TOAST Documentation

Release 2.3.0

Theodore Kisner, Reijo Keskitalo

Aug 13, 2019

Contents

Contents:

CHAPTER 1

Introduction

TOAST is a software framework for simulating and processing timestream data collected by telescopes. Telescopes which collect data as timestreams rather than images give us a unique set of analysis challenges. Detector data usually contains noise which is correlated in time as well as sources of correlated signal from the instrument and the environment. Large pieces of data must often be analyzed simultaneously to extract an estimate of the sky signal. TOAST has evolved over several years. The current codebase contains an internal C++ library to allow for optimization of some calculations, while the public interface is written in Python.

The TOAST framework contains:

- Tools for distributing data among many processes
- Tools for performing operations on the local pieces of the data
- Generic operators for common processing tasks (filtering, pointing expansion, map-making)
- Basic classes for performing I/O in a limited set of formats
- Well-defined interfaces for adding custom I/O classes and processing operators

The highest-level control of the workflow is done by the user, often by writing a small Python "pipeline" script (some examples are included). Such pipeline scripts make use of TOAST functions for distributing data and then call built-in or custom operators to process the timestream data.

1.1 Data Organization

The TOAST framework groups data into one or more "observations". Each observation represents data from a group of detectors for some time span. Detectors in the same observation must have the same number of samples for the length of the observation. We currently also assume that the noise properties of the detectors are constant across this observation (i.e. the noise is stationary). A TOAST "dataset" is simply a collection of one or more observations.

1.2 Workflow

Example: Satellite

Example: Ground-Based

1.3 Support for Specific Experiments

If you are a member of one of these projects:

- Planck
- LiteBIRD
- Simons Array
- Simons Observatory
- CMB-S4

Then there are additional software repositories you have access to that contain extra TOAST classes and scripts for processing data from your experiment.

CHAPTER 2

Installation

TOAST is written in C++ and python3 and depends on several commonly available packages. It also has some optional functionality that is only enabled if additional external libraries are available.

2.1 Compiled Dependencies

TOAST compilation requires a C++11 compatible compiler as well as a compatible MPI C++ compiler wrapper. You must also have an FFT library and both FFTW and Intel's MKL are supported by configure checks. Additionally a BLAS/LAPACK installation is required.

Several optional compiled dependencies will enable extra features in TOAST. If the Elemental library is found at configure time then internal atmosphere simulation code will be enabled in the build. If the MADAM destriping mapmaker is available at runtime, then the python code will support calling that library.

2.2 Python Dependencies

You should have a reasonably new (>= 3.4.0) version of python3. We also require several common scientific python packages:

- numpy
- scipy
- matplotlib
- pyephem
- mpi4py (>= 2.0.0)
- healpy

For mpi4py, ensure that this package is compatible with the MPI C++ compiler used during TOAST installation. When installing healpy, you might encounter difficulties if you are in a cross-compile situation. In that case, I recommend installing the repackaged healpix here.

There are obviously several ways to meet these python requirements.

2.2.1 Option #0

If you are using machines at NERSC, see nersc.

2.2.2 Option #1

If you are using a linux distribution which is fairly recent (e.g. the latest Ubuntu version), then you can install all the dependencies with the system package manager:

```
%> apt-get install fftw-dev python3-scipy \
   python3-matplotlib python3-ephem python3-healpy \
   python3-mpi4py
```

On OS X, you can also get the dependencies with macports. However, on some systems OpenMPI from macports is broken and MPICH should be installed as the dependency for the mpi4py package.

2.2.3 Option #2

If your OS is old, you could use a virtualenv to install updated versions of packages into an isolated location. This is also useful if you want to separate your packages from the system installed versions, or if you do not have root access to the machine. Make sure that you have python3 and the corresponding python3-virtualenv packages installed on your system. Also make sure that you have some kind of MPI (OpenMPI or MPICH) installed with your system package manager. Then:

1. create a virtualenv and activate it.
2. once inside the virtualenv, pip install the dependencies

2.2.4 Option #3

Use Anaconda. Download and install Miniconda or the full Anaconda distribution. Make sure to install the Python3 version. If you are starting from Miniconda, install the dependencies that are available through conda:

```
%> conda install -c conda-forge numpy scipy matplotlib mpi4py healpy pyephem
```

2.3 Using Configure

TOAST uses autotools to configure, build, and install both the compiled code and the python tools. If you are running from a git checkout (instead of a distribution tarball), then first do:

```
%> ./autogen.sh
```

Now run configure:

```
%> ./configure --prefix=/path/to/install
```

See the top-level "platforms" directory for other examples of running the configure script. Now build and install the tools:

```
%> make install
```

In order to use the installed tools, you must make sure that the installed location has been added to the search paths for your shell. For example, the "<prefix>/bin" directory should be in your PATH and the python install location "<prefix>/lib/pythonX.X/site-packages" should be in your PYTHONPATH.

2.4 Testing the Installation

After installation, you can run both the compiled and python unit tests. These tests will create an output directory in your current working directory:

```
%> python -c "import toast.tests; toast.tests.run()"
```

CHAPTER 3

Utilities

TOAST contains a variety of utilities for controlling the runtime environment, logging, timing, streamed random number generation, quaternion operations, FFTs, and special function evaluation. In some cases these utilities provide a common interface to compile-time selected vendor math libraries.

3.1 Environment Control

The run-time behavior of the TOAST package can be controlled by the manipulation of several environment variables. The current configuration can also be queried.

class toast.utils.**Environment**

Global runtime environment.

This singleton class provides a unified place to parse environment variables at runtime and to change global settings that impact the overall package.

current_threads(*self: toast._libtoast.Environment*) → int

Return the current threading concurrency in use.

function_timers(*self: toast._libtoast.Environment*) → bool

Return True if function timing has been enabled.

get() → toast._libtoast.Environment

Get a handle to the global environment class.

log_level(*self: toast._libtoast.Environment*) → str

Return the string of the current Logging level.

max_threads(*self: toast._libtoast.Environment*) → int

Returns the maximum number of threads used by compiled code.

print(*self: toast._libtoast.Environment*) → None

Print the current environment to STDOUT.

set_log_level(*self: toast._libtoast.Environment*, *level: str*) → None

Set the Logging level.

Parameters **level** (*str*) – one of DEBUG, INFO, WARNING, ERROR or CRITICAL.

Returns None

set_threads (*self: toast._libtoast.Environment*, *nthread: int*) → None
Set the number of threads in use.

Parameters **nthread** (*int*) – The number of threads to use.

Returns None

signals (*self: toast._libtoast.Environment*) → List[str]
Return a list of the currently available signals.

tod_buffer_length (*self: toast._libtoast.Environment*) → int
Returns the number of samples to buffer for TOD operations.

use_mpi (*self: toast._libtoast.Environment*) → bool
Return True if TOAST was compiled with MPI support **and** MPI is supported in the current runtime environment.

version (*self: toast._libtoast.Environment*) → str
Return the current source code version string.

3.2 Logging

Although python provides logging facilities, those are not accessible to C++. The logging class provided in TOAST is usable from within the compiled libtoast code and also from python, and uses logging level independent from the builtin python logger.

class toast.utils.**Logger**
Simple Logging class.

This class mimics the python logger in C++. The log level is controlled by the TOAST_LOGLEVEL environment variable. Valid levels are DEBUG, INFO, WARNING, ERROR and CRITICAL. The default is INFO.

critical (*self: toast._libtoast.Logger*, *msg: str*) → None
Print a CRITICAL level message.

Parameters **msg** (*str*) – The message to print.

Returns None

debug (*self: toast._libtoast.Logger*, *msg: str*) → None
Print a DEBUG level message.

Parameters **msg** (*str*) – The message to print.

Returns None

error (*self: toast._libtoast.Logger*, *msg: str*) → None
Print an ERROR level message.

Parameters **msg** (*str*) – The message to print.

Returns None

get () → toast._libtoast.Logger
Get a handle to the global logger.

info (*self: toast._libtoast.Logger*, *msg: str*) → None
Print an INFO level message.

Parameters **msg** (*str*) – The message to print.

Returns None

warning (*self: toast._libtoast.Logger*, *msg: str*) → None
Print a WARNING level message.

Parameters **msg** (*str*) – The message to print.

Returns None

3.3 Vector Math Operations

The following functions ...

toast.utils.**vsin** (*in: buffer*, *out: buffer*) → None
Compute the Sine for an array of float64 values.

The results are stored in the output buffer. To guarantee SIMD vectorization, the input and output arrays should be aligned (i.e. use an AlignedF64).

Parameters

- **in** (*array_like*) – 1D array of float64 values.
- **out** (*array_like*) – 1D array of float64 values.

Returns None

3.4 Random Number Generation

The following functions ...

toast._libtoast.**rng_dist_uint64** (*key1: int*, *key2: int*, *counter1: int*, *counter2: int*, *data: buffer*) → None
Generate random unsigned 64bit integers.

The provided input array is populated with values. The dtype of the input array should be compatible with unsigned 64bit integers. To guarantee SIMD vectorization, the input array should be aligned (i.e. use an AlignedU64).

Parameters

- **key1** (*uint64*) – The first element of the key.
- **key2** (*uint64*) – The second element of the key.
- **counter1** (*uint64*) – The first element of the counter.
- **counter2** (*uint64*) – The second element of the counter. This is effectively the sample index in the stream defined by the other 3 values.
- **data** (*array*) – The array to populate.

Returns None.

#workflow.rst #data.rst #tod.rst #intervals.rst #noise.rst #pointing.rst #sim.rst #maptools.rst #timing.rst #nersc.rst #dev.rst

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index