts of data in parallel and distributed manner, mpi4py is needed.

2.1.3 Installation guide

After you have successfully installed the prerequisites, do the following.

First clone this package

```
$ git clone https://github.com/TianlaiProject/tlpipe.git
```

Then change to the top directory of this package, install it by the usual methods, either the standard

```
$ python setup.py install [--user]
```

or to develop the package

```
$ python setup.py develop [--user]
```

It should also be installable directly with *pip* using the command

```
$ pip install [-e] git+https://github.com/TianlaiProject/tlpipe.git
```

Note: If you have installed `tlpipe` in the develop mode, you doesn't need to re-install the package every time after you have changed its (pure python) code. This is useful when you are the developer of the package or you want to do some development/contributions to the package.

CHAPTER 3

Tutorial

3.1 Tutorial

Note: This is intended to be a tutorial for the user of *tlpipe* package, who will just use the already presented tasks in the package to do some data analysis. For the developers of this package and those who want to do some developments/continuations, you may want to refer *Developer's guide* for a deeper introduction.

Contents

3.1.1 Prepare for the input pipe file

An input pipe file is actually a *python* script file, so it follows plain python syntax, but to emphasis that it is just used as an input pipe file for a data analysis pipeline, usually it is named with a suffix ".pipe" instead of ".py".

The only required argument to run a data analysis pipeline is the input pipe file, in which one specifies all tasks to be imported and excuted, all parameter settings for each task and also the excuting order (or flow controlling) of the pipeline.

Here we take the waterfall plot as an example to show how to write an input pipe file.

Non-iterative pipeline

1. Create and open an file named *plot_wf.pipe* (the name can be choosen arbitrary);
2. Speicify a variable *pipe_tasks* to hold analysis tasks that will be imported and excuted, and (**optionally**) a variable *pipe_outdir* to set the output directory (the default value is './output/'). You can set other parameters related to the pipeline according to your need or just use the default values. All paramters and their default values can be checked by method `show_params()`, **note**: all these parameters should be prepended with a prefix "pipe_";

```
# -*- mode: python; -*-

# input file for the analysis pipeline
# execute this pipeline by either command of the following two:
# tlpipe dir/to/plot_wf.pipe
# mpiexec -n N tlpipe dir/to/plot_wf.pipe


pipe_tasks = []
pipe_outdir = './output/'
pipe_logging = 'notset'
# pipe_logging = 'info'
```

3. Import tasks and set task parameters:
   (a) Import `Dispatch` to select data to plot;

```
import glob
data_dir = 'dir/to/data' # your data directory
files = sorted(glob.glob(data_dir+'/*.hdf5')) # all data files as a list


# data selection
from tlpipe.timestream import dispatch
pipe_tasks.append(dispatch.Dispatch)
### parameters for Dispatch
dp_input_files = files # data files as list
dp_freq_select = (500, 510) # frequency indices, from 500 to 510
dp_feed_select = [1, 2, 32, 33] # feed no. as a list
dp_out = 'dp'
```

   (b) Import `Detect` to find and mask noise source signal;

```
# find and mask noise source signal
from tlpipe.timestream import detect_ns
pipe_tasks.append(detect_ns.Detect)
```

(continues on next page)

(continued from previous page)

```
### parameters for Detect
dt_in = dp_out
# dt_feed = 1
dt_out = 'dt'
```

(c) Import `Plot` to plot;

```
from tlpipe.plot import plot_waterfall
pipe_tasks.append(plot_waterfall.Plot)
### parameters for Plot
pwf_in = dt_out
pwf_flag_ns = True # mask noise source signal
pwf_fig_name = 'waterfall/wf' # figure name to save
pwf_out = 'pwf'
```

The final input pipe file looks like `download`:

```
# -*- mode: python; -*-

# input file for the analysis pipeline
# execute this pipeline by either command of the following two:
# tlpipe dir/to/plot_wf.pipe
# mpiexec -n N tlpipe dir/to/plot_wf.pipe


pipe_tasks = []
pipe_outdir = './output/'
pipe_logging = 'notset'
# pipe_logging = 'info'


import glob
data_dir = 'dir/to/data' # your data directory
files = sorted(glob.glob(data_dir+'/*.hdf5')) # all data files as a list


# data selection
from tlpipe.timestream import dispatch
pipe_tasks.append(dispatch.Dispatch)
### parameters for Dispatch
dp_input_files = files # data files as list
dp_freq_select = (500, 510) # frequency indices, from 500 to 510
dp_feed_select = [1, 2, 32, 33] # feed no. as a list
dp_out = 'dp'

# find and mask noise source signal
from tlpipe.timestream import detect_ns
pipe_tasks.append(detect_ns.Detect)
### parameters for Detect
dt_in = dp_out
# dt_feed = 1
dt_out = 'dt'

# plot waterfall of selected data
from tlpipe.plot import plot_waterfall
pipe_tasks.append(plot_waterfall.Plot)
### parameters for Plot
```

(continues on next page)

(continued from previous page)

```
pwf_in = dt_out
pwf_flag_ns = True # mask noise source signal
pwf_fig_name = 'waterfall/wf' # figure name to save
pwf_out = 'pwf'
```

Note:

1. To show all pipeline related parameters and their default values, you can do:

```
>>> from tlpipe.pipeline import pipeline
>>> pipeline.Manager.prefix
'pipe_'
>>> pipeline.Manager.show_params()
Parameters of Manager:
copy:  True
tasks:  []
logging:  info
flush:  False
timing:  False
overwrite:  False
outdir:  output/
```

2. Each imported task should be appended into the list *pipe_tasks* in order to be excuted by the pipeline;

3. Each task's paramters should be prepended with its own prefix. See the source file of each task to get the prefix and all paramters that can be set. You can also get the prefix and paramters (and their default values) by the following method (take `Dispatch` for example):

```
>>> from tlpipe.timestream import dispatch
>>> dispatch.Dispatch.prefix
'dp_'
>>> dispatch.Dispatch.show_params()
Parameters of task Dispatch:
out:  None
requires:  None
in:  None
iter_start:  0
iter_step:  1
input_files:  None
iter_num:  None
copy:  False
iterable:  False
output_files:  None
time_select:  (0, None)
stop:  None
libver:  latest
corr:  all
exclude:  []
check_status:  True
dist_axis:  0
freq_select:  (0, None)
feed_select:  (0, None)
tag_output_iter:  True
tag_input_iter:  True
start:  0
mode:  r
```

(continues on next page)

(continued from previous page)

```
pol_select:  (0, None)
extra_inttime:  150
days:  1.0
drop_days:  0.0
exclude_bad:  True
```

4. Usally the input of one task should be ether read from the data files, for example:

```
dp_input_files = files # data files as list
```

or is the output of a previously excuted task (to construct a task chain), for example:

```
dt_in = dp_out
```

```
pwf_in = dt_out
```

Iterative pipeline

To make the pipeline iteratively run for several days data, or more than one group (treat a list of files as a separate group) of data, you should set the parameter *iterable* of each task you want to iterate to *True*, and optionally specify an iteration number. If no iteration number is specified, the pipeline will iteratively run until all input data has been processed. Take again the above waterfall plot as an example, suppose you want to iteratively plot the waterfall of 2 days data, or two separate groups of data, the input pipe file *plot_wf_iter.pipe* `download` is like:

```
# -*- mode: python; -*-

# input file for the analysis pipeline
# execute this pipeline by either command of the following two:
# tlpipe dir/to/plot_wf_iter.pipe
# mpiexec -n N tlpipe dir/to/plot_wf_iter.pipe


pipe_tasks = []
pipe_outdir = './output/'
pipe_logging = 'notset'
# pipe_logging = 'info'


import glob
data_dir1 = 'dir1/to/data' # your data directory
data_dir2 = 'dir2/to/data' # your data directory

###  one way
files = sorted(glob.glob(data_dir1+'/*.hdf5')) # more than 1 day's data␣
↪files as a list

### or another way
group1 = sorted(glob.glob(data_dir1+'/*.hdf5'))
group2 = sorted(glob.glob(data_dir2+'/*.hdf5'))
files = [ group1, group2 ] # or two groups of data, each as a list of data␣
↪files


```

(continues on next page)

(continued from previous page)

```
# data selection
from tlpipe.timestream import dispatch
pipe_tasks.append(dispatch.Dispatch)
### parameters for Dispatch
dp_input_files = files # data files as list
dp_freq_select = (500, 510) # frequency indices, from 500 to 510
dp_feed_select = [1, 2, 32, 33] # feed no. as a list
dp_iterable = True
dp_iter_num = 2 # set the number of iterations
dp_tag_input_iter = False
dp_out = 'dp'

# find and mask noise source signal
from tlpipe.timestream import detect_ns
pipe_tasks.append(detect_ns.Detect)
### parameters for Detect
dt_in = dp_out
# dt_feed = 1
dt_iterable = True
dt_out = 'dt'

# plot waterfall of selected data
from tlpipe.plot import plot_waterfall
pipe_tasks.append(plot_waterfall.Plot)
### parameters for Plot
pwf_in = dt_out
pwf_iterable = True
pwf_flag_ns = True # mask noise source signal
pwf_fig_name = 'waterfall/wf' # figure name to save
pwf_out = 'pwf'
```

Note: The number of iterations can be set only once in the first task, as after the first task has been executed the specified number of iterations, it will no longer produce its output for the subsequent tasks, those task will stop to iterate when there is no input for it.

Non-trivial control flow

You can run several tasks iteratively, and then run some other tasks non-iteratively when the iterative tasks all have done.

For example, if you want the waterfall plot of two days averaged data, you can iteratively run several tasks, each iteration for one day data, and then combine (accumulate and average) the two days data and plot its waterfall, just as follows shown in *plot_wf_nontrivial.pipe* `download`:

```
# -*- mode: python; -*-

# input file for the analysis pipeline
# execute this pipeline by either command of the following two:
# tlpipe dir/to/plot_wf_nontrivial.pipe
# mpiexec -n N tlpipe dir/to/plot_wf_nontrivial.pipe


pipe_tasks = []
```

(continues on next page)

(continued from previous page)

```
pipe_outdir = './output/'
pipe_logging = 'notset'
# pipe_logging = 'info'


import glob
data_dir = 'dir/to/data' # your data directory
files = sorted(glob.glob(data_dir+'/*.hdf5')) # at least 2 days data files as a list


# data selection
from tlpipe.timestream import dispatch
pipe_tasks.append(dispatch.Dispatch)
### parameters for Dispatch
dp_input_files = files # data files as list
dp_freq_select = (500, 510) # frequency indices, from 500 to 510
dp_feed_select = [1, 2, 32, 33] # feed no. as a list
dp_iterable = True
dp_iter_num = 2 # set the number of iterations
dp_tag_input_iter = False
dp_out = 'dp'

# find and mask noise source signal
from tlpipe.timestream import detect_ns
pipe_tasks.append(detect_ns.Detect)
### parameters for Detect
dt_in = dp_out
# dt_feed = 1
dt_iterable = True
dt_out = 'dt'

# plot waterfall of selected data
from tlpipe.plot import plot_waterfall
pipe_tasks.append(plot_waterfall.Plot)
### parameters for Plot
pwf_in = dt_out
pwf_iterable = True
pwf_flag_ns = True # mask noise source signal
pwf_fig_name = 'waterfall/wf' # figure name to save
pwf_out = 'pwf'

# convert raw timestream to timestream
from tlpipe.timestream import rt2ts
pipe_tasks.append(rt2ts.Rt2ts)
### parameters for Rt2ts
r2t_in = dt_out # can also be pwf_out as it is the same
r2t_iterable = True
r2t_out = 'r2t'

# re-order the data to have RA from 0 to 2pi
from tlpipe.timestream import re_order
pipe_tasks.append(re_order.ReOrder)
### parameters for ReOrder
ro_in = r2t_out
ro_iterable = True
ro_out = 'ro'
```

(continues on next page)

(continued from previous page)

```
66
67  # accumulate the re-ordered data from different days
68  from tlpipe.timestream import accumulate
69  pipe_tasks.append(accumulate.Accum)
70  ### parameters for Accum
71  ac_in = ro_out
72  ac_iterable = True
73  ac_out = 'ac'
74
75  # barrier above iterative tasks before executing the following tasks.
76  from tlpipe.timestream import barrier
77  pipe_tasks.append(barrier.Barrier)
78  ### parameters for Barrier
79
80  # average the accumulated data
81  from tlpipe.timestream import average
82  pipe_tasks.append(average.Average)
83  ### parameters for Average
84  av_in = ac_out
85  av_output_files = [ 'average/file_%d.hdf5' %i for i in range(1, 7) ] # here␣
    ↪save intermediate results
86  av_out = 'av'
87
88  # waterfall plot of the averaged data
89  from tlpipe.plot import plot_waterfall
90  pipe_tasks.append((plot_waterfall.Plot, 'pwf1_')) # here use a new prefix␣
    ↪pwf1_ instead of the default pwf_ to discriminate from the previous plot_
    ↪waterfall
91  ### parameters for Plot
92  pwf1_in = av_out
93  pwf1_input_files = av_output_files # here you can read data from the saved␣
    ↪intermediate data files if you do not set pwf1_in
94  pwf1_flag_ns = True
95  pwf1_fig_name = 'vis_av/vis'
96  pwf1_out = 'pwf1'
```

Note: Notice the use of the task `Barrier` to block the control flow before the executing of its subsequent tasks. As the task `Barrier` won't get its input from any other tasks, the pipeline will restart at the begining every time when it gets to execute `Barrier`. Once everything before `Barrier` has been executed, it will unblocks its subsequent tasks and allow them to proceed normally.

Note: Note in real data analysis, the data should be RFI flagged, calibrated, and maybe some other processes done before the data accumulating and averaging, here for simplicity and easy understanding, we have omitted all those processes. One can refer to the real data analysis pipeline input files in the package's *input* directory.

Execute several times a same task

Special care need to be taken when executing several times a same task. Since the input pipe file is just a plain python script, it will be first executed before the parameters parsing process, the assignment of a variable will override the same named variable before it during the excuting of the pipe file script. So for the need of executing several times a same task, different prefixes should be set for each of these tasks (i.e., except for the first appeared which could have

just use the default prefix of the task, all others need to set a different prefix). To do this, you need to append a 2-tuple to the list *pipe_tasks*, with its first element being the imported task, and the second element being a new prefix to use. See for example the line

```
pipe_tasks.append((plot_waterfall.Plot, 'pwf1_')) # here use a new prefix␣
↪pwf1_ instead of the default pwf_ to discriminate from the previous plot_
↪waterfall
```

in *plot_wf_nontrivial.pipe* in the above example.

Save intermediate data

To save data that has been processed by one task (used for maybe break point recovery, etc.), you can just set the *output_files* paramter of this task to be a list of file names (can only save as *hdf5* data files), then data will be split into almost equal chunks along the time axis and save each chunk to one of the data file. For example, see the line

```
av_output_files = [ 'average/file_%d.hdf5' %i for i in range(1, 7) ] # here␣
↪save intermediate results
```

in *plot_wf_nontrivial.pipe* in the above example.

Recovery from intermediate data

You can recovery the pipeline from a break point (where you have saved the intermediate data) by reading data from data files you have saved. To do this, instead of set the *in* parameter, you need to set the *input_files* paramter to a list with elements being the saved data files. For example, see the line

```
pwf1_input_files = av_output_files # here you can read data from the saved␣
↪intermediate data files if you do not set pwf1_in
```

in *plot_wf_nontrivial.pipe* in the above example.

Note: If the *in* paramter and the *input_files* parameter are both set, the task will get its input from the *in* paramter instead of reading data from the *input_files* as it is much slower to read the data from the files. So in order to recovery from the break point, you should not set the *in* parameter, or should set *in* to be None, which is the default value.

3.1.2 Run the pipeline

Single process run

If you do not have an MPI environment installed, or you just want a single process run, just do (in case *plot_wf.pipe* is in you working directory)

```
$ tlpipe plot_wf.pipe
```

or (in case *plot_wf.pipe* isn't in you working directory)

```
$ tlpipe dir/to/plot_wf.pipe
```

If you want to submit and run the pipeline in the background, do like

```
$ nohup tlpipe dir/to/plot_wf.pipe &> output.txt &
```

Multiple processes run

To run the pipeline in parallel and distributed maner on a cluster using multiple processes, you can do something like (in case *plot_wf.pipe* is in you working directory)

```
$ mpiexec -n N tlpipe plot_wf.pipe
```

or (in case *plot_wf.pipe* isn't in you working directory)

```
$ mpiexec -n N tlpipe dir/to/plot_wf.pipe
```

If you want to submit and run the pipeline in the background on several nodes, for example, *node2*, *node3*, *node4*, do like

```
$ nohup mpiexec -n N -host node2,node3,node4 --map-by node tlpipe dir/to/plot_wf.pipe␣
↪&> output.txt &
```

Note: In the above commands, **N** is the number of processes you want to run!

3.1.3 Pipeline products and intermediate results

Pipeline products and intermediate results will be in the directory setting by *pipe_outdir*.

3.1.4 Other excutable commands

- *h5info*: Check what's in a (or a list of) HDF5 data file(s). For its use, do some thing like

  ```
  $ h5info data.hdf5
  ```

  or

  ```
  $ h5info data1.hdf5, data2.hdf5, data3.hdf5
  ```

CHAPTER 4

Developer's guid

4.1 Developer's guide

4.1.1 Write a general task

A pipeline task is a subclass of `TaskBase` intended to perform some small, modular piece analysis.

To write a general task, you can use the following template `general_task.py`:

```
"""A general task template."""

from tlpipe.pipeline.pipeline import TaskBase, PipelineStopIteration


class GeneralTask(TaskBase):
    """A general task template."""

    # input parameters and their default values as a dictionary
    params_init = {
                    'task_param': 'param_val',
                  }

    # prefix of this task
    prefix = 'gt_'

    def __init__(self, parameter_file_or_dict=None, feedback=2):

        # Read in the parameters.
        super(self.__class__, self).__init__(parameter_file_or_dict, feedback)

        # Do some initialization here if necessary
        print 'Initialize the task.'

```

(continues on next page)

(continued from previous page)

```
    def setup(self):
        # Set up works here if necessary
        print "Setting up the task."

    def next(self):
        # Doing the actual work here
        print 'Executing the task with paramter task_param = %s' % self.
↪params['task_param']
        # stop the task
        raise PipelineStopIteration()

    def finish(self):
        # Finishing works here if necessary
        print "Finished the task."
```

The developer of the task must specify what input parameters the task expects if it has and a prefix, as well as code to perform the actual processing for the task.

Input parameters are specified by adding class attributes *params_init* which is a dictionary whose entries are key and default value pairs. A *prefix* is used to identify and read the corresponding parameters from the input pipe file for this task.

To perform the actual processing for the task, you could first do the necessary initialization in `__init__()`, then implement three methods `setup()`, `next()` and `finish()`. Usually the only necessary method to be implemented is `next()`, in which the actual processing works are done, the other methods `__init__()`, `setup()`, `finish()` do not need if there is no specifical initialization, setting up, and finishing work to do. These methods are executed in order, with `next()` possibly being executed many times. Iteration of `next()` is halted by raising a `PipelineStopIteration`.

To make it work, you could put it somewhere like in *tlpipe/tlpipe/timestream/*, and write a input pipe file like `general_task.pipe`:

```
# -*- mode: python; -*-

# input file for pipeline manager
# execute this pipeline by either command of the following two:
# tlpipe dir/to/general_task.pipe
# mpiexec -n N tlpipe dir/to/general_task.pipe


pipe_tasks = []
pipe_outdir = './output/'


from tlpipe.timestream import general_task
pipe_tasks.append(general_task.GeneralTask)
### parameters for GeneralTask
gt_task_param = 'new_val'
```

then execute the task by run

```
$ tlpipe general_task.pipe
```

4.1.2 Write a task to process timestream data

To write a task to process the timestream data (i.e., the visibility and auxiliary data), you can use the following template `ts_template.py`:

```
"""Timestream task template."""

import timestream_task


class TsTemplate(timestream_task.TimestreamTask):
    """Timestream task template."""

    params_init = {
                    'task_param': 'param_val',
                  }

    prefix = 'tt_'

    def process(self, ts):

        print 'Executing the task with paramter task_param = %s' % self.
↪params['task_param']
        print
        print 'Timestream data is contained in %s' % ts

        return super(TsTemplate, self).process(ts)
```

Here, instead of inherit from `TaskBase`, we inherit from its subclass `TimestreamTask`, and implement the method `process()` (and maybe also `__init__()`, `setup()`, and `finish()` if necessary). The timestream data is contained in the argument *ts*, which may be an instance of `RawTimestream` or `Timestream`.

Note: You do not need to override the method `next()` now, because in the class `OneAndOne`, which is the super class of `TimestreamTask`, we have

```
class OneAndOne(TaskBase):

    def next(self, input=None):
        # ...
        output = self.read_process_write(input)
        # ...
        return output

    def read_process_write(self, input):
        # ...
        output = self.process(input)
        # ...
        return output
```

4.1.3 Use data operate functions in timestream tasks

To write a task to process the timestream data, you (in most cases) only need to implement `process()` with the input timestream data contained in its argument *ts*, as stated above. To help with the data processing, you could use some of the data operate functions defined in the corresponding timestream data container class, which can automatically split the data along one axis or some axes among multiple process and iteratively process all these data slices. For example, to write to task to process the raw timestream data along the axis of baseline, i.e., to process a time-frequency slice of the raw data each time, you can have the task like `ts_task.py`:

```
"""A task to process the raw timestream data along the axis of baseline."""

import timestream_task


class TsTask(timestream_task.TimestreamTask):
    """A task to process the raw timestream data along the axis of baseline."
↪""

    params_init = {
                  }

    prefix = 'tt_'

    def process(self, ts):

        # distribute data along the axis of baseline
        ts.redistribute('baseline')

        # use data operate function of `ts`
        ts.bl_data_operate(self.func)

        return super(TsTask, self).process(ts)

    def func(self, vis, vis_mask, li, gi, bl, ts, **kwargs):
        """Function that does the actual task."""

        # `vis` is the time-frequency slice of the visibility
        print vis.shape
        # `vis_mask` is the time-frequency slice of the visibility mask
        print vis_mask.shape
        # `li`, `gi` is the local and global index of this slice
        # `bl` is the corresponding baseline
        print li, gi, bl
```

To execute the task, put it somewhere like in *tlpipe/tlpipe/timestream/*, and write a input pipe file like `ts_task.pipe`:

```
# -*- mode: python; -*-

# input file for pipeline manager
# execute this pipeline by either command of the following two:
# tlpipe dir/to/ts_task.pipe
# mpiexec -n N tlpipe dir/to/ts_task.pipe


pipe_tasks = []
pipe_outdir = './output/'
```

(continues on next page)

(continued from previous page)

```
pipe_logging = 'notset'
# pipe_logging = 'info'
pipe_timing = True
pipe_flush = True


import glob
data_dir = 'dir/to/data' # your data directory
files = sorted(glob.glob(data_dir+'/*.hdf5')) # all data files as a list


# data selection
from tlpipe.timestream import dispatch
pipe_tasks.append(dispatch.Dispatch)
### parameters for Dispatch
dp_input_files = files # data files as list
dp_freq_select = (500, 510) # frequency indices, from 500 to 510
dp_feed_select = [1, 2, 32, 33] # feed no. as a list
dp_out = 'dp'

from tlpipe.timestream import ts_task
pipe_tasks.append(ts_task.TsTask)
### parameters for TsTask
tt_in = dp_out
tt_out = 'tt'
```

then execute the task by run

```
$ tlpipe ts_task.pipe
```

These are some data operate functions that you can use:

Data operate functions of `RawTimestream` and `Timestream`:

class tlpipe.container.timestream_common.**TimestreamCommon**

data_operate(*func*, *op_axis=None*, *axis_vals=0*, *full_data=False*, *copy_data=False*, *keep_dist_axis=False*, ***kwargs*)

all_data_operate(*func*, *copy_data=False*, ***kwargs*)

time_data_operate(*func*, *full_data=False*, *copy_data=False*, *keep_dist_axis=False*, ***kwargs*)

freq_data_operate(*func*, *full_data=False*, *copy_data=False*, *keep_dist_axis=False*, ***kwargs*)

bl_data_operate(*func*, *full_data=False*, *copy_data=False*, *keep_dist_axis=False*, ***kwargs*)

time_and_freq_data_operate(*func*, *full_data=False*, *copy_data=False*, *keep_dist_axis=False*, ***kwargs*)

time_and_bl_data_operate(*func*, *full_data=False*, *copy_data=False*, *keep_dist_axis=False*, ***kwargs*)

freq_and_bl_data_operate(*func*, *full_data=False*, *copy_data=False*, *keep_dist_axis=False*, ***kwargs*)

Additional data operate functions of `Timestream`:

class tlpipe.container.timestream.**Timestream**

pol_data_operate(*func*, *full_data=False*, *copy_data=False*, *keep_dist_axis=False*, ***kwargs*)

time_and_pol_data_operate(*func*, *full_data=False*, *copy_data=False*, *keep_dist_axis=False*, ***kwargs*)

freq_and_pol_data_operate(*func*, *full_data=False*, *copy_data=False*, *keep_dist_axis=False*, ***kwargs*)

pol_and_bl_data_operate(*func*, *full_data=False*, *copy_data=False*, *keep_dist_axis=False*, ***kwargs*)

time_freq_and_pol_data_operate(*func*, *full_data=False*, *copy_data=False*, *keep_dist_axis=False*, ***kwargs*)

time_freq_and_bl_data_operate(*func*, *full_data=False*, *copy_data=False*, *keep_dist_axis=False*, ***kwargs*)

time_pol_and_bl_data_operate(*func*, *full_data=False*, *copy_data=False*, *keep_dist_axis=False*, ***kwargs*)

freq_pol_and_bl_data_operate(*func*, *full_data=False*, *copy_data=False*, *keep_dist_axis=False*, ***kwargs*)

CHAPTER 5

Reference

5.1 `core` – Core functionalities

5.1.1 Constants

constants	Various constants.

tlpipe.core.constants

Various constants.

Please add a doc string if you add further constants.

Constants

c: Speed of light in m/s;

k_B: Boltzmann constant in $m^2 \cdot kg \cdot s^{-2} \cdot \mathrm{K}^{-1}$;

nu_21cm: 21cm hyperfine frequency in MHz;

sday: One sidereal day in s;

yr_s: a year in s;

5.1.2 Tianlai array model

tl_array

5.2 kiyopy – Kiyoshi Masui's paramter parsing package

5.2.1 Pameter parser

parse_ini

5.2.2 Custom exceptions

custom_exceptions	Custom Exceptions.

tlpipe.kiyopy.custom_exceptions

Custom Exceptions.

These mostly don't do anything special, but are defined such that the exceptions I raise don't conflict with generic exceptions raised by built-ins.

Exceptions

DataError	Exception to raise if data of some sort is invalid or does not have expected properties.
FileParameterTypeError	Exception to raise if a parameter read from file should be a certain type and is not.
NextIteration	Exception raised to skip iterations in a nested loop in a controled way.
ParameterFileError	Exception to raise if reading a parameter file fails.

tlpipe.kiyopy.custom_exceptions.DataError

exception tlpipe.kiyopy.custom_exceptions.**DataError**
Exception to raise if data of some sort is invalid or does not have expected properties.

tlpipe.kiyopy.custom_exceptions.FileParameterTypeError

exception tlpipe.kiyopy.custom_exceptions.**FileParameterTypeError**
Exception to raise if a parameter read from file should be a certain type and is not.

tlpipe.kiyopy.custom_exceptions.NextIteration

exception tlpipe.kiyopy.custom_exceptions.**NextIteration**
Exception raised to skip iterations in a nested loop in a controled way.

tlpipe.kiyopy.custom_exceptions.ParameterFileError

exception tlpipe.kiyopy.custom_exceptions.**ParameterFileError**
Exception to raise if reading a parameter file fails.

5.2.3 Utilities

utils	A few utilities that I have found the need for.

tlpipe.kiyopy.utils

A few utilities that I have found the need for.

Most of these are only a few lines long, but this way I don't have to remember how to do the same thing over and over.

Functions

abbreviate_file_path(fname)	Abbrviates file paths.
mkdir_p(path)	Same functionality as shell command mkdir -p.
mkparents(path)	Given a file name, makes all the parent directories.

tlpipe.kiyopy.utils.abbreviate_file_path

tlpipe.kiyopy.utils.**abbreviate_file_path**(*fname*)
Abbrviates file paths.

Givin any file path return an abbreviated path showing only the deepest most directory and the file name (Usefull for writing feedback that doesn't flood your screen.

tlpipe.kiyopy.utils.mkdir_p

tlpipe.kiyopy.utils.**mkdir_p**(*path*)
Same functionality as shell command mkdir -p.

tlpipe.kiyopy.utils.mkparents

tlpipe.kiyopy.utils.**mkparents**(*path*)
Given a file name, makes all the parent directories.

5.3 pipeline – Pipeline control and tasks

5.3.1 Pipeline

pipeline

5.4 container – Timestream data containers

5.4.1 Data containers

container
timestream_common
raw_timestream
timestream

5.5 timestream – Timestream operating tasks

5.5.1 Timestream base task

timestream_task

5.5.2 Operating tasks

dispatch
detect_ns
rfi_flagging
line_rfi
time_flag
freq_flag
multiscale_flag
combine_mask
sir_operate
rfi_stats
bad_detect
delay_transform
ns_cal
rt2ts
ps_fit
ps_cal
apply_gain
temperature_convert
phs2src
phs2zen
phase_closure
ps_subtract
daytime_mask
sun_mask
re_order
accumulate
barrier
average
freq_rebin
map_making
gen_beam

5.6 rfi – RFI flagging methods

5.6.1 Surface fitting methods

surface_fit	
local_fit	Local fit method.
local_average_fit	Local average fit method.
local_median_fit	Local median fit method.
local_minimum_fit	Local minimum fit method.
gaussian_filter	Gaussian filter method.
interpolate	Spline interpolation method.

tlpipe.rfi.surface_fit

Classes

SurfaceFitMethod(time_freq_vis[, ...])	Abstract base class for surface fitting methods.

tlpipe.rfi.surface_fit.SurfaceFitMethod

class tlpipe.rfi.surface_fit.**SurfaceFitMethod**(*time_freq_vis*, *time_freq_vis_mask=None*)

Abstract base class for surface fitting methods.

A surface fit to the correlated visibilities $V(\nu, t)$ as a function of frequency ν and time t can produce a surface $\hat{V}(\nu, t)$ that represents the astronomical information in the signal. Requiring $\hat{V}(\nu, t)$ to be a smooth surface is a good assumption for most astronomical continuum sources, as their observed amplitude tend not to change rapidly with time and frequency, whereas specific types of RFI can create sharp edges in the time-frequency domain. The residuals between the fit and the data contain the system noise and the RFI, which can then be thresholded without the chance of flagging astronomical sources that have visibilities with high amplitude.

Note: Because of the smoothing in both time and frequency direction, this method is not directly usable when observing strong line sources or strong pulsars.

__init__(*time_freq_vis*, *time_freq_vis_mask=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

Methods

fit()	Abstract method that needs to be implemented by sub-classes.

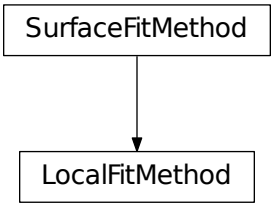
tlpipe.rfi.surface_fit.SurfaceFitMethod.fit

SurfaceFitMethod.**fit**()
Abstract method that needs to be implemented by sub-classes.

tlpipe.rfi.local_fit

Local fit method.

Inheritance diagram



Classes

LocalFitMethod(time_freq_vis[, ...])	Abstract base class for local fit method.

tlpipe.rfi.local_fit.LocalFitMethod

class tlpipe.rfi.local_fit.**LocalFitMethod**(*time_freq_vis*, *time_freq_vis_mask=None*, *time_window_size=20*, *freq_window_size=40*)

Abstract base class for local fit method.

__init__(*time_freq_vis*, *time_freq_vis_mask=None*, *time_window_size=20*, *freq_window_size=40*)

x.__init__(...) initializes x; see help(type(x)) for signature

Methods

calculate_background(x, y)	
fit()	Fit the background.

tlpipe.rfi.local_fit.LocalFitMethod.calculate_background

LocalFitMethod.**calculate_background**(*x*, *y*)

tlpipe.rfi.local_fit.LocalFitMethod.fit

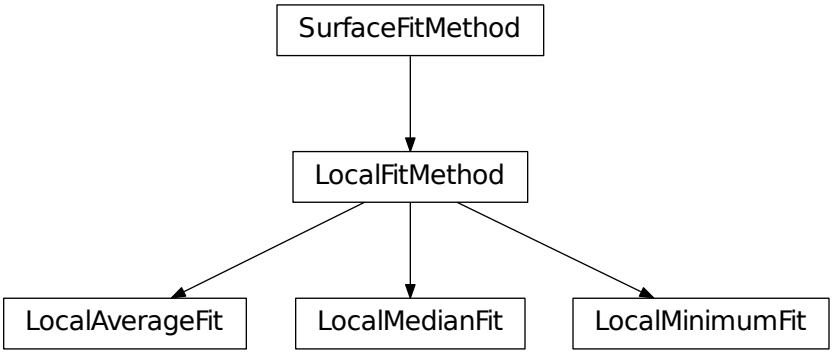
LocalFitMethod.**fit**()

Fit the background.

tlpipe.rfi.local_average_fit

Local average fit method.

Inheritance diagram



Classes

LocalAverageFit(time_freq_vis[, ...])	Local average fit method.

tlpipe.rfi.local_average_fit.LocalAverageFit

class tlpipe.rfi.local_average_fit.**LocalAverageFit**(*time_freq_vis*, *time_freq_vis_mask=None*, *time_window_size=20*, *freq_window_size=40*)

Local average fit method.

In this method, the background value is caculated by the local average of a sliding window of size $N \times M$ around each data value.

__init__(*time_freq_vis*, *time_freq_vis_mask=None*, *time_window_size=20*, *freq_window_size=40*)
x.__init__(...) initializes x; see help(type(x)) for signature

Methods

calculate_background(x, y)	
fit()	Fit the background.

tlpipe.rfi.local_average_fit.LocalAverageFit.calculate_background

LocalAverageFit.**calculate_background**(*x*, *y*)

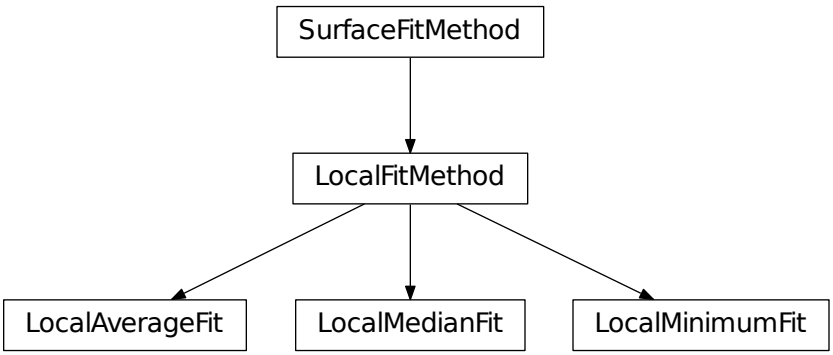
tlpipe.rfi.local_average_fit.LocalAverageFit.fit

LocalAverageFit.**fit**()
Fit the background.

tlpipe.rfi.local_median_fit

Local median fit method.

Inheritance diagram



Classes

LocalMedianFit(time_freq_vis[, ...])	Local median fit method.

tlpipe.rfi.local_median_fit.LocalMedianFit

class tlpipe.rfi.local_median_fit.**LocalMedianFit**(*time_freq_vis*, *time_freq_vis_mask=None*, *time_window_size=20*, *freq_window_size=40*)

Local median fit method.

In this method, the background value is caculated by the local median of a sliding window of size $N \times M$ around each data value.

__init__(*time_freq_vis*, *time_freq_vis_mask=None*, *time_window_size=20*, *freq_window_size=40*)

x.__init__(...) initializes x; see help(type(x)) for signature

Methods

calculate_background(x, y)	
fit()	Fit the background.

tlpipe.rfi.local_median_fit.LocalMedianFit.calculate_background

LocalMedianFit.**calculate_background**(*x*, *y*)

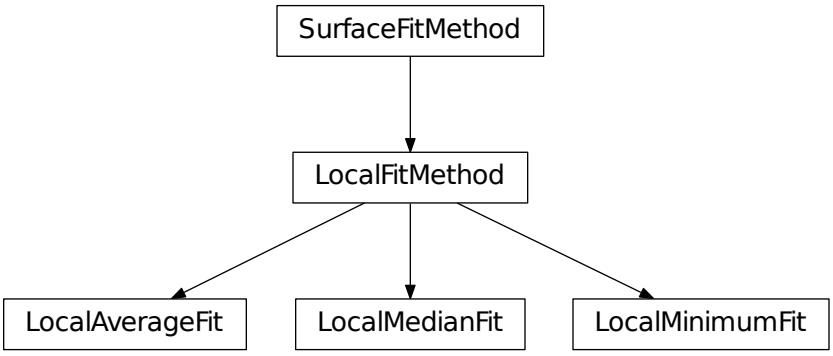
tlpipe.rfi.local_median_fit.LocalMedianFit.fit

LocalMedianFit.**fit**()
Fit the background.

tlpipe.rfi.local_minimum_fit

Local minimum fit method.

Inheritance diagram



Classes

LocalMinimumFit(time_freq_vis[, ...])	Local minimum fit method.

tlpipe.rfi.local_minimum_fit.LocalMinimumFit

class tlpipe.rfi.local_minimum_fit.**LocalMinimumFit**(*time_freq_vis*, *time_freq_vis_mask=None*, *time_window_size=20*, *freq_window_size=40*)

Local minimum fit method.

In this method, the background value is caculated by the local minimum of a sliding window of size $N \times M$ around each data value.

__init__(*time_freq_vis*, *time_freq_vis_mask=None*, *time_window_size=20*, *freq_window_size=40*)
x.__init__(...) initializes x; see help(type(x)) for signature

Methods

calculate_background(x, y)	
fit()	Fit the background.

tlpipe.rfi.local_minimum_fit.LocalMinimumFit.calculate_background

LocalMinimumFit.**calculate_background**(*x*, *y*)

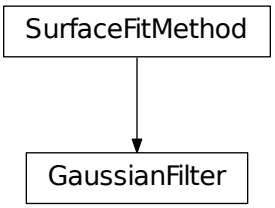
tlpipe.rfi.local_minimum_fit.LocalMinimumFit.fit

LocalMinimumFit.**fit**()
Fit the background.

tlpipe.rfi.gaussian_filter

Gaussian filter method.

Inheritance diagram



Classes

GaussianFilter(time_freq_vis[, ...])	Gaussian filter method.

tlpipe.rfi.gaussian_filter.GaussianFilter

class tlpipe.rfi.gaussian_filter.**GaussianFilter**(*time_freq_vis*, *time_freq_vis_mask=None*, *time_kernal_size=7.5*, *freq_kernal_size=15.0*, *fill_val=0*)

Gaussian filter method.

In this method, the background is caculated by a Gaussian high pass filtering process.

__init__(*time_freq_vis*, *time_freq_vis_mask=None*, *time_kernal_size=7.5*, *freq_kernal_size=15.0*, *fill_val=0*)

x.__init__(...) initializes x; see help(type(x)) for signature

Methods

fit()	Fit the background.

tlpipe.rfi.gaussian_filter.GaussianFilter.fit

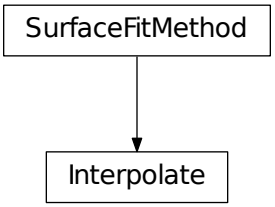
GaussianFilter.**fit**()

Fit the background.

tlpipe.rfi.interpolate

Spline interpolation method.

Inheritance diagram



Classes

Interpolate(time_freq_vis[, ...])	Spline interpolation method.

tlpipe.rfi.interpolate.Interpolate

class tlpipe.rfi.interpolate.**Interpolate**(*time_freq_vis*, *time_freq_vis_mask=None*, *direction='vertical'*, *order=3*, *ext=0*, *mask_ratio=0.1*)

Spline interpolation method.

This is not really a surface fit method, but intended to fill masked (or invalid) values presented in the data by spline interpolation.

__init__(*time_freq_vis*, *time_freq_vis_mask=None*, *direction='vertical'*, *order=3*, *ext=0*, *mask_ratio=0.1*)

x.__init__(...) initializes x; see help(type(x)) for signature

Methods

fit()	Fit the background.
interpolate_horizontally()	
interpolate_vertically()	

tlpipe.rfi.interpolate.Interpolate.fit

Interpolate.**fit**()

Fit the background.

tlpipe.rfi.interpolate.Interpolate.interpolate_horizontally

Interpolate.**interpolate_horizontally**()

tlpipe.rfi.interpolate.Interpolate.interpolate_vertically

Interpolate.**interpolate_vertically**()

5.6.2 Combinatorial thresholding methods

combinatorial_threshold	
var_threshold	The VarThreshold method.
sum_threshold	The SumThreshold method.

tlpipe.rfi.combinatorial_threshold

Classes

CombinatorialThreshold(time_freq_vis[, ...])	Abstract base class for combinatorial thresholding methods.

tlpipe.rfi.combinatorial_threshold.CombinatorialThreshold

class tlpipe.rfi.combinatorial_threshold.**CombinatorialThreshold**(*time_freq_vis*, *time_freq_vis_mask=None*, *first_threshold=6.0*, *exp_factor=1.5*, *distribution='Rayleigh'*, *max_threshold_length=1024*)

Abstract base class for combinatorial thresholding methods.

The method will flag a combination of samples when a property of this combination exceeds some limit. The more connected samples are combined, the lower the sample threshold.

For more details, see Offringa et al., 2000, MNRAS, 405, 155, *Post-correlation radio frequency interference classification methods*.

For this implementation, the sequence of thresholds are determined by the following formula:

$$\alpha \times \frac{\rho^i}{w} \times (\sigma \times \beta) + \eta$$

in which α is the first threshold set to 6.0, $\rho = 1.5$, i is the current iteration, w the current window size, σ the standard deviation of the values, β the base sensitivity, set to 1.0, and η the median.

__init__(*time_freq_vis*, *time_freq_vis_mask=None*, *first_threshold=6.0*, *exp_factor=1.5*, *distribution='Rayleigh'*, *max_threshold_length=1024*)
x.__init__(...) initializes x; see help(type(x)) for signature

Methods

execute([sensitivity, direction])	Execute the thresholding method.
execute_threshold(factor, direction)	Abstract method that needs to be implemented by sub-classes.
init_threshold_with_flase_rate(resolution, ...)	

tlpipe.rfi.combinatorial_threshold.CombinatorialThreshold.execute

CombinatorialThreshold.**execute**(*sensitivity=1.0*, *direction=('time', 'freq')*)
Execute the thresholding method.

tlpipe.rfi.combinatorial_threshold.CombinatorialThreshold.execute_threshold

CombinatorialThreshold.**execute_threshold**(*factor*, *direction*)
Abstract method that needs to be implemented by sub-classes.

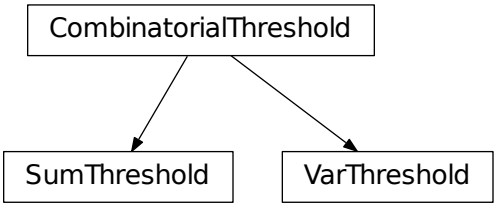
tlpipe.rfi.combinatorial_threshold.CombinatorialThreshold.init_threshold_with_flase_rate

CombinatorialThreshold.**init_threshold_with_flase_rate**(*resolution*, *false_alarm_rate*)

tlpipe.rfi.var_threshold

The VarThreshold method.

Inheritance diagram



Classes

VarThreshold(time_freq_vis[, ...])	The VarThreshold method.

tlpipe.rfi.var_threshold.VarThreshold

class tlpipe.rfi.var_threshold.**VarThreshold**(*time_freq_vis*, *time_freq_vis_mask=None*, *first_threshold=6.0*, *exp_factor=1.2*, *distribution='Rayleigh'*, *max_threshold_length=1024*)

The VarThreshold method.

For more details, see Offringa et al., 2000, MNRAS, 405, 155, *Post-correlation radio frequency interference classification methods*.

__init__(*time_freq_vis*, *time_freq_vis_mask=None*, *first_threshold=6.0*, *exp_factor=1.2*, *distribution='Rayleigh'*, *max_threshold_length=1024*)
x.__init__(...) initializes x; see help(type(x)) for signature

Methods

execute([sensitivity, direction])	Execute the thresholding method.
execute_threshold(factor, direction)	Abstract method that needs to be implemented by sub-classes.
horizontal_var_threshold(length, threshold)	
init_threshold_with_flase_rate(resolution, ...)	
vertical_var_threshold(length, threshold)	

tlpipe.rfi.var_threshold.VarThreshold.execute

VarThreshold.**execute**(*sensitivity=1.0*, *direction=('time', 'freq')*)
Execute the thresholding method.

tlpipe.rfi.var_threshold.VarThreshold.execute_threshold

VarThreshold.**execute_threshold**(*factor*, *direction*)
Abstract method that needs to be implemented by sub-classes.

tlpipe.rfi.var_threshold.VarThreshold.horizontal_var_threshold

VarThreshold.**horizontal_var_threshold**(*length*, *threshold*)

tlpipe.rfi.var_threshold.VarThreshold.init_threshold_with_flase_rate

VarThreshold.**init_threshold_with_flase_rate**(*resolution*, *false_alarm_rate*)

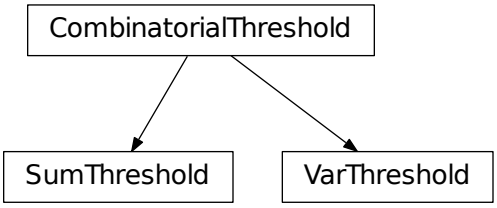
tlpipe.rfi.var_threshold.VarThreshold.vertical_var_threshold

VarThreshold.**vertical_var_threshold**(*length*, *threshold*)

tlpipe.rfi.sum_threshold

The SumThreshold method.

Inheritance diagram



Classes

SumThreshold(time_freq_vis[, ...])	The SumThreshold method.

tlpipe.rfi.sum_threshold.SumThreshold

class tlpipe.rfi.sum_threshold.**SumThreshold**(*time_freq_vis*, *time_freq_vis_mask=None*, *first_threshold=6.0*, *exp_factor=1.5*, *distribution='Rayleigh'*, *max_threshold_length=1024*, *min_connected=1*)

The SumThreshold method.

For more details, see Offringa et al., 2000, MNRAS, 405, 155, *Post-correlation radio frequency interference classification methods*.

__init__(*time_freq_vis*, *time_freq_vis_mask=None*, *first_threshold=6.0*, *exp_factor=1.5*, *distribution='Rayleigh'*, *max_threshold_length=1024*, *min_connected=1*)

x.__init__(...) initializes x; see help(type(x)) for signature

Methods

execute([sensitivity, direction])	Execute the thresholding method.
execute_threshold(factor, direction)	Abstract method that needs to be implemented by sub-classes.
horizontal_sum_threshold(length, threshold)	
init_threshold_with_flase_rate(resolution, ...)	
vertical_sum_threshold(length, threshold)	

tlpipe.rfi.sum_threshold.SumThreshold.execute

SumThreshold.**execute**(*sensitivity=1.0*, *direction=('time', 'freq')*)
Execute the thresholding method.

tlpipe.rfi.sum_threshold.SumThreshold.execute_threshold

SumThreshold.**execute_threshold**(*factor*, *direction*)
Abstract method that needs to be implemented by sub-classes.

tlpipe.rfi.sum_threshold.SumThreshold.horizontal_sum_threshold

SumThreshold.**horizontal_sum_threshold**(*length*, *threshold*)

tlpipe.rfi.sum_threshold.SumThreshold.init_threshold_with_flase_rate

SumThreshold.**init_threshold_with_flase_rate**(*resolution*, *false_alarm_rate*)

tlpipe.rfi.sum_threshold.SumThreshold.vertical_sum_threshold

SumThreshold.**vertical_sum_threshold**(*length*, *threshold*)

5.6.3 Mathematical morphological methods

dilate_operator	This implements the mathematical morphological dilate operation.
sir_operator	This implements the scale-invariant rank (SIR) operator.

tlpipe.rfi.dilate_operator

This implements the mathematical morphological dilate operation.

Functions

dilate1d(mask, size)
horizontal_dilate(mask, size[, overwrite])
vertical_dilate(mask, eta[, overwrite])

tlpipe.rfi.dilate_operator.dilate1d

tlpipe.rfi.dilate_operator.**dilate1d**(*mask*, *size*)

tlpipe.rfi.dilate_operator.horizontal_dilate

tlpipe.rfi.dilate_operator.**horizontal_dilate**(*mask*, *size*, *overwrite=True*)

tlpipe.rfi.dilate_operator.vertical_dilate

tlpipe.rfi.dilate_operator.**vertical_dilate**(*mask*, *eta*, *overwrite=True*)

tlpipe.rfi.sir_operator

This implements the scale-invariant rank (SIR) operator.

The operator considers a sample to be contaminated with RFI when the sample is in a subsequence of mostly flagged samples. To be more precise, it will flag a subsequence when more than $(1-\eta)N$ of its samples are flagged, with N the number of samples in the subsequence and η a constant, $0 \leq \eta \leq 1$. Using ρ to denote the operator, the output $\rho(X)$ can be formally defined as

$$\rho(X) \equiv \bigcup \{[Y1, Y2) \mid \#(X \cap [Y1, Y2)) \geq (1-\eta)(Y2 - Y1)\},$$

with $[Y1, Y2)$ a half-open interval of a one-dimensional set, and the hash symbol $\#$ denoting the count-operator that returns the number of elements in the set. In words, the equation defines $\rho(X)$ to consist of all the samples that are in an interval $[Y1, Y2)$, in which the ratio of samples in the input X is greater or equal than $(1-\eta)$. Parameter η represents the aggressiveness of the method: with $\eta = 0$, no additional samples are flagged and $\rho(X) = X$. On the other hand, $\eta = 1$ implies all samples will be flagged.

For more details, see Offringa et al., 2012, A&A, 539, A95, *A morphological algorithm for improving radio-frequency interference detection*.

Functions

horizontal_sir
sir1d
vertical_sir

tlpipe.rfi.sir_operator.horizontal_sir

tlpipe.rfi.sir_operator.**horizontal_sir**()

tlpipe.rfi.sir_operator.sir1d

tlpipe.rfi.sir_operator.**sir1d**()

tlpipe.rfi.sir_operator.vertical_sir

tlpipe.rfi.sir_operator.**vertical_sir**()

5.7 cal – Calibration

Coming soon...

5.8 map – Map-making

5.8.1 Driftscan map-making

Richard Shaw's m-mode formalism method.

5.9 foreground – Foreground subtraction

Coming soon...

5.10 ps – Power spectrum estimation

Coming soon...

5.11 plot – Tasks for plottting

5.11.1 Plotting tasks

plot_integral
plot_slice
plot_waterfall
plot_phase

5.12 utils – Utility functions

5.12.1 Utilities

date_util	Date and time utils.
np_util	
path_util	Path utils.
pickle_util	Some functions related to pickle.
robust_stats	Robust statistical utilities.
sg_filter	
rpca_decomp	
multiscale	
hist_eq	

tlpipe.utils.date_util

Date and time utils.

Functions

get_ephdate(local_time[, tzone])	Convert *local_time* to ephem utc date.
get_juldate(local_time[, tzone])	Convert *local_time* to Julian date.

tlpipe.utils.date_util.get_ephdate

tlpipe.utils.date_util.**get_ephdate**(*local_time*, *tzone='UTC+08h'*)

Convert *local_time* to ephem utc date.

Parameters

- **local_time** (*string, python date or datetime object, etc.*) – Local time that can be passed to ephem.Date function. Refer to http://rhodesmill.org/pyephem/date.html
- **tzone** (*string, optional*) – Time zone in format 'UTC[+/-]xxh'. Defaut: UTC+08h.

Returns **utc_time** – A float number representation of a UTC PyEphem date.

Return type float

See also:

get_juldate

tlpipe.utils.date_util.get_juldate

tlpipe.utils.date_util.**get_juldate**(*local_time*, *tzone='UTC+08h'*)

Convert *local_time* to Julian date.

Parameters

- **local_time** (*string, python date or datetime object, etc.*) – Local time that can be passed to ephem.Date function. Refer to http://rhodesmill.org/pyephem/date.html
- **tzone** (*string, optional*) – Time zone in format 'UTC[+/-]xxh'. Defaut: UTC+08h.

Returns **julian_date** – A float number representation of a Julian date.

Return type float

See also:

get_ephdate

tlpipe.utils.np_util

Functions

average(a[, axis, weights, returned])	Return the weighted average of array over the given axis.
unique(ar[, return_index, return_inverse, ...])	Find the unique elements of an array.

tlpipe.utils.np_util.average

tlpipe.utils.np_util.**average**(*a*, *axis=None*, *weights=None*, *returned=False*)

Return the weighted average of array over the given axis.

Copied from newer version of numpy, as old version raise "ComplexWarning: Casting complex values to real discards the imaginary part".

Parameters

- **a** (*array_like*) – Data to be averaged. Masked entries are not taken into account in the computation.
- **axis** (*int, optional*) – Axis along which the average is computed. The default is to compute the average of the flattened array.
- **weights** (*array_like, optional*) – The importance that each element has in the computation of the average. The weights array can either be 1-D (in which case its length must be the size of *a* along the given axis) or of the same shape as *a*. If `weights=None`, then all data in *a* are assumed to have a weight equal to one. If *weights* is complex, the imaginary parts are ignored.
- **returned** (*bool, optional*) – Flag indicating whether a tuple `(result, sum of weights)` should be returned as output (True), or just the result (False). Default is False.

Returns **average, [sum_of_weights]** – The average along the specified axis. When returned is *True*, return a tuple with the average as the first element and the sum of the weights as the second element. The return type is *np.float64* if *a* is of integer type and floats smaller than *float64*, or the input data-type, otherwise. If returned, *sum_of_weights* is always *float64*.

Return type (tuple of) scalar or MaskedArray

Examples

```
>>> a = np.ma.array([1., 2., 3., 4.], mask=[False, False, True, True])
>>> np.ma.average(a, weights=[3, 1, 0, 0])
1.25
```

```
>>> x = np.ma.arange(6.).reshape(3, 2)
>>> print x
[[ 0.  1.]
 [ 2.  3.]
 [ 4.  5.]]
>>> avg, sumweights = np.ma.average(x, axis=0, weights=[1, 2, 3],
...                                 returned=True)
>>> print avg
[2.66666666667 3.66666666667]
```

tlpipe.utils.np_util.unique

tlpipe.utils.np_util.**unique**(*ar*, *return_index=False*, *return_inverse=False*, *return_counts=False*)

Find the unique elements of an array.

Returns the sorted unique elements of an array. There are two optional outputs in addition to the unique elements: the indices of the input array that give the unique values, and the indices of the unique array that reconstruct the input array.

Copied from newer version of numpy, as old version has no *return_counts* argument.

Parameters

- **ar** (*array_like*) – Input array. This will be flattened if it is not already 1-D.
- **return_index** (*bool, optional*) – If True, also return the indices of *ar* that result in the unique array.
- **return_inverse** (*bool, optional*) – If True, also return the indices of the unique array that can be used to reconstruct *ar*.
- **return_counts** (*bool, optional*) – New in version 1.9.0.

  If True, also return the number of times each unique value comes up in *ar*.

Returns

- **unique** (*ndarray*) – The sorted unique values.
- **unique_indices** (*ndarray, optional*) – The indices of the first occurrences of the unique values in the (flattened) original array. Only provided if *return_index* is True.
- **unique_inverse** (*ndarray, optional*) – The indices to reconstruct the (flattened) original array from the unique array. Only provided if *return_inverse* is True.
- **unique_counts** (*ndarray, optional*) – .. versionadded:: 1.9.0 The number of times each of the unique values comes up in the original array. Only provided if *return_counts* is True.

See also:

numpy.lib.arraysetops() Module with a number of other functions for performing set operations on arrays.

Examples

```
>>> np.unique([1, 1, 2, 2, 3, 3])
array([1, 2, 3])
>>> a = np.array([[1, 1], [2, 3]])
>>> np.unique(a)
array([1, 2, 3])
```

Return the indices of the original array that give the unique values:

```
>>> a = np.array(['a', 'b', 'b', 'c', 'a'])
>>> u, indices = np.unique(a, return_index=True)
>>> u
array(['a', 'b', 'c'],
      dtype='|S1')
>>> indices
```

(continues on next page)

(continued from previous page)

```
array([0, 1, 3])
>>> a[indices]
array(['a', 'b', 'c'],
      dtype='|S1')
```

Reconstruct the input array from the unique values:

```
>>> a = np.array([1, 2, 6, 4, 2, 3, 2])
>>> u, indices = np.unique(a, return_inverse=True)
>>> u
array([1, 2, 3, 4, 6])
>>> indices
array([0, 1, 4, 3, 1, 2, 1])
>>> u[indices]
array([1, 2, 6, 4, 2, 3, 2])
```

tlpipe.utils.path_util

Path utils.

Functions

input_path(path[, iteration])	Normalize the given input *path*.
iter_path(path, iteration)	Insert current iteration flag to the file path.
output_path(path[, relative, mkdir, iteration])	Normalize the given output *path*.

tlpipe.utils.path_util.input_path

tlpipe.utils.path_util.**input_path**(*path*, *iteration=None*)

Normalize the given input *path*.

This function supposes that *path* is a absolute path if it starts with /, else it is relative to os.environ['TL_OUTPUT'].

Parameters

- **path** (*string or list of strings*) – The input path.
- **iteration** (*None or integer, optional*) – The iteration number. If it is not None, it will be inserted into the path before the path's basename. Default is None.

Returns **norm_path** – The normalized absolute input path.

Return type string or list of strings

tlpipe.utils.path_util.iter_path

tlpipe.utils.path_util.**iter_path**(*path*, *iteration*)

Insert current iteration flag to the file path.

Parameters

- **path** (*string*) – The output path.

- **iteration** (*integer*) – The iteration number.

Returns **new_path** – The generated new path which has the iteration been inserted.

Return type string

tlpipe.utils.path_util.output_path

tlpipe.utils.path_util.**output_path**(*path*, *relative=True*, *mkdir=True*, *iteration=None*)

Normalize the given output *path*.

This function supposes that *path* is relative to os.environ['TL_OUTPUT'] if *relative* is True.

Parameters

- **path** (*string or list of strings*) – The output path.
- **relative** (*bool, optional*) – The *path* is relative to os.environ['TL_OUTPUT'] if True. Default is True.
- **mkdir** (*bool, optional*) – Make the path directory if True. Default is True.
- **iteration** (*None or integer, optional*) – The iteration number. If it is not None, it will be inserted into the path before the path's basename. Default is None.

Returns **norm_path** – The normalized absolute output path.

Return type string or list of strings

tlpipe.utils.pickle_util

Some functions related to pickle.

Functions

get_value(val)	Return unpickled value of *val* if it is a pickled object, else just return itself.

tlpipe.utils.pickle_util.get_value

tlpipe.utils.pickle_util.**get_value**(*val*)

Return unpickled value of *val* if it is a pickled object, else just return itself.

tlpipe.utils.robust_stats

Robust statistical utilities.

This implements the Median Absolute Deviation (MAD) and some Winsorized statistical methods.

A sample $x_1, \cdots, x_n$ is sorted in ascending order. For the chosen $0 \leq \gamma \leq 0.5$ and $k = [\gamma n]$ winsorization of the sorted data consists of setting

$$W_i = \begin{cases} x_{(k+1)}, & \text{if} \quad x_{(i)} \leq x_{(k+1)} \\ x_{(i)}, & \text{if} \quad x_{(k+1)} \leq x_{(i)} \leq x_{(n-k)} \\ x_{(n-k)}, & \text{if} \quad x_{(i)} \geq x_{(n-k)}. \end{cases}$$

The winsorized sample mean is $\hat{\mu}_w = \frac{1}{n}\sum_{i=1}^{n} W_i$ and the winsorized sample variance is $D_w = \frac{1}{n-1}\sum_{i=1}^{n}(W_i - \hat{\mu}_w)^2$.

For this implementation, the statistics is computed for winsorized data with $\gamma = 0.1$.

Functions

MAD(a)	Median absolute deviation divides 0.6745.
mad(a)	Median absolute deviation.
winsorized_mean_and_std(a)	
winsorized_mode(a)	

tlpipe.utils.robust_stats.MAD

tlpipe.utils.robust_stats.**MAD**(*a*)
Median absolute deviation divides 0.6745.

tlpipe.utils.robust_stats.mad

tlpipe.utils.robust_stats.**mad**(*a*)
Median absolute deviation.

tlpipe.utils.robust_stats.winsorized_mean_and_std

tlpipe.utils.robust_stats.**winsorized_mean_and_std**(*a*)

tlpipe.utils.robust_stats.winsorized_mode

tlpipe.utils.robust_stats.**winsorized_mode**(*a*)

tlpipe.utils.sg_filter

Functions

savitzky_golay(y, window_size, order[, ...])	Smooth (and optionally differentiate) data with a Savitzky-Golay filter.

tlpipe.utils.sg_filter.savitzky_golay

tlpipe.utils.sg_filter.**savitzky_golay**(*y*, *window_size*, *order*, *deriv=0*, *rate=1*)
Smooth (and optionally differentiate) data with a Savitzky-Golay filter.

The Savitzky-Golay filter removes high frequency noise from data. It has the advantage of preserving the original shape and features of the signal better than other types of filtering approaches, such as moving averages techniques.

Copied from http://scipy.github.io/old-wiki/pages/Cookbook/SavitzkyGolay

Parameters

- **y** (*array_like, shape (N,)*) – the values of the time history of the signal.
- **window_size** (*int*) – the length of the window. Must be an odd integer number.
- **order** (*int*) – the order of the polynomial used in the filtering. Must be less then *window_size* - 1.
- **deriv** (*int*) – the order of the derivative to compute (default = 0 means only smoothing)

Returns **ys** – the smoothed signal (or it's n-th derivative).

Return type ndarray, shape (N)

Notes

The Savitzky-Golay is a type of low-pass filter, particularly suited for smoothing noisy data. The main idea behind this approach is to make for each point a least-square fit with a polynomial of high order over a odd-sized window centered at the point.

Examples

```
>>> t = np.linspace(-4, 4, 500)
>>> y = np.exp( -t**2 ) + np.random.normal(0, 0.05, t.shape)
>>> ysg = savitzky_golay(y, window_size=31, order=4)
>>> import matplotlib.pyplot as plt
>>> plt.plot(t, y, label='Noisy signal')
>>> plt.plot(t, np.exp(-t**2), 'k', lw=1.5, label='Original signal')
>>> plt.plot(t, ysg, 'r', label='Filtered signal')
>>> plt.legend()
>>> plt.show()
```

References

tlpipe.utils.rpca_decomp

Functions

MAD(a)	Median absolute deviation divides 0.6745.
decompose(M[, rank, S, lmbda, threshold, ...])	Stable principal component decomposition of an Hermitian matrix.
l0_norm(a)	Return the l_0-norm (i.e., number of non-zero elements) of an array.
l1_norm(a)	Return the l_1-norm of an array.
mad(a)	Median absolute deviation.
shrink(a, lmbda)	Soft thresholding operator, which works for both real and complex array.
sign(a)	Sign of an array, which works for both real and complex array.
truncate(a, lmbda)	Hard thresholding operator, which works for both real and complex array.

tlpipe.utils.rpca_decomp.MAD

tlpipe.utils.rpca_decomp.**MAD**(*a*)
Median absolute deviation divides 0.6745.

tlpipe.utils.rpca_decomp.decompose

tlpipe.utils.rpca_decomp.**decompose**(*M*, *rank=1*, *S=None*, *lmbda=None*, *threshold='hard'*, *max_iter=100*, *tol=1e-08*, *check_hermitian=False*, *debug=False*)
Stable principal component decomposition of an Hermitian matrix.

tlpipe.utils.rpca_decomp.l0_norm

tlpipe.utils.rpca_decomp.**l0_norm**(*a*)
Return the l_0-norm (i.e., number of non-zero elements) of an array.

tlpipe.utils.rpca_decomp.l1_norm

tlpipe.utils.rpca_decomp.**l1_norm**(*a*)
Return the l_1-norm of an array.

tlpipe.utils.rpca_decomp.mad

tlpipe.utils.rpca_decomp.**mad**(*a*)
Median absolute deviation. Works for both real and complex array.

tlpipe.utils.rpca_decomp.shrink

tlpipe.utils.rpca_decomp.**shrink**(*a*, *lmbda*)
Soft thresholding operator, which works for both real and complex array.

tlpipe.utils.rpca_decomp.sign

tlpipe.utils.rpca_decomp.**sign**(*a*)
Sign of an array, which works for both real and complex array.

tlpipe.utils.rpca_decomp.truncate

tlpipe.utils.rpca_decomp.**truncate**(*a*, *lmbda*)
Hard thresholding operator, which works for both real and complex array.

tlpipe.utils.multiscale

Functions

convolve(a, phi)	Convolve *a* along each axis sequentially by *phi*.
median_wavelet_detrend(a[, level, scale, ...])	Return the detrended component (i.e., smooth component being subtracted) of the median-wavelet transfrom.
median_wavelet_smooth(a[, level, scale, ...])	Return the smooth component of the median-wavelet transform.
median_wavelet_transform(a[, level, scale, ...])	Median-wavelet transfrom.
multiscale_median_detrend(a[, level, scale])	Return the detrended component (i.e., smooth component being subtracted) of the multiscale median transfrom.
multiscale_median_flag(a[, level, scale, ...])	
multiscale_median_smooth(a[, level, scale])	Return the smooth component of the multiscale median transform.
multiscale_median_transform(a[, level, ...])	Multiscale median transform.
starlet_detrend(a[, level, phi])	Return the detrended component (i.e., smooth component being subtracted) of the first generation starlet transfrom.
starlet_smooth(a[, level, phi])	Return the smooth component of the first generation starlet transform.
starlet_transform(a[, level, gen2, ...])	Computes the starlet transform (i.e.
up_sampling(a)	Up-sampling an array by interleaving it with zero values.

tlpipe.utils.multiscale.convolve

tlpipe.utils.multiscale.**convolve**(*a*, *phi*)

Convolve *a* along each axis sequentially by *phi*.

tlpipe.utils.multiscale.median_wavelet_detrend

tlpipe.utils.multiscale.**median_wavelet_detrend**(*a*, *level=None*, *scale=2*, *tau=5.0*, *phi=array([0.0625, 0.25, 0.375, 0.25, 0.0625])*)

Return the detrended component (i.e., smooth component being subtracted) of the median-wavelet transfrom.

tlpipe.utils.multiscale.median_wavelet_smooth

tlpipe.utils.multiscale.**median_wavelet_smooth**(*a*, *level=None*, *scale=2*, *tau=5.0*, *phi=array([0.0625, 0.25, 0.375, 0.25, 0.0625])*)

Return the smooth component of the median-wavelet transform.

tlpipe.utils.multiscale.median_wavelet_transform

tlpipe.utils.multiscale.**median_wavelet_transform**(*a*, *level=None*, *scale=2*, *tau=5.0*, *approx_only=False*, *phi=array([0.0625, 0.25, 0.375, 0.25, 0.0625])*)

Median-wavelet transfrom.

tlpipe.utils.multiscale.multiscale_median_detrend

tlpipe.utils.multiscale.**multiscale_median_detrend**(*a*, *level=None*, *scale=2*)

Return the detrended component (i.e., smooth component being subtracted) of the multiscale median transfrom.

tlpipe.utils.multiscale.multiscale_median_flag

tlpipe.utils.multiscale.**multiscale_median_flag**(*a*, *level=None*, *scale=2*, *tau=5.0*, *return_mask=True*)

tlpipe.utils.multiscale.multiscale_median_smooth

tlpipe.utils.multiscale.**multiscale_median_smooth**(*a*, *level=None*, *scale=2*)

Return the smooth component of the multiscale median transform.

tlpipe.utils.multiscale.multiscale_median_transform

tlpipe.utils.multiscale.**multiscale_median_transform**(*a*, *level=None*, *scale=2*, *approx_only=False*)

Multiscale median transform.

tlpipe.utils.multiscale.starlet_detrend

tlpipe.utils.multiscale.**starlet_detrend**(*a*, *level=None*, *phi=array([0.0625, 0.25, 0.375, 0.25, 0.0625])*)

Return the detrended component (i.e., smooth component being subtracted) of the first generation starlet transfrom.

tlpipe.utils.multiscale.starlet_smooth

tlpipe.utils.multiscale.**starlet_smooth**(*a*, *level=None*, *phi=array([0.0625, 0.25, 0.375, 0.25, 0.0625])*)

Return the smooth component of the first generation starlet transform.

tlpipe.utils.multiscale.starlet_transform

tlpipe.utils.multiscale.**starlet_transform**(*a*, *level=None*, *gen2=False*, *approx_only=False*, *phi=array([0.0625, 0.25, 0.375, 0.25, 0.0625])*)

Computes the starlet transform (i.e. the undecimated isotropic wavelet transform) of an array.

The output is a python list containing the sub-bands. If the keyword Gen2 is set, then it is the 2nd generation starlet transform which is computed: i.e., g = Id - h*h instead of g = Id - h.

tlpipe.utils.multiscale.up_sampling

tlpipe.utils.multiscale.**up_sampling**(*a*)

Up-sampling an array by interleaving it with zero values.

tlpipe.utils.hist_eq

Functions

hist_eq(img)	Implementation of Histogram equalization.

tlpipe.utils.hist_eq.hist_eq

tlpipe.utils.hist_eq.**hist_eq**(*img*)

Implementation of Histogram equalization.

Histogram equalization is a method in image processing of contrast adjustment using the image's histogram. More details see https://en.wikipedia.org/wiki/Histogram_equalization

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

Symbols

A

C

D

E

F

G

H

U

V

W