Tianlai data processing pipeline --- *tlpipe*

Shifan Zuo (左世凡)

Department of Astronomy, Tsinghua University

21cm Science Mini-Workshop, 10 December 2020

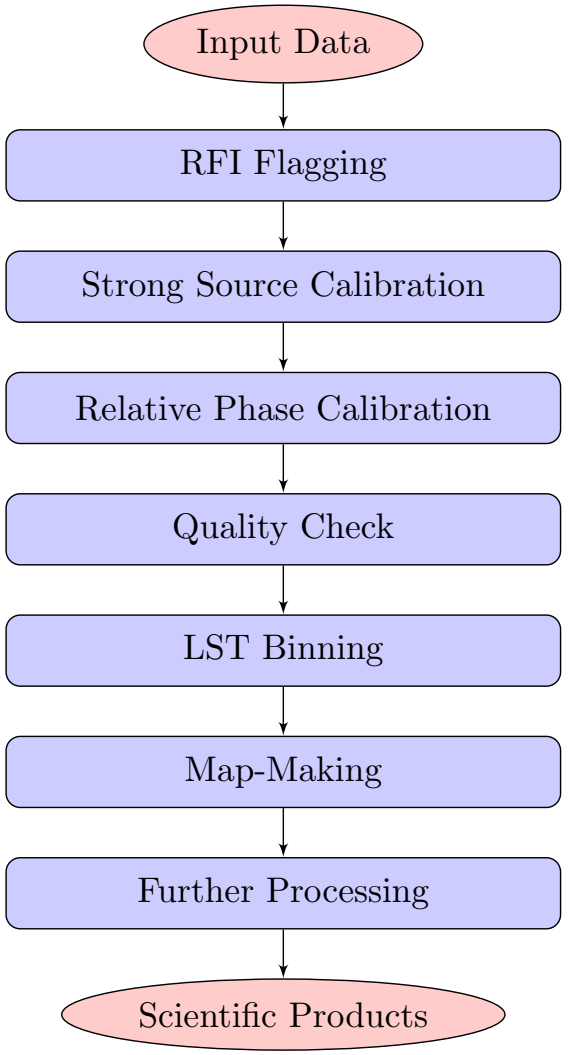
Outline

- ❑ Tianlai experiment and its data processing
- ❑ Tianlai data formats
- ❑ Tianlai data processing pipeline --- ***tlpipe***
- ❑ Tianlai data processing results
- ❑ Summary and future work

The Tianlai Experiment

A 21cm intensity mapping experiment, consists of two sub-arrays: 3 cylinders + 16 dishes.

Tianlai Data Processing



Difficulties:

- Two different sub-arrays
  - 3 cylinders + 16 dishes
- Unconventional observing mode
  - drift scan
- Wide field of view
  - ~120° in North-South direction for cylinder
- Large amounts of data
  - ~4 TB/day for cylinder + dish
- Multiple data processing steps
  - See left
- Some steps are complex to do
  - e.g. RFI flagging, calibration, map-making, etc.

Customized Data Processing Pipeline

- Processing of interferometer array data is often complicated.
- **Miriad** and **CASA** are large packages for very general propose interferometer array data processing.
- The overhead of learning and using these packages is quite high.
- It is often necessary to develop a customized data processing pipelines .
- Many 21cm intensity mapping experiments have developed their own customized data processing pipeline.
- We have developed a customized data processing pipeline software for Tianlai, named ***tlpipe***.

Tianlai Data Formats

- The Tianlai array raw data files are written in the HDF5 format.
- HDF5 data format is used throughout the data analysis pipeline.
- HDF5 format has much more flexibility than some other data formats.
- HDF5 supports data chunking, external (i.e. distributed) object storage, filter pipeline for data compression, parallel I/O, etc.
- It is particularly well suited for processing large amounts of scientific data.
- The underlying data model of FITS can be easily ported to HDF5.

Header of Tianlai HDF5 Data File

Table A.1: Header of an example Tianlai HDF5 data file.

File: 20180322181759_20180322182754.hdf5*
/.attrs["comment"]: Note potential instrumental RFI.
/.attrs["observer"]: XXX**
/.attrs["history"]: Recorded from the correlator.
/.attrs["keywordver"]: 0.0
/.attrs["sitename"]: Hongliuxia Observatory
/.attrs["sitelat"]: 44.15268333
/.attrs["sitelon"]: 91.80686667
/.attrs["siteelev"]: 1493.7
/.attrs["timezone"]: UTC+08h
/.attrs["epoch"]: 2000.0
/.attrs["telescope"]: Tianlai-Cylinder-I
/.attrs["nants"]: 3
/.attrs["npols"]: 2
/.attrs["nfeeds"]: 96
/.attrs["cylen"]: 40.0
/.attrs["cywid"]: 15.0
/.attrs["recvver"]: 0.0
/.attrs["lofreq"]: 935.0
/.attrs["corrver"]: 0.0
/.attrs["samplingbits"]: 8
/.attrs["corrmode"]: 1
/.attrs["inttime"]: 3.99507456
/.attrs["obstime"]: 2018/03/22 18:17:59.457007
/.attrs["sec1970"]: 1521713879.46
/.attrs["nfreq"]: 1008
/.attrs["freqstart"]: 685.9765625
/.attrs["freqstep"]: 0.1220703125
antpointing shape = (1, 96, 4)
antpointing.attrs["unit"]: degree
blorder shape = (18528, 2)
channo shape = (96, 2)
feedno shape = (96,)
feedpos shape = (96, 3)
feedpos.attrs["unit"]: meter
noisesource shape = (1, 3)
noisesource.attrs["unit"]: second
nspos shape = (1, 3)
nspos.attrs["unit"]: meter
pointingtime shape = (1, 2)
pointingtime.attrs["unit"]: second
polerr shape = (96, 2)
polerr.attrs["unit"]: degree
vis shape = (150, 1008, 18528)
weather shape = (1, 10)

* Data is taken from 2018/03/22 18:17:59 to 2018/03/22/ 18:27:54 in Beijing Standard Time.

** The name of the actual observer is not shown here.

Overview of the Pipeline --- *tlpipe*

https://github.com/TianlaiProject/tlpipe

- A customized data processing pipeline software for Tianlai.
- Python with C extension.
- Using an object oriented programming (OOP) scheme.
- Modular, flexible and extendable.
- Using MPI framework for high performance parallel computing.
- Use HDF5 data formats, with parallel I/O support.
- Include a customized very general executing framework and data container.
- Provides about 50 data analysis, exploration, visualization and processing tasks.
- New tasks can be easily added according to unified interfaces.
- Open sourced with GNU General Public License.

Code Organization of *tlpipe*

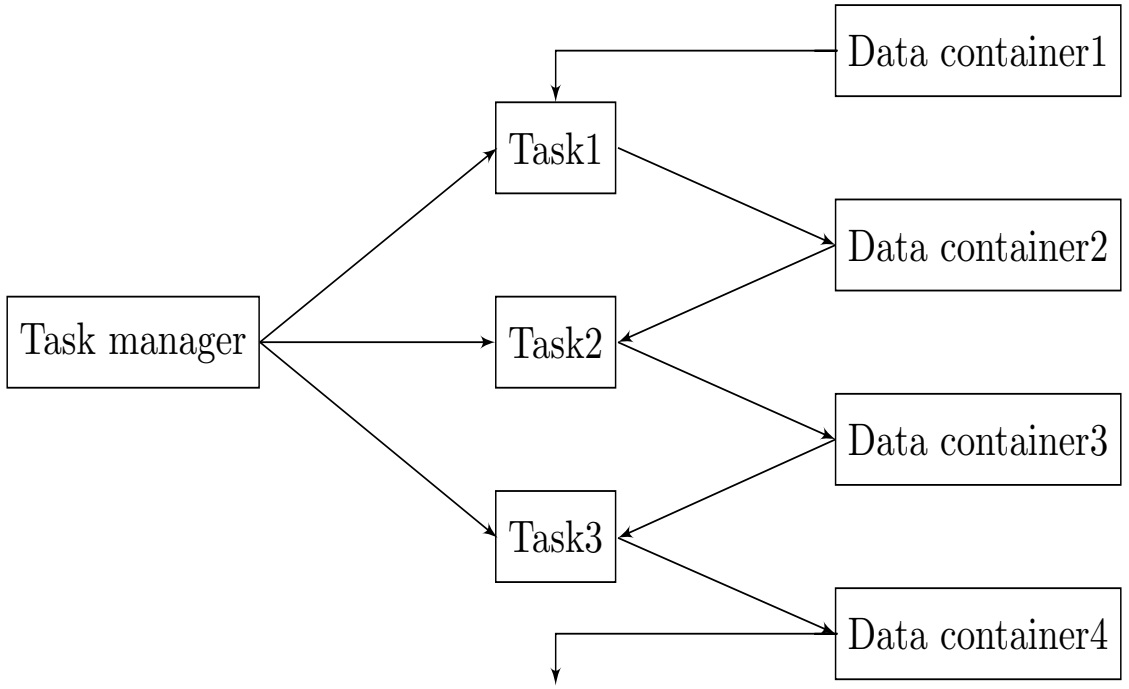
Sub-packages of ***tlpipe***:

- **core:** The definition and implementation of the beam and the interferometer array.
- **pipeline:** Pipeline control including the task executing framework,.
- **container:** The customized data containers.
- **timestream:** Tasks for processing the observed time stream data.
- **rfi:** RFI flagging algorithms.
- **cal:** A catalog of calibrator sources and things related to calibration.
- **map:** The m-mode map-making method.
- **foreground:** For foreground removal.
- **powerspectrum:** For power spectrum estimation.
- **plot:** Visualization utilities.
- **utils:** Miscellaneous functions used in the code.
- **test:** Unit testing code for the package.

Executing Framework of the Pipeline

- **Task manager:** Controls the execution of the tasks.
- **Task:** Individual or independent data processing step.
- **Data container:** Holds the data and some descriptive metadata to be processed by a task.

The task manager controls the execution of the tasks according to the settings in an input parameter file.



Task

Task: Individual or independent data processing step.

A pipeline task is a subclass of **TaskBase.**

The three methods defined by **TaskBase** are:

- setup(), for setting-up the task, called only once
- next(), needs to be overridden to perform specific job, may be called multiple times
- finish() , for finalizing the task, called only once

setup(), finish() do not need if there is no special setting-up, and finalizing work to do.

These common interfaces enable the task to be executed by the task manager.

Task can be very general and do anything.

A general task template:

```
1  """A general task template."""
2
3  from tlpipe.pipeline.pipeline import TaskBase, PipelineStopIteration
4
5
6  class GeneralTask(TaskBase):
7      """A general task template."""
8
9      # input parameters and their default values as a dictionary
10     params_init = {
11                     'task_param': 'param_val',
12                   }
13
14     # prefix of this task
15     prefix = 'gt_'
16
17     def __init__(self, parameter_file_or_dict=None, feedback=2):
18
19         # Read in the parameters.
20         super(self.__class__, self).__init__(parameter_file_or_dict, feedback)
21
22         # Do some initialization here if necessary
23         print 'Initialize the task.'
24
25     def setup(self):
26         # Set up works here if necessary
27         print "Setting up the task."
28
29     def next(self):
30         # Doing the actual work here
31         print 'Executing the task with paramter task_param = %s' % self.params['task_
32         # stop the task
33         raise PipelineStopIteration()
34
35     def finish(self):
36         # Finishing works here if necessary
37         print "Finished the task."
```

Task Parameters

The developer of the task can specify what input parameters the task expects 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 parameter file for this task.

A general task template:

```
1  """A general task template."""
2
3  from tlpipe.pipeline.pipeline import TaskBase, PipelineStopIteration
4
5
6  class GeneralTask(TaskBase):
7      """A general task template."""
8
9      # input parameters and their default values as a dictionary
10     params_init = {
11                     'task_param': 'param_val',
12                   }
13
14     # prefix of this task
15     prefix = 'gt_'
16
17     def __init__(self, parameter_file_or_dict=None, feedback=2):
18
19         # Read in the parameters.
20         super(self.__class__, self).__init__(parameter_file_or_dict, feedback)
21
22         # Do some initialization here if necessary
23         print 'Initialize the task.'
24
25     def setup(self):
26         # Set up works here if necessary
27         print "Setting up the task."
28
29     def next(self):
30         # Doing the actual work here
31         print 'Executing the task with paramter task_param = %s' % self.params['task_
32         # stop the task
33         raise PipelineStopIteration()
34
35     def finish(self):
36         # Finishing works here if necessary
37         print "Finished the task."
```

input parameters and default values

prefix

Input Parameter File

Parameters needed by the task are not hard-coded, instead they are fed to the pipeline by an input parameter file.

An input parameter file is a normal python file. It is flexible, powerful and easy to write.

The input parameters file fully determine the pipeline control flow and the behavior of each task in the pipeline.

Example input parameter file: general_task.pipe

```
1  # -*- mode: python; -*-
2
3  # input file for pipeline manager
4  # execute this pipeline by either command of the following two:
5  # tlpipe dir/to/general_task.pipe
6  # mpiexec -n N tlpipe dir/to/general_task.pipe
7
8
9  pipe_tasks = []
10 pipe_outdir = './output/'
11
12
13 from tlpipe.timestream import general_task
14 pipe_tasks.append(general_task.GeneralTask)
15 ### parameters for GeneralTask
16 gt_task_param = 'new_val'
```

prefix + parameter

Data Processing Task

To write a task to process the time stream data (i.e., the visibility and auxiliary data), you can use the template →

Now inherit from **TimestreamTask**, which inherits from **OneAndOne**, which finally inherits from **TaskBase**.

Now implement the process() method to perform data processing actions.

process() method accept a **data container** as input and produce another **data container** as output.

Write a task inheriting from **TimestreamTask** can save you a lot of efforts.

A data processing task template:

```
"""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_p
        print
        print 'Timestream data is contained in %s' % ts

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

```
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
```

Customized Data Container

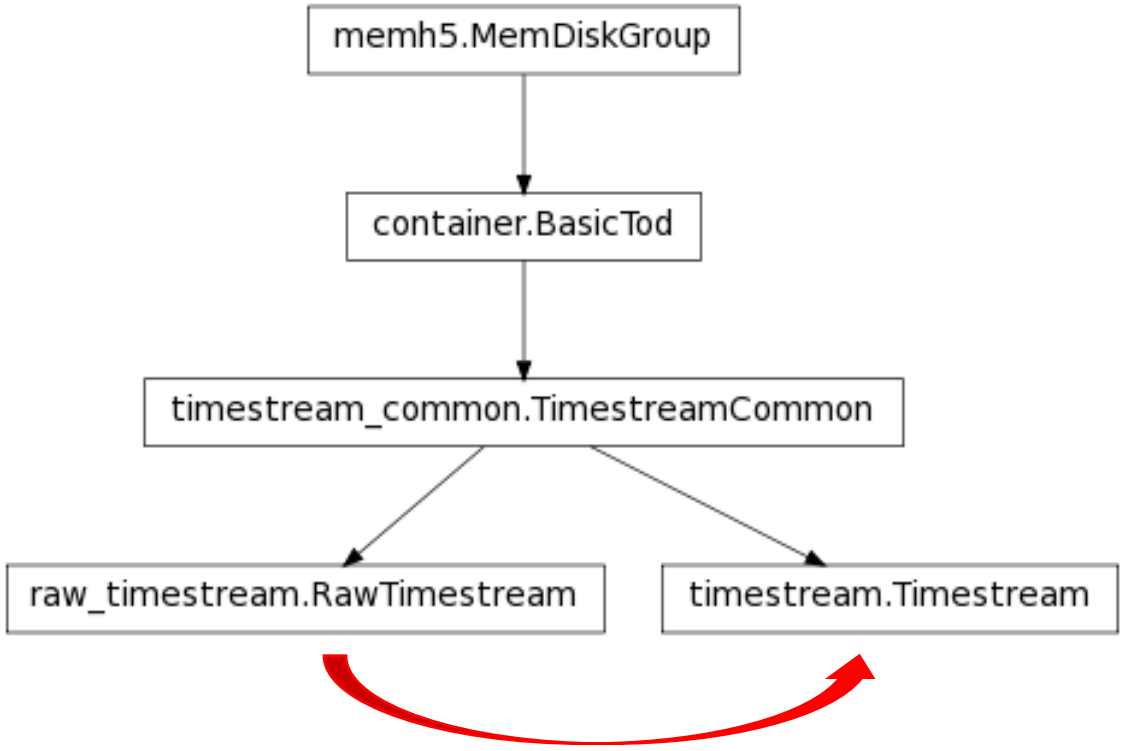
The data container holds the data and some descriptive metadata to be processed by a task.

The data container is combined with the data processing tasks, usually as an input to the task, and as an output after the operations are finished by the task.

Two customized customized data containers are used:

- **RawTimestream:** holds the the raw visibility data with mixed polarization and baseline
- **Timestream:** holds the visibility data with polarization and baseline separated

RawTimestream can be converted into **Timestream**.



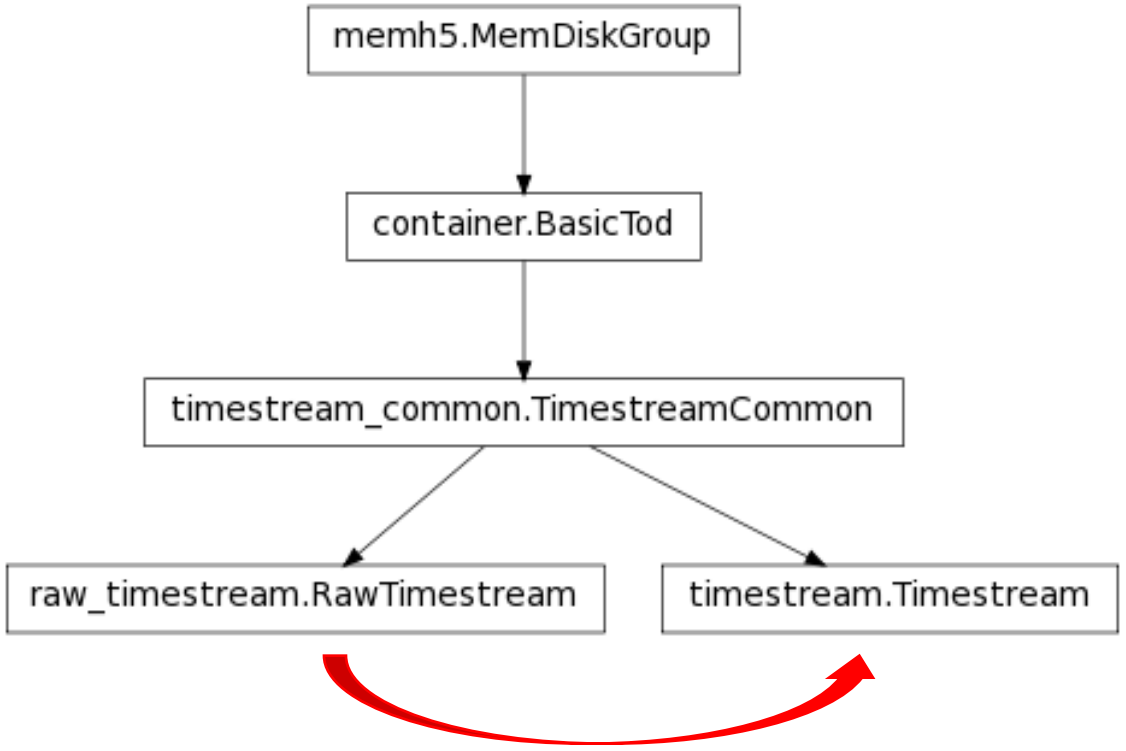
Data Container as Memory Mirror of HDF5 Files

The data container mirrors all the corresponding HDF5 file objects (specifically h5py objects), such as **the group**, **the data set**, and **the attribute**, and preserves their corresponding organization and structural relations in memory.

You can think the data container as a memory version of a HDF5 file.

This correspondence enables the user to apply familiar HDF5 file operation methods on the data container, and also makes the data I/O conversion between the data container in memory and the corresponding HDF5 files on disk simple.

Data contained in data container can be distributed on different machines by using MPI.



Data Operation Interfaces

The data container provides general data operation interfaces for the tasks to act on the data.

The tasks can access the data by using these interfaces, so that the inner consistency of the data is maintained.

By using these data operation interfaces the code can automatically split the data along one axis or some axes among multiple processes and iteratively process all these data slices.

Makes parallel programming easy.

It is encouraged to use these data operation interfaces instead of directly access the data in the data container.

Some data operation interfaces:

- all_data_operate()
- time_data_operate()
- freq_data_operate()
- bl_data_operate()
- pol_data_operate()
- time_and_freq_data_operate()
- time_and_bl_data_operate()
- freq_and_bl_data_operate()
- time_and_pol_data_operate()
- freq_and_pol_data_operate()
- pol_and_bl_data_operate()
- …

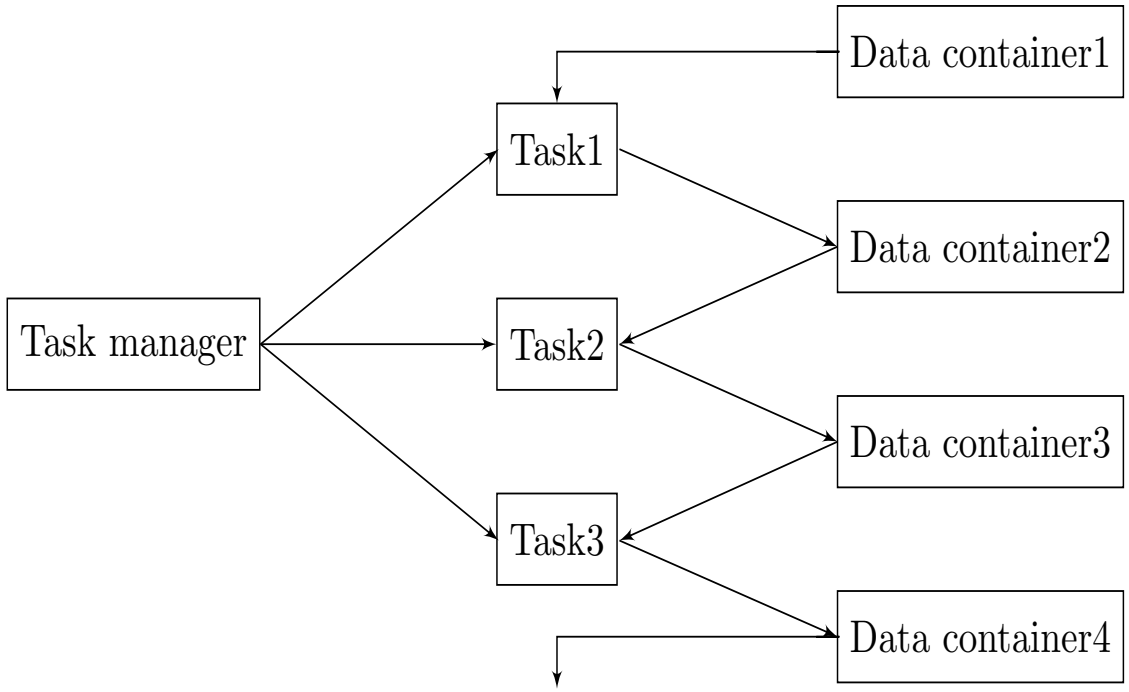
Pipeline Executing

1. Prepare for the input parameter file
2. Execute the pipeline by:

   $ tlpipe example.pipe

   or

   $ mpiexec –n N example.pipe



Algorithm 1 The algorithm that `tlpipe` follows to execute tasks.

1: Read in the input parameter file and parses it.
2: Put the tasks listed in the file into a list according to their execution order.
3: Initializes an instance for each task by using the parameters specified in the parameter file.
4: **for** task in task list **do**
5: **if** no input data container **then**
6: **if** a list of data files is given **then**
7: Create a data container by loading data from the specified data files as the input.
8: **else**
9: Raise an exception.
10: **end if**
11: **else**
12: Pass the input data container to the task.
13: Execute the task.
14: Take the output data container and maybe pass it to the next task.
15: **end if**
16: **end for**
17: Deletes the instances and finishes the pipeline.

Implemented Tasks

We have implemented about 50 tasks in the master branch of the code, more are on other branches, and new tasks will be added as we continue developing ***tlpipe***.

- **dispatch.py:** Read in and dispatch data.
- **detect_ns.py:** Detect noise source signal.
- **rfi_flagging.py:** RFI flagging by using SumThreshold method.
- **sir_operate.py:** RFI flagging by using Scale Invariant Rank (SIR) method.
- **rfi_stats.py:** RFI statistics.
- **ns_cal.py:** Relative phase calibration by using noise source signal.
- **ps_cal.py:** Absolute calibration by using point source on sky.
- **rt2ts.py:** Convert RawTimestream to Timestream.
- **freq_rebin.py:** Rebin the frequency channels.
- **accumulate.py:** Accumulate data from different days.
- **average.py:** Average data of different days.
- **map_making.py:** Map-making by using m-mode method.
- **plot_watefall.py:** Plot waterfall images of the visibility.
- **plot_slice.py:** Plot time or frequency slices of the visibility.
- …

Tianlai Observing Data

Tianlai is an interferometer array consists of two subarrays: 3 cylinders + 16 dishes

The observable of an interferometer array is called visibility.

Visibility $V_{ij} = g_i g_j \int A_i(\hat{n}) A_j^*(\hat{n}) T(\hat{n}) e^{2\pi i \hat{n}\cdot\vec{u}_{ij}} d^2\hat{n} + \mathrm{n_{ij}}$

The observed visibilities are saved as a 3D data set: (time, frequency, channel).
When polarization and baseline are separated, it is a 4D data set: (time, frequency, polarization, baseline).

The observational data are naturally divided in the time dimension and saved as a separate HDF5 file.

Data used in this presentation is cylinder data observed in 2016/09 and 2018/03.

Data Reading and Pre-Processing

A few data reading and pre-processing steps are done by ***tlpipe*** before further data processing steps:

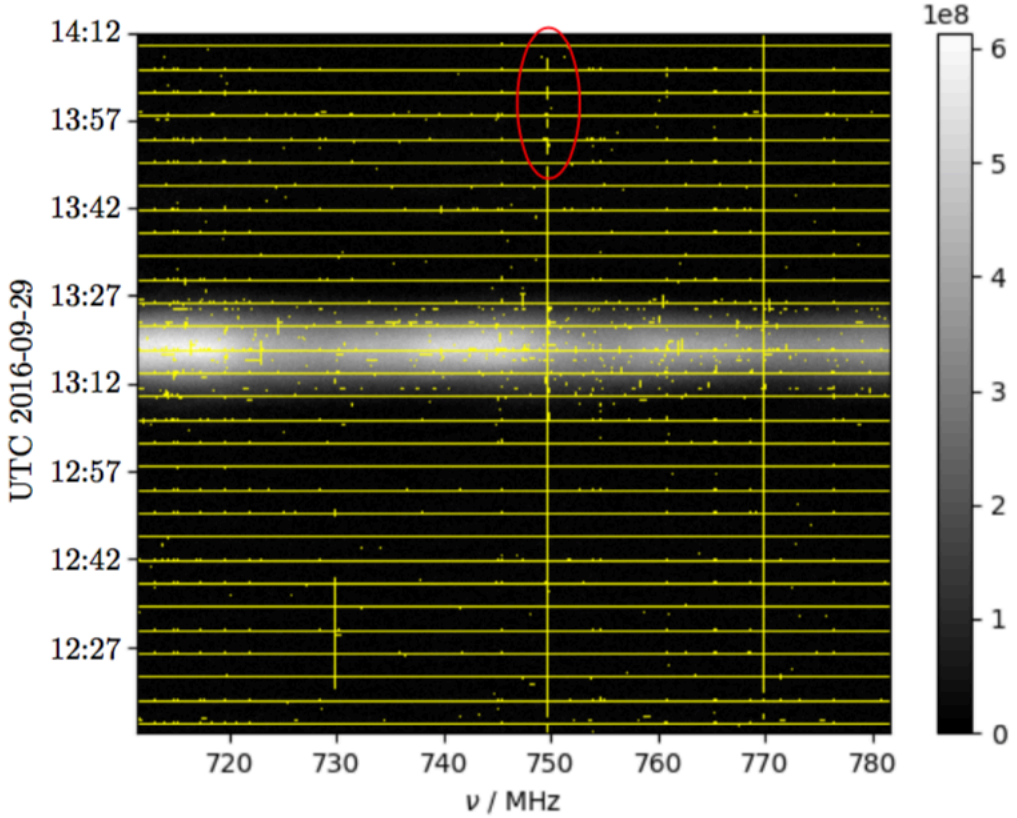
1. Select particular time points, frequency bins or a list of feeds by setting the corresponding parameters **time_select**, **frequency_select, feed_select** in the input parameter file.
2. The selected data is loaded into a data container, usually the **RawTimestream** data container, for analysis.
3. A Boolean mask array with the same dimension as the visibility array is generated after the data is loaded.
4. Maybe convert the **RawTimestream** data container into a **Timestream** data container according to one's data processing needs.
5. Do further processing steps.

RFI Flagging

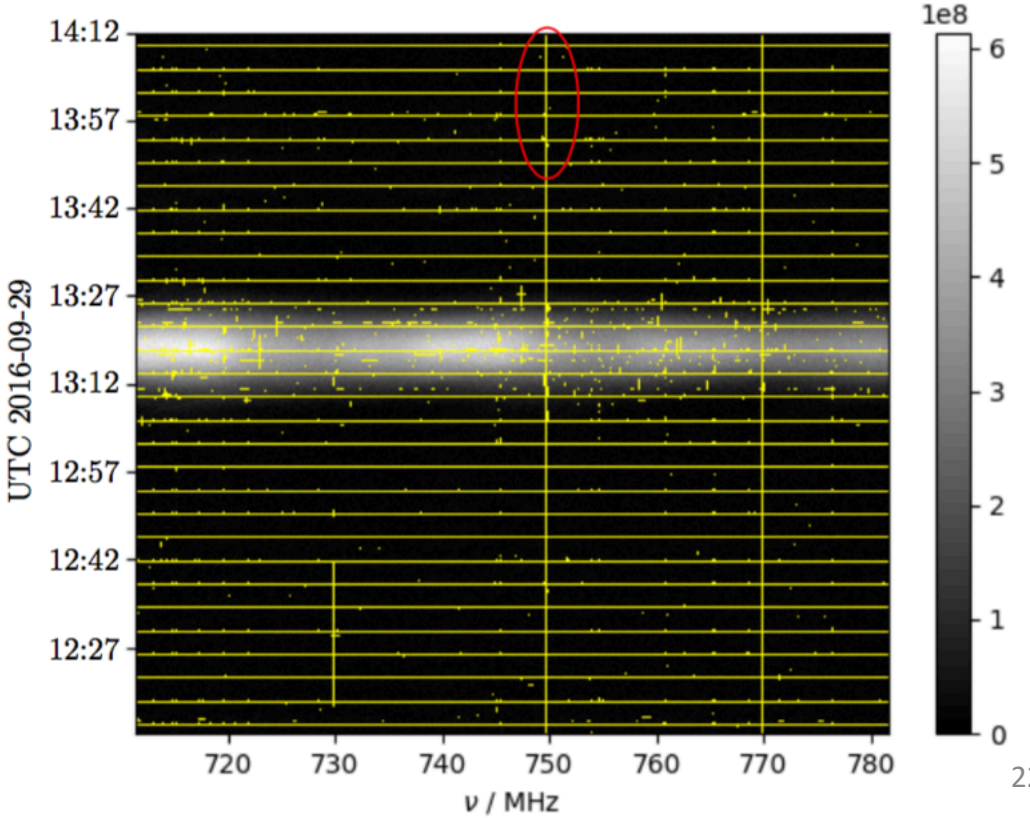
A two-step RFI flagging process is used: **SumThreshold** + **Scale Invariant Rank (SIR)**

Offringa et al., 2010 Offringa et al., 2012

SumThreshold



Scale Invariant Rank (SIR)

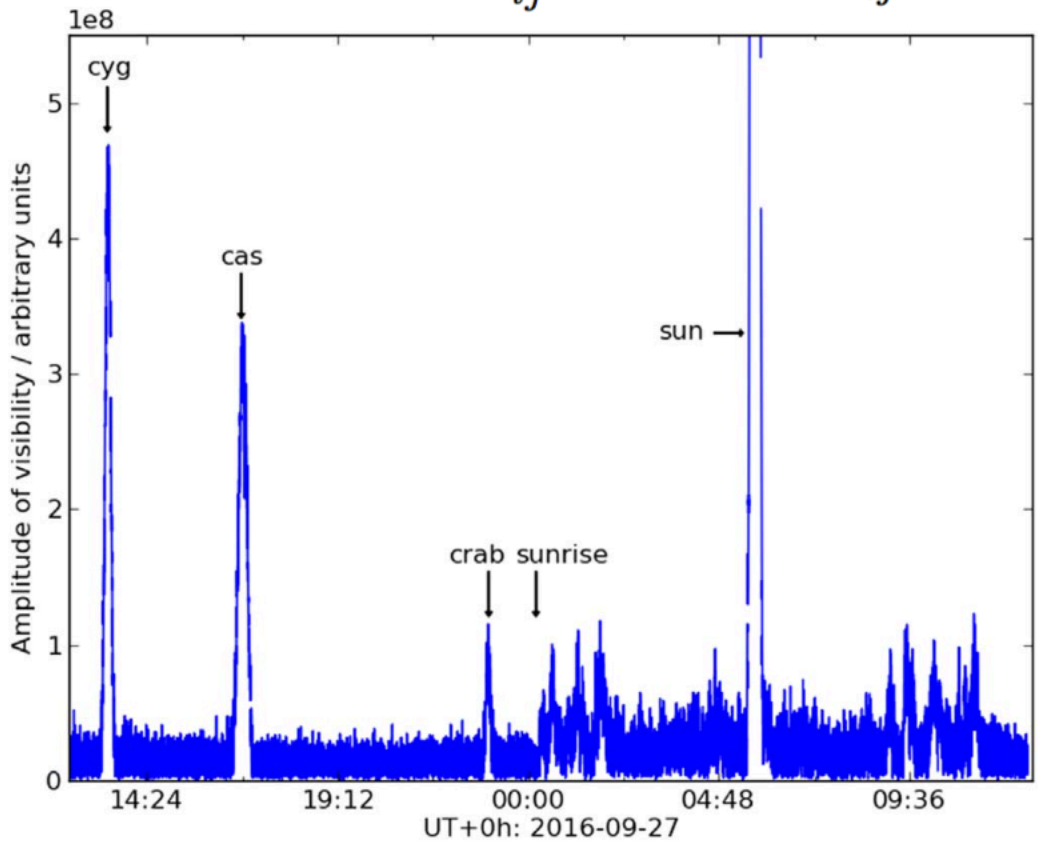


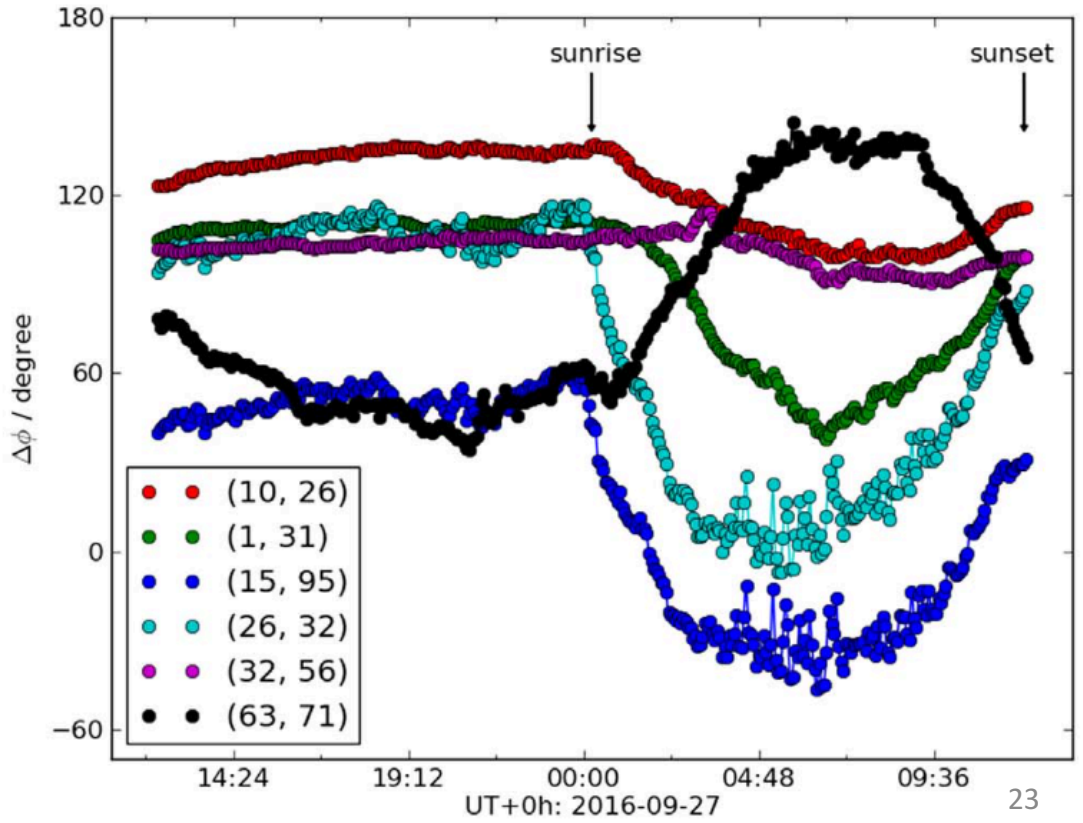
Relative Phase Calibration

There are only a few sources on the sky that are strong enough for absolute calibration, so a noise source is used to calibrate the relative phase change.

$$\phi_{ij} = \mathrm{Arg}(V_{ij}^{\mathrm{on}} - V_{ij}^{\mathrm{off}}) = k\Delta L_{ij} + \mathrm{const.},$$

$$V_{ij}^{\mathrm{rel\text{-}cal}} = e^{-i\phi_{ij}} V_{ij}.$$





Absolute Calibration

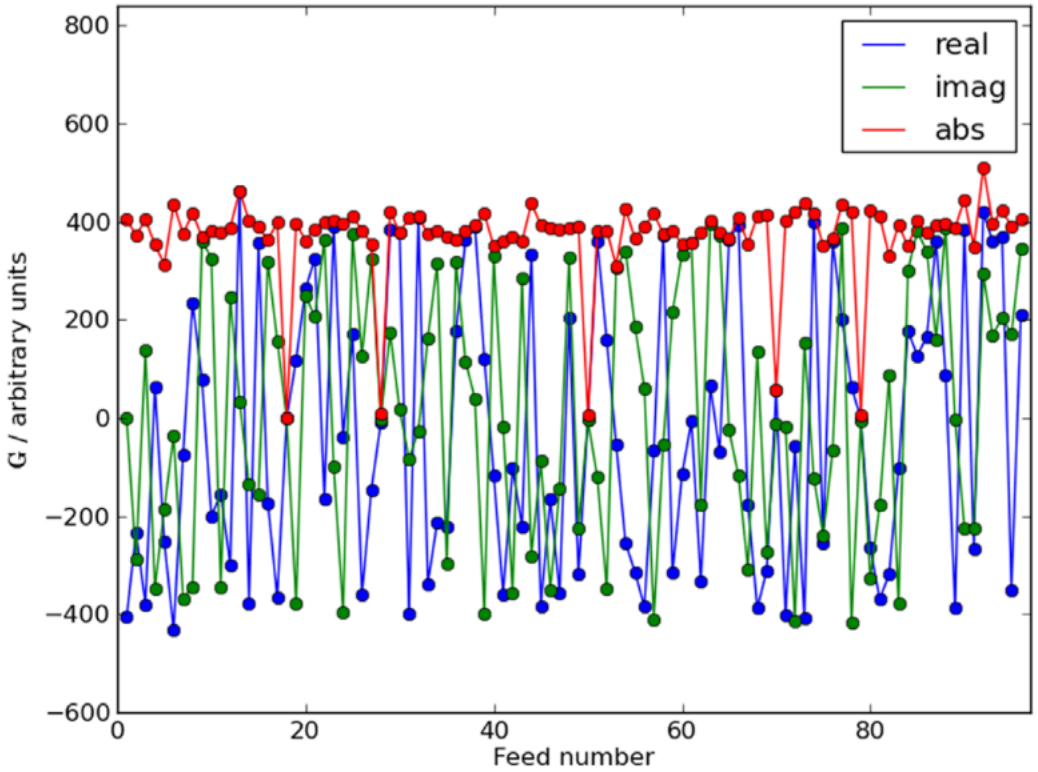
Use strong sky sources for absolute calibration.

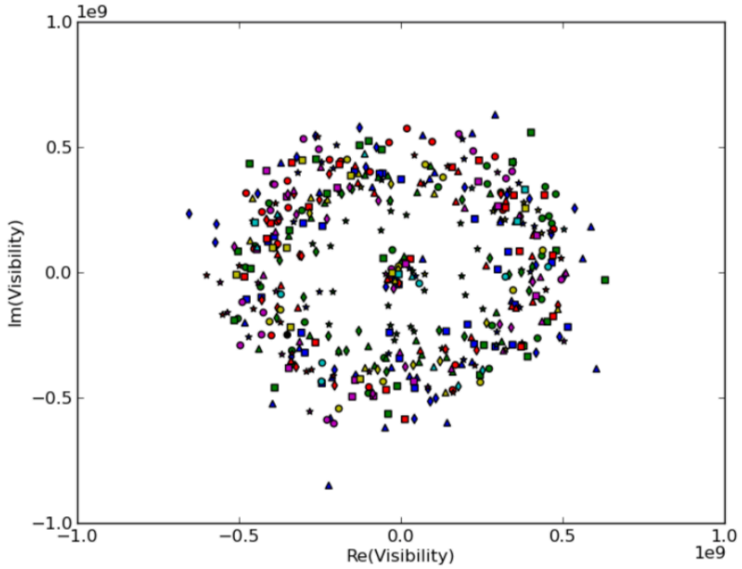
Use an eigenvector based method.

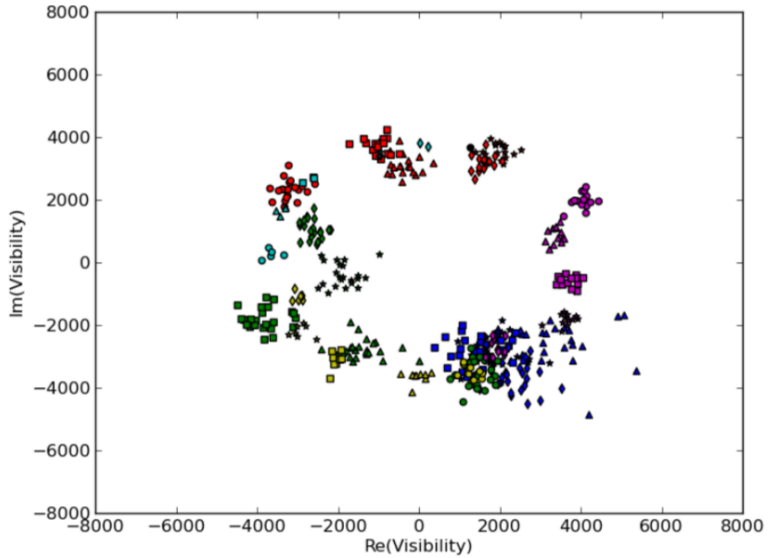
For a strong point source, we have $V_{ij}^0 = S_c\, G_i G_j^*$,

where $G_i = g_i A_i(\hat{\boldsymbol{n}}_0) e^{-2\pi i \hat{\boldsymbol{n}}_0 \cdot \boldsymbol{u}_i}$;

In matrix form, $\boldsymbol{V}_0 = S_c\, \boldsymbol{G}\boldsymbol{G}^\dagger$. $\boldsymbol{G}$ is an eigenvector of $\boldsymbol{V}_0$







Map-Making

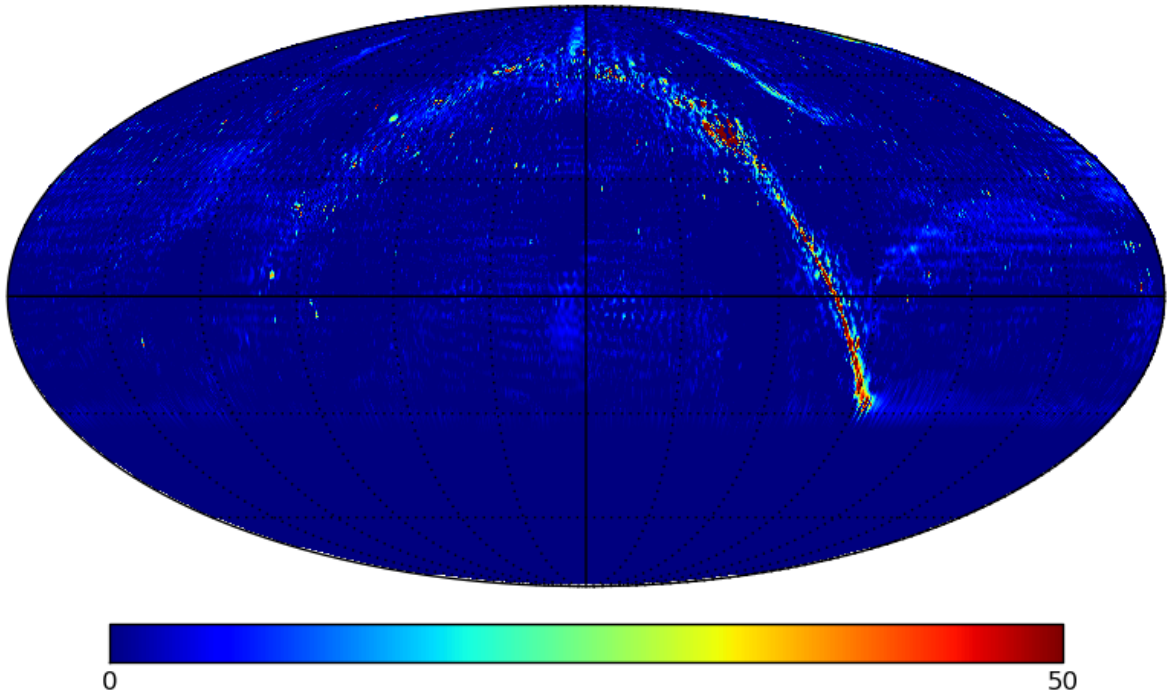
Use a Tikhonov regularization-based m-mode map-making method.

$$V^m_{(ij)} = \sum_{l} B^m_{(ij)\,l} a^m_l + n^m_{(ij)},$$

$$\boldsymbol{v} = \mathbf{B}\,\boldsymbol{a} + \boldsymbol{n}.$$

$$\min_{\boldsymbol{a}} \|\boldsymbol{v} - \mathbf{B}\boldsymbol{a}\|^2 + \varepsilon\|\boldsymbol{a}\|^2$$

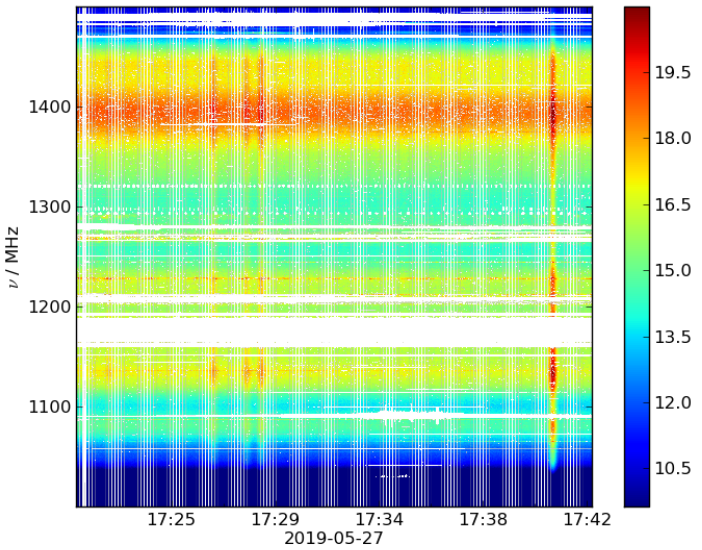
$$\hat{\boldsymbol{a}} = (\mathbf{B}^*\mathbf{B} + \varepsilon\mathbf{I})^{-1}\mathbf{B}^*\boldsymbol{v},$$



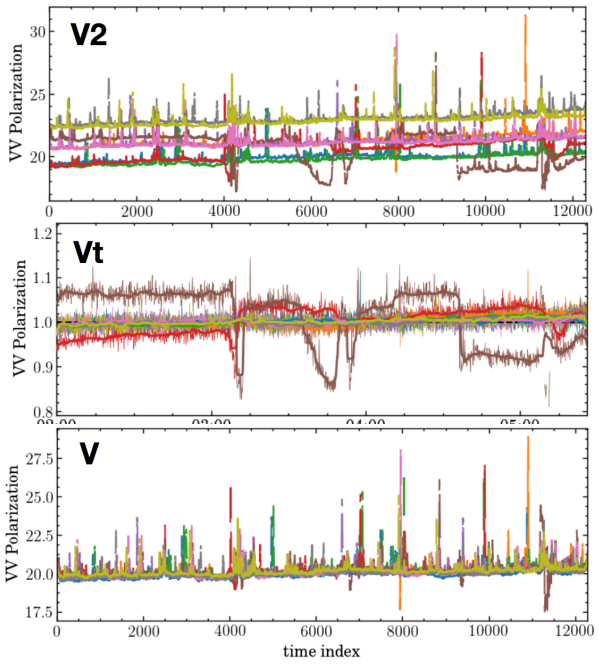
tlpipe for FAST Intensity Mapping

tlpipe is designed to be general and flexible enough that it can also be used to processing data observed by other instruments, for example, the FAST telescope.

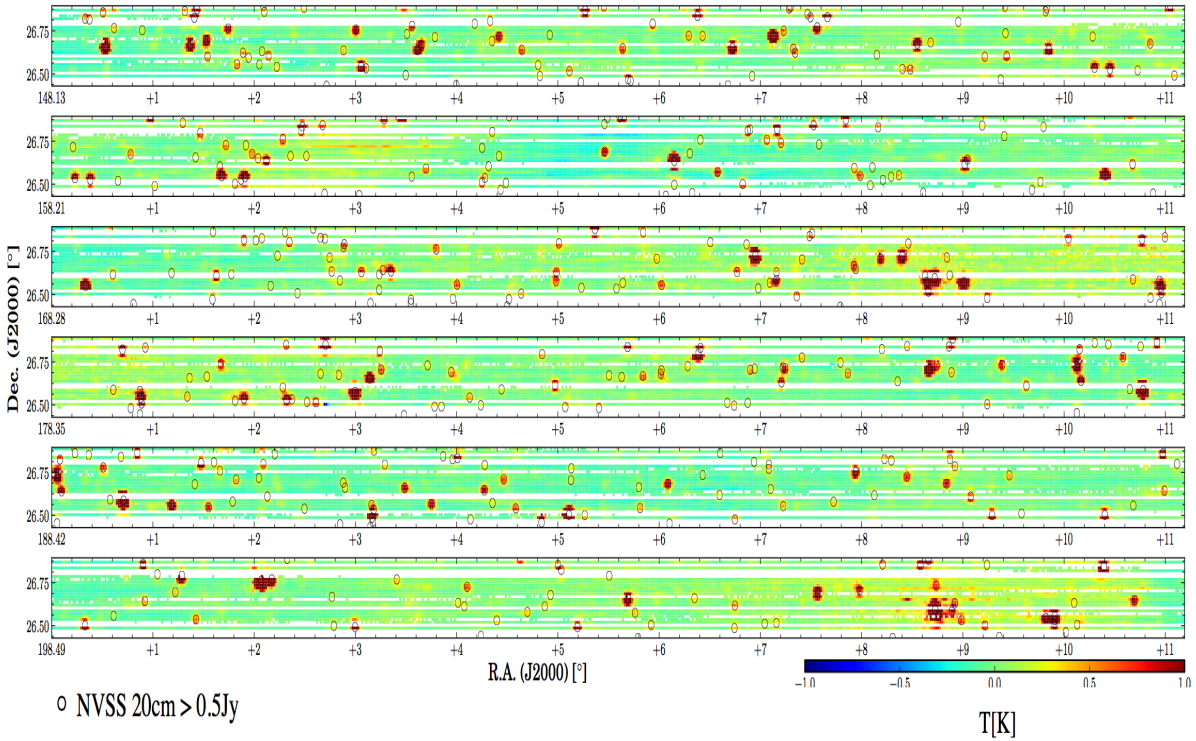
RFI flagging



Calibration



map-making



Summary and Future Development

- We have developed ***tlpipe*** as data processing pipeline for the Tianlai array.
- ***tlpipe*** now has a full set of function modules: reading data file, RFI flagging, calibration, data binning, and final map-making.
- Functionalities like data selection, transformation, visualization are also provided.
- Foreground subtraction and power spectrum estimation are going to be implemented in near future.
- Welcome to use the pipeline and contribute to it.