TOAST Documentation

Release 2.3.4

Theodore Kisner, Reijo Keskitalo

Nov 20, 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 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 packages are available. The best installation method will depend on your specific needs. We try to clarify the different options below.

2.1 User Installation

If you are using TOAST to build simulation and analysis workflows, including mixing built-in functionality with your own custom tools, then you can use of these methods to get started. If you want to hack on the TOAST package itself, see the section *Developer Installation*.

If you want to use TOAST at NERSC, see *Using TOAST at NERSC*.

2.1.1 Conda Packages

The easiest way to install TOAST and all of its optional dependencies is to use the conda package manager. The conda-forge ecosystem allows us to create packages that are built consistently with all their dependencies. We recommend following the setup guidelines used by conda-forge, specifically:

1. Install a "miniconda" base system (not the full Anaconda distribution).
2. Set the conda-forge channel to be the top priority package source, with strict ordering if available.
3. Leave the base system (a.k.a. the "root" environment) with just the bare minimum of packages.
4. Always create a new environment (i.e. not the base one) when setting up a python stack for a particular purpose. This allows you to upgrade the conda base system in a reliable way, and to wipe and recreate whole conda environments whenever needed.

Here are the detailed steps of how you could do this from the UNIX shell, installing the base conda system to `${HOME}/conda`. First download the installer. For OS X you would do:

```
curl -SL \
https://repo.anaconda.com/miniconda/Miniconda3-latest-MacOSX-x86_64.sh \
-o miniconda.sh
```

For Linux you would do this:

```
curl -SL \
https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh \
-o miniconda.sh
```

Next we will run the installer. The install prefix should not exist previously:

```
bash miniconda.sh -b -p "${HOME}/conda"
```

Now load this conda "root" environment:

```
source ${HOME}/conda/etc/profile.d/conda.sh
conda activate
```

We are going to make sure to preferentially get packages from the conda-forge channel:

```
conda config --add channels conda-forge
conda config --set channel_priority strict
```

Next, we are going to create a conda environment for a particular purpose (installing TOAST). You can create as many environments as you like and install different packages within them- they are independent. In this example, we will call this environment "toast", but you can call it anything:

```
conda create -y -n toast
```

Now we can activate our new (and mostly empty) toast environment:

```
conda activate toast
```

Finally, we can install the toast package. I recommend installing the MPICH version of TOAST. There is also a version of TOAST without MPI, but most of the parallelism in TOAST comes from using MPI:

```
conda install toast=*=*mpich*
```

OR:

```
conda install toast=*=*nompi*
```

There is also an OpenMPI version of the package, but that is mainly intended for installing toast into environments that also use / require OpenMPI. Assuming this is the only conda installation on your system, you can add the line `source ${HOME}/conda/etc/profile.d/conda.sh` to your shell resource file (usually `~/.bashrc` on Linux or `~/.profile` on OS X). You can read many articles on login shells versus non-login shells and decide where to put this line for your specific use case.

Now you can always activate your toast environment with:

```
conda activate toast
```

And leave that environment with:

```
conda deactivate
```

If you want to use other packages with TOAST (e.g. Jupyter Lab), then you can activate the toast environment and install them with conda. See the conda documentation for more details on managing environments, installing packages, etc.

2.1.2 Minimal Install with PIP

If you cannot or do not want to use the conda package manager, then it is possible to install a "minimal" version of TOAST with pip. If you install TOAST this way, it will be missing support for MPI and atmospheric simulations. Additionally, you must first ensure that you have a serial compiler installed and that a BLAS/LAPACK library is available in the default compiler search paths. You should also install the FFTW package, either through your OS package manager or manually. After doing those steps, you can do:

```
$> pip install https://github.com/hpc4cmb/toast/archive/2.3.5.tar.gz
```

Specify the URL to the version tarball you want to install (see the releases on the TOAST github page).

2.1.3 Something Else

If you have a custom install situation that is not met by the above solutions, then you should follow the instructions below for a "Developer install".

2.2 Developer Installation

Here we will discuss several specific system configurations that are known to work. The best one for you will depend on your OS and preferences.

2.2.1 Ubuntu Linux

You can install all but one required TOAST dependency using packages provided by the OS. Note that this assumes a recent version of ubuntu (tested on 19.04):

```
apt update
apt install \
    cmake \
    build-essential \
    gfortran \
    libopenblas-dev \
    libmpich-dev \
    liblapack-dev \
    libfftw3-dev \
    libsuitesparse-dev \
    python3-dev \
    libpython3-dev \
    python3-scipy \
    python3-matplotlib \
    python3-healpy \
    python3-astropy \
    python3-pyephem
```

NOTE: if you are using another package on your system that requires OpenMPI, then you may get a conflict installing libmpich-dev. In that case, just install libopenmpi-dev instead.

Next, download a release of libaatm and install it. For example:

```
cd libaatm
mkdir build
cd build
cmake \
    -DCMAKE_INSTALL_PREFIX=/usr/local \
    ..
make -j 4
sudo make install
```

You can also install it to the same prefix as TOAST or to a separate location for just the TOAST dependencies. If you install it somewhere other than /usr/local then make sure it is in your environment search paths (see the "installing TOAST" section).

You can also now install the optional dependencies:

- libconviqt for 4PI beam convolution.
- libmadam for optimized destriping mapmaking.

2.2.2 Other Linux

If you have a different distro or an older version of Ubuntu, you should try to install at least these packages with your OS package manager:

```
gcc
g++
mpich or openmpi
lapack
fftw
suitesparse
python3
python3 development library (e.g. libpython3-dev)
virtualenv (e.g. python3-virtualenv)
```

The you can create a python3 virtualenv, activate it, and then use pip to install these packages:

```
pip install \
    scipy \
    matplotlib \
    healpy \
    astropy \
    pyephem
```

Then install libaatm as discussed in the previous section.

2.2.3 Conda Isolated Environment

This is still a work in progress. Conda provides compilers as well as packages, but in order to use them we must isolate **everything** from the surrounding OS. The obvious appeal is that we can then install all dependencies easily and just build TOAST using the conda compilers. We will add more details here after more testing.

2.2.4 OS X with MacPorts

2.2.5 OS X with Homebrew

2.2.6 Full Custom Install with CMBENV

The cmbenv package can generate an install script that selectively compiles packages using specified compilers. This allows you to "pick and choose" what packages are installed from the OS versus being built from source. See the example configs in that package and the README. For example, there is an "ubuntu-19.04" config that gets everything from OS packages but also compiles the optional dependencies like libconviqt and libmadam.

2.2.7 Installing TOAST

Decide where you want to install your development copy of TOAST. I recommend picking a standalone directory somewhere. For this example, we will use `${HOME}/software/toast`. This should **NOT** be the same location as your git checkout.

We want to define a small shell function that will load this directory into our environment. You can put this function in your shell resource file (`~/.bashrc` or `~/.profile`):

```
load_toast () {
    dir="${HOME}/software/toast"
    export PATH="${dir}/bin:${PATH}"
    export CPATH="${dir}/include:${CPATH}"
    export LIBRARY_PATH="${dir}/lib:${LIBRARY_PATH}"
    export LD_LIBRARY_PATH="${dir}/lib:${LD_LIBRARY_PATH}"
    pysite=$(python3 --version 2>&1 | awk '{print $2}' | sed -e "s#\(.*\)\.\(.*\)\..*
↪#\1.\2#")
    export PYTHONPATH="${dir}/lib/python${pysite}/site-packages:${PYTHONPATH}"
}
```

When installing dependencies, you may have chosen to install libaatm, libconviqt, and libmadam into this same location. If so, load this location into your search paths now, before installing TOAST:

```
load_toast
```

TOAST uses CMake to configure, build, and install both the compiled code and the python tools. Within the `toast` git checkout, run the following commands:

```
mkdir -p build && cd build
cmake -DCMAKE_INSTALL_PREFIX=$HOME/software/toast ..
make -j 2 install
```

This will compile and install TOAST in the folder `~/software/toast`. Now, every time you want to use toast, just call the shell function:

```
load_toast
```

If you need to customize the way TOAST gets compiled, the following variables can be defined in the invocation to `cmake` using the `-D` flag:

CMAKE_INSTALL_PREFIX Location where TOAST will be installed. (We used it in the example above.)

CMAKE_C_COMPILER Path to the C compiler

CMAKE_C_FLAGS Flags to be passed to the C compiler (e.g., `-O3`)

CMAKE_CXX_COMPILER Path to the C++ compiler

CMAKE_CXX_FLAGS Flags to be passed to the C++ compiler

MPI_C_COMPILER Path to the MPI wrapper for the C compiler

MPI_CXX_COMPILER Path to the MPI wrapper for the C++ compiler

PYTHON_EXECUTABLE Path to the Python interpreter

BLAS_LIBRARIES Full path to the BLAS dynamical library

LAPACK_LIBRARIES Full path to the LAPACK dynamical library

FFTW_ROOT The install prefix of the FFTW package

SUITESPARSE_INCLUDE_DIR_HINTS The include directory for SuiteSparse headers

SUITESPARSE_LIBRARY_DIR_HINTS The directory containing SuiteSparse libraries

See the top-level "platforms" directory for other examples of running CMake.

2.3 Testing the Installation

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

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

2.4 Building the Documentation

You will need the two Python packages `sphinx` and `sphinx_rtd_theme`, which can be installed using `pip` or `conda` (if you are running Anaconda):

```
cd docs && make clean && make html
```

The documentation will be available in `docs/_build/html`.

CHAPTER 3

Data Model

TOAST works with data organized into *observations*. Each observation is independent of any other observation. An observation consists of co-sampled detectors for some span of time. The intrinsic detector noise is assumed to be stationary within an observation. Typically there are other quantities which are constant for an observation (e.g. elevation, weather conditions, satellite procession axis, etc).

An observation is just a dictionary with at least one member ("tod") which is an instance of a class that derives from the *toast.TOD* base class. Every experiment will have their own TOD derived classes, but TOAST includes some built-in ones as well.

The inputs to a TOD class constructor are at least:

1. The detector names for the observation.
2. The number of samples in the observation.
3. The geometric offset of the detectors from the boresight.
4. Information about how detectors and samples are distributed among processes.

class toast.tod.**TOD**(*mpicomm*, *detectors*, *samples*, *detindx=None*, *detranks=1*, *detbreaks=None*, *sampsizes=None*, *sampbreaks=None*, *meta=None*)

Base class for an object that provides detector pointing and timestreams for a single observation.

This class provides high-level functions that are common to all derived classes. It also defines the internal methods that should be overridden by all derived classes. These internal methods throw an exception if they are called. A TOD base class should never be directly instantiated.

Parameters

- **mpicomm** (*mpi4py.MPI.Comm*) – the MPI communicator over which the data is distributed, or None.
- **detectors** (*list*) – The list of detector names.
- **samples** (*int*) – The total number of samples.
- **detindx** (*dict*) – the detector indices for use in simulations. Default is { x[0] : x[1] for x in zip(detectors, range(len(detectors))) }.

- **detranks** (*int*) – The dimension of the process grid in the detector direction. If not None, the MPI communicator size must be evenly divisible by this number.
- **detbreaks** (*list*) – Optional list of hard breaks in the detector distribution.
- **sampsizes** (*list*) – Optional list of sample chunk sizes which cannot be split.
- **sampbreaks** (*list*) – Optional list of hard breaks in the sample distribution.
- **meta** (*dict*) – Optional dictionary of metadata properties.

COMMON_FLAG_NAME = 'common_flags'
Default cache name for common flags.

FLAG_NAME = 'flags'
Default cache name for flags.

HWP_ANGLE_NAME = 'hwp_angle'
Default cache name for HWP angle.

POINTING_NAME = 'quat'
Default cache name for pointing quaternions.

POSITION_NAME = 'position'
Default cache name for position.

SIGNAL_NAME = 'signal'
Default cache name for signal.

TIMESTAMP_NAME = 'timestamps'
Default cache name for timestamps.

VELOCITY_NAME = 'velocity'
Default cache name for velociyt.

detectors
The total list of detectors.

Type (list)

detindx
The detector indices.

Type (dict)

detoffset()
Return dictionary of detector quaternions.

This returns a dictionary with the detector names as the keys and the values are 4-element numpy arrays containing the quaternion offset from the boresight.

Parameters **None** –

Returns (dict): the dictionary of quaternions.

dist_chunks
this is a list of 2-tuples, one for each column of the process grid. Each element of the list is the same as the information returned by the "local_chunks" member for a given process column.

Type (list)

dist_samples
This is a list of 2-tuples, with one element per column of the process grid. Each tuple is the same information returned by the "local_samples" member for the corresponding process grid column rank.

Type (list)

grid_comm_col
a communicator across all detectors in the same column of the process grid (or None).

Type (mpi4py.MPI.Comm)

grid_comm_row
a communicator across all detectors in the same row of the process grid (or None).

Type (mpi4py.MPI.Comm)

grid_ranks
the ranks of this process in the (detector, sample) directions.

Type (tuple)

grid_size
the dimensions of the process grid in (detector, sample) directions.

Type (tuple)

local_chunks
the first element of the tuple is the index of the first chunk assigned to this process (i.e. the index in the list given by the "total_chunks" member). The second element of the tuple is the number of chunks assigned to this process.

Type (2-tuple)

local_common_flags(*name=None*, ***kwargs*)
Locally stored common flags.

Parameters **name** (*str*) – Optional cache key to use.

Returns A cache reference to a common flag vector. If 'name' is None a default name 'common_flags' is used and the vector may be constructed and cached using the 'read_common_flags' method. If 'name' is given, then the flags must already be cached.

local_dets
The detectors assigned to this process.

Type (list)

local_flags(*det*, *name=None*, ***kwargs*)
Locally stored flags.

Parameters

- **det** (*str*) – Name of the detector.
- **name** (*str*) – Optional cache key to use.

Returns A cache reference to a flag vector. If 'name' is None a default name 'flags' is used and the vector may be constructed and cached using the 'read_flags' method. If 'name' is given, then the flags must already be cached.

local_hwp_angle(*name=None*, ***kwargs*)
Locally stored half-wave plate angle.

Parameters **name** (*str*) – Optional cache key to use.

Returns A cache reference to a hwp angle vector. If 'name' is None a default name 'hwp_angle' is used and the vector may be constructed and cached using the 'read_hwp_angle' method. If 'name' is given, then the angles must already be cached.

local_intervals(*intervals*)
Translate observation-wide intervals into local sample indices.

local_pointing(*det*, *name=None*, ***kwargs*)
Locally stored pointing.

Parameters

- **det** (*str*) – Name of the detector.
- **name** (*str*) – Optional cache key to use.

Returns A cache reference to a pointing array. If 'name' is None a default name 'quat' is used and the array may be constructed and cached using the 'read_pntg' method. If 'name' is given, then the pointing must already be cached.

local_position(*name=None*, ***kwargs*)
Locally stored position.

Parameters **name** (*str*) – Optional cache key to use.

Returns A cache reference to a position array. If 'name' is None a default name 'position' is used and the array may be constructed and cached using the 'read_position' method. If 'name' is given, then the position must already be cached.

local_samples
The first element of the tuple is the first global sample assigned to this process. The second element of the tuple is the number of samples assigned to this process.

Type (2-tuple)

local_signal(*det*, *name=None*, ***kwargs*)
Locally stored signal.

Parameters

- **det** (*str*) – Name of the detector.
- **name** (*str*) – Optional cache key to use.

Returns A cache reference to a signal vector. If 'name' is None a default name 'signal' is used and the vector may be constructed and cached using the 'read' method. If 'name' is given, then the signal must already be cached.

local_times(*name=None*, ***kwargs*)
Timestamps covering locally stored data.

Parameters **name** (*str*) – Optional cache key to use.

Returns A cache reference to a timestamp vector. If 'name' is None a default name 'timestamps' is used and the vector may be constructed and cached using the 'read_times' method. If 'name' is given, then the times must already be cached.

local_velocity(*name=None*, ***kwargs*)
Locally stored velocity.

Parameters **name** (*str*) – Optional cache key to use.

Returns A cache reference to a velocity array. If 'name' is None a default name 'velocity' is used and the array may be constructed and cached using the 'read_velocity' method. If 'name' is given, then the velocity must already be cached.

mpicomm
the communicator assigned to this TOD.

Type (mpi4py.MPI.Comm)

read(*detector=None*, *local_start=0*, *n=0*, ***kwargs*)
Read detector data.

This returns the timestream data for a single detector.

Parameters

- **detector** (*str*) – the name of the detector.
- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.
- **n** (*int*) – the number of samples to read. If zero, read to end.

Returns An array containing the data.

read_boresight(*local_start=0*, *n=0*, ***kwargs*)
Read boresight quaternion pointing.

This returns the pointing of the boresight in quaternions.

Parameters

- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.
- **n** (*int*) – the number of samples to read. If zero, read to end.

Returns A 2D array of shape (n, 4)

read_boresight_azel(*local_start=0*, *n=0*, ***kwargs*)
Read boresight Azimuth / Elevation quaternion pointing.

This returns the pointing of the boresight in the horizontal coordinate system, if it exists.

Parameters

- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.
- **n** (*int*) – the number of samples to read. If zero, read to end.

Returns A 2D array of shape (n, 4)

Raises `NotImplementedError` – if the telescope is not on the Earth.

read_common_flags(*local_start=0*, *n=0*, ***kwargs*)
Read common flags.

This reads the common set of flags that should be applied to all detectors.

Parameters

- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.
- **n** (*int*) – the number of samples to read. If zero, read to end.

Returns a numpy array containing the flags.

Return type (array)

read_flags(*detector=None*, *local_start=0*, *n=0*, ***kwargs*)
Read detector flags.

This returns the detector-specific flags.

Parameters

- **detector** (*str*) – the name of the detector.
- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.

- **n** (*int*) – the number of samples to read. If zero, read to end.

Returns An array containing the detector flags.

read_hwp_angle(*local_start=0*, *n=0*, ***kwargs*)
Read half-wave plate angle

This reads the common HWP angle that should be applied to all detectors.

Parameters

- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.
- **n** (*int*) – the number of samples to read. If zero, read to end.

Returns

a numpy array containing the angles or None if the angle is not defined.

Return type (array)

read_pntg(*detector=None*, *local_start=0*, *n=0*, ***kwargs*)
Read detector quaternion pointing.

This returns the pointing for a single detector in quaternions.

Parameters

- **detector** (*str*) – the name of the detector.
- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.
- **n** (*int*) – the number of samples to read. If zero, read to end.

Returns A 2D array of shape (n, 4)

read_position(*local_start=0*, *n=0*, ***kwargs*)
Read telescope position.

This reads the telescope position in solar system barycenter coordinates (in Kilometers).

Parameters

- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.
- **n** (*int*) – the number of samples to read. If zero, read to end.

Returns

a 2D numpy array containing the x,y,z coordinates at each sample.

Return type (array)

read_times(*local_start=0*, *n=0*, ***kwargs*)
Read timestamps.

This reads the common set of timestamps that apply to all detectors in the TOD.

Parameters

- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.
- **n** (*int*) – the number of samples to read. If zero, read to end.

Returns a numpy array containing the timestamps.

Return type (array)

read_velocity(*local_start=0*, *n=0*, ***kwargs*)
Read telescope velocity.

This reads the telescope velocity in solar system barycenter coordinates (in Kilometers/s).

Parameters

- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.
- **n** (*int*) – the number of samples to read. If zero, read to end.

Returns

a 2D numpy array containing the x,y,z velocity components at each sample.

Return type (array)

total_chunks
the full list of sample chunk sizes that were used in the data distribution.

Type (list)

total_samples
the total number of samples in this TOD.

Type (int)

write(*detector=None*, *local_start=0*, *data=None*, ***kwargs*)
Write detector data.

This writes the detector data.

Parameters

- **detector** (*str*) – the name of the detector.
- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.
- **data** (*array*) – the data array.

write_boresight(*local_start=0*, *data=None*, ***kwargs*)
Write boresight quaternion pointing.

This writes the quaternion pointing for the boresight.

Parameters

- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.
- **data** (*array*) – 2D array of quaternions with shape[1] == 4.

write_boresight_azel(*local_start=0*, *data=None*, ***kwargs*)
Write boresight Azimuth / Elevation quaternion pointing.

This writes the quaternion pointing for the boresight in the horizontal coordinate system, if it exists.

Parameters

- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.
- **data** (*array*) – 2D array of quaternions with shape[1] == 4.

Raises *RuntimeError or AttributeError* – if the telescope is not on the Earth.

write_common_flags(*local_start=0*, *flags=None*, ***kwargs*)
Write common flags.

This writes the common set of flags that should be applied to all detectors.

Parameters

- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.
- **flags** (*array*) – array containing the flags to write.

write_flags(*detector=None*, *local_start=0*, *flags=None*, ***kwargs*)
Write detector flags.

This writes the detector-specific flags.

Parameters

- **detector** (*str*) – the name of the detector.
- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.
- **flags** (*array*) – the detector flags.

write_hwp_angle(*local_start=0*, *hwpangle=None*, ***kwargs*)
Write half-wave plate angle

This writes the common HWP angle that should be applied to all detectors.

Parameters

- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.
- **flags** (*array*) – array containing the flags to write.

write_pntg(*detector=None*, *local_start=0*, *data=None*, ***kwargs*)
Write detector quaternion pointing.

This writes the quaternion pointing for a single detector.

Parameters

- **detector** (*str*) – the name of the detector.
- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.
- **data** (*array*) – 2D array of quaternions with shape[1] == 4.

write_position(*local_start=0*, *pos=None*, ***kwargs*)
Write telescope position.

This writes the telescope position in solar system barycenter coordinates (in Kilometers).

Parameters

- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.
- **pos** (*array*) – the 2D array of x,y,z coordinates at each sample.

write_times(*local_start=0*, *stamps=None*, ***kwargs*)
Write timestamps.

This writes the common set of timestamps that apply to all detectors in the TOD.

Parameters

- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.
- **stamps** (*array*) – the array of timestamps to write.

write_velocity(*local_start=0*, *vel=None*, ***kwargs*)
Write telescope velocity.

This writes the telescope velocity in solar system barycenter coordinates (in Kilometers/s).

Parameters

- **local_start** (*int*) – the sample offset relative to the first locally assigned sample.
- **vel** (*array*) – the 2D array of x,y,z velocity components at each sample.

The TOD class can act as a storage container for different "flavors" of timestreams as well as a source and sink for the observation data (with the **read_***() and **write_***() methods). The TOD base class has one member which is a *Cache* class.

class toast.cache.**Cache**(*pymem=False*)

Data cache with explicit memory management.

This class acts as a dictionary of named arrays. Each array may be multi-dimensional.

Parameters **pymem** (*bool*) – if True, use python memory rather than external allocations in C. Only used for testing.

add_alias(*alias*, *name*)

Add an alias to a name that already exists in the cache.

Parameters

- **alias** (*str*) – alias to create
- **name** (*str*) – an existing key in the cache

Returns None

aliases()

Return a dictionary of all the aliases to keys in the cache.

Returns Dictionary of aliases.

Return type (dict)

clear(*pattern=None*)

Clear one or more buffers.

Parameters **pattern** (*str*) – a regular expression to match against the buffer names when determining what should be cleared. If None, then all buffers are cleared.

Returns None

create(*name*, *type*, *shape*)

Create a named data buffer of the given type and shape.

Parameters

- **name** (*str*) – the name to assign to the buffer.
- **type** (*numpy.dtype*) – one of the supported numpy types.
- **shape** (*tuple*) – a tuple containing the shape of the buffer.

Returns a reference to the allocated array.

Return type (array)

destroy(*name*)

Deallocate the specified buffer.

Only call this if all numpy arrays that reference the memory are out of use. If the specified name is an alias, then the alias is simply deleted. If the specified name is an actual buffer, then all aliases pointing to that buffer are also deleted.

Parameters **name** (*str*) – the name of the buffer or alias to destroy.

Returns None

exists(*name*)
Check whether a buffer exists.

Parameters **name** (*str*) – the name of the buffer to search for.

Returns True if a buffer or alias exists with the given name.

Return type (bool)

keys()
Return a list of all the keys in the cache.

Returns List of key strings.

Return type (list)

put(*name*, *data*, *replace=False*)
Create a named data buffer to hold the provided data.

If replace is True, existing buffer of the same name is first destroyed. If replace is True and the name is an alias, it is promoted to a new data buffer.

Parameters

- **name** (*str*) – the name to assign to the buffer.
- **data** (*numpy.ndarray*) – Numpy array
- **replace** (*bool*) – Overwrite any existing keys

Returns a numpy array wrapping the raw data buffer.

Return type (array)

reference(*name*)
Return a numpy array pointing to the buffer.

The returned array will wrap a pointer to the raw buffer, but will not claim ownership. When the numpy array is garbage collected, it will NOT attempt to free the memory (you must manually use the destroy method).

Parameters **name** (*str*) – the name of the buffer to return.

Returns a numpy array wrapping the raw data buffer.

Return type (array)

report(*silent=False*)
Report memory usage.

Parameters **silent** (*bool*) – Count and return the memory without printing.

Returns Amount of allocated memory in bytes

Return type (int)

This class looks like a dictionary of numpy arrays, but the memory is allocated outside of Python, which means it can be explicitly managed / freed. This *cache* member is where alternate flavors of the timestream data are stored.

Each observation can also have a noise model associated with it. An instance of a Noise class (or derived class) describes the noise properties for all detectors in the observation.

class toast.tod.**Noise**(*, *detectors*, *freqs*, *psds*, *mixmatrix=None*, *indices=None*)
Noise objects act as containers for noise PSDs.

Noise is a base class for an object that describes the noise properties of all detectors for a single observation.

Parameters

- **detectors** (*list*) – Names of detectors.
- **freqs** (*dict*) – Dictionary of arrays of frequencies for *psds*.
- **psds** (*dict*) – Dictionary of arrays which contain the PSD values for each detector or *mixmatrix* key.
- **mixmatrix** (*dict*) – Mixing matrix describing how the PSDs should be combined for detector noise. If provided, must contain entries for every detector, and every key specified for a detector must be defined in *freqs* and *psds*.
- **indices** (*dict*) – Integer index for every PSD, useful for generating indepedendent and repeateable noise realizations. If absent, runnign indices will be assigned and provided.

detectors
List of detector names

Type list

keys
List of PSD names

Type list

Raises

- KeyError – If *freqs*, *psds*, *mixmatrix* or *indices* do not include all relevant entries.
- ValueError – If vector lengths in *freqs* and *psds* do not match.

detectors
list of strings containing the detector names.

Type (list)

freq(*key*)
Get the frequencies corresponding to *key*.

Parameters **key** (*str*) – Detector name or mixing matrix key.

Returns Frequency bins that are used for the PSD.

Return type (array)

index(*key*)
Return the PSD index for *key*

Parameters **key** (*std*) – Detector name or mixing matrix key.

Returns PSD index.

Return type index (int)

keys
list of strings containing the PSD names.

Type (list)

multiply_invntt(*key*, *data*)
Filter the data with inverse noise covariance.

multiply_ntt(*key*, *data*)
Filter the data with noise covariance.

psd(*key*)
Get the PSD corresponding to *key*.

Parameters **key** (*str*) – Detector name or mixing matrix key.

Returns PSD matching the key.

Return type (array)

rate(*key*)
Get the sample rate for *key*.

Parameters **key** (*str*) – the detector name or mixing matrix key.

Returns the sample rate in Hz.

Return type (float)

weight(*det*, *key*)
Return the mixing weight for noise *key* in *det*.

Parameters

- **det** (*str*) – Detector name
- **key** (*std*) – Mixing matrix key.

Returns Mixing matrix weight

Return type weight (float)

The data used by a TOAST workflow consists of a list of observations, and is encapsulated by the *toast.Data* class.

class toast.dist.**Data**(*comm=<toast.Comm World MPI communicator = None World MPI size = 1 World MPI rank = 0 Group MPI communicator = None Group MPI size = 1 Group MPI rank = 0 Rank MPI communicator = None >*)
Class which represents distributed data

A Data object contains a list of observations assigned to each process group in the Comm.

Parameters **comm** (toast.Comm) – the toast Comm class for distributing the data.

clear()
Clear the list of observations.

comm
The toast.Comm over which the data is distributed.

info(*handle=None*, *flag_mask=255*, *common_flag_mask=255*, *intervals=None*)
Print information about the distributed data.

Information is written to the specified file handle. Only the rank 0 process writes. Optional flag masks are used when computing the number of good samples.

Parameters

- **handle** (*descriptor*) – file descriptor supporting the write() method. If None, use print().
- **flag_mask** (*int*) – bit mask to use when computing the number of good detector samples.
- **common_flag_mask** (*int*) – bit mask to use when computing the number of good telescope pointings.
- **intervals** (*str*) – optional name of an intervals object to print from each observation.

Returns None

obs = None
The list of observations.

split(*key*)
Split the Data object.

Split the Data object based on the value of *key* in the observation dictionary.

Parameters **key** (*str*) – Observation key to use.

Returns List of 2-tuples of the form (value, data)

If you are running with a single process, that process has all observations and all data within each observation locally available. If you are running with more than one process, the data with be distributed across processes.

3.1 Data Distribution

Although you can use TOAST without MPI, the package is designed for data that is distributed across many processes. When passing the data through a toast workflow, the data is divided up among processes based on the details of the *toast.Comm* class that is used and also the shape of the process grid in each observation.

A toast.Comm instance takes the global number of processes available (MPI.COMM_WORLD) and divides them into groups. Each process group is assigned one or more observations. Since observations are independent, this means that different groups can be independently working on separate observations in parallel. It also means that inter-process communication needed when working on a single observation can occur with a smaller set of processes.

class toast.mpi.**Comm**(*world=None*, *groupsize=0*)
Class which represents a two-level hierarchy of MPI communicators.

A Comm object splits the full set of processes into groups of size "group". If group_size does not divide evenly into the size of the given communicator, then those processes remain idle.

A Comm object stores three MPI communicators: The "world" communicator given here, which contains all processes to consider, a "group" communicator (one per group), and a "rank" communicator which contains the processes with the same group-rank across all groups.

If MPI is not enabled, then all communicators are set to None.

Parameters

- **world** (*mpi4py.MPI.Comm*) – the MPI communicator containing all processes.
- **group** (*int*) – the size of each process group.

comm_group
The communicator shared by processes within this group.

comm_rank
The communicator shared by processes with the same group_rank.

comm_world
The world communicator.

group
The group containing this process.

group_rank
The rank of this process in the group communicator.

group_size
The size of the group containing this process.

ngroups
The number of process groups.

world_rank
The rank of this process in the world communicator.

world_size
The size of the world communicator.

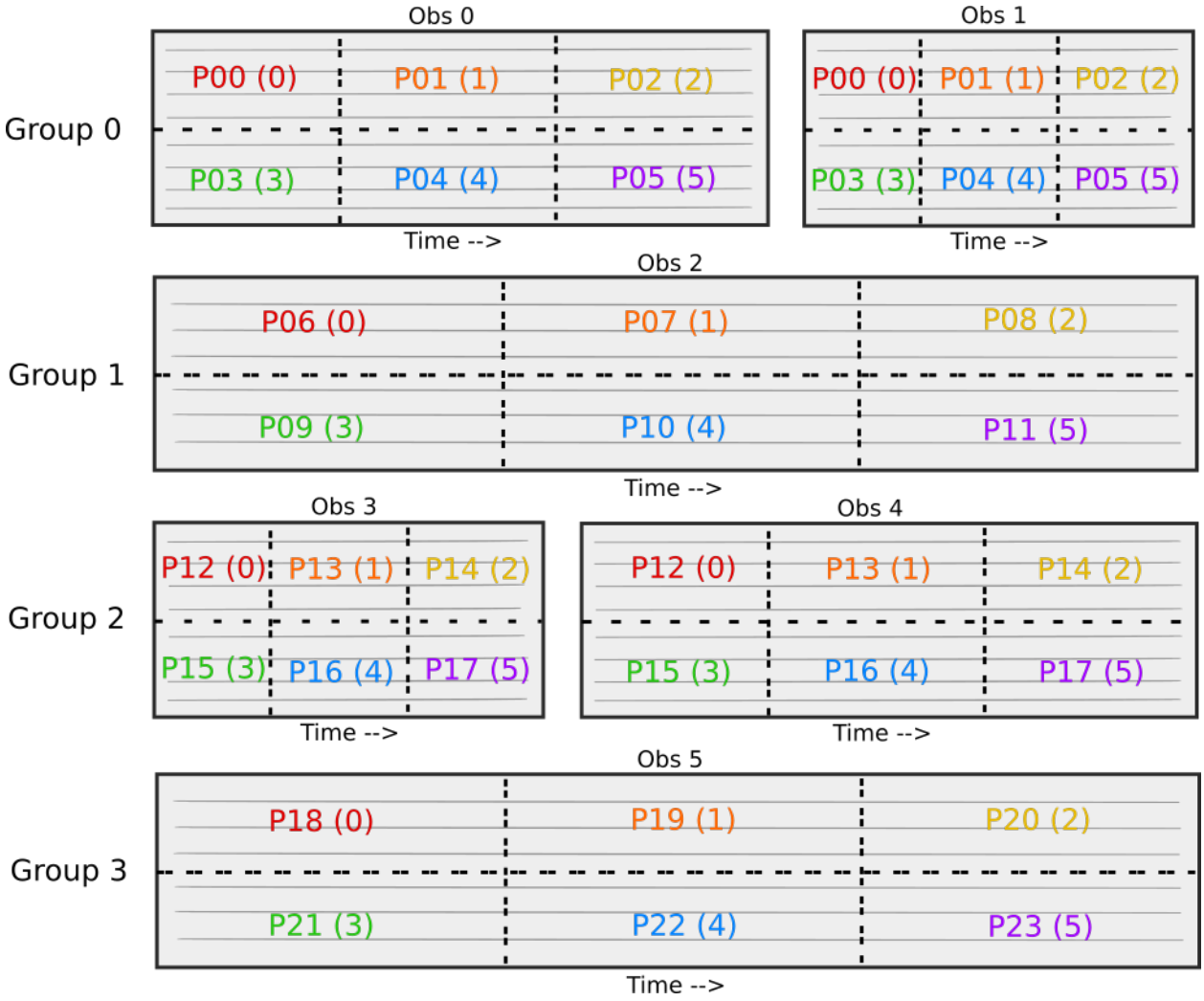
Just to reiterate, if your *toast.Comm* has multiple process groups, then each group will have an independent list of observations in *toast.Data.obs*.

What about the data *within* an observation? A single observation is owned by exactly one of the process groups. The MPI communicator passed to the TOD constructor is the group communicator. Every process in the group will store some piece of the observation data. The division of data within an observation is controlled by the *detranks* option to the TOD constructor. This option defines the dimension of the rectangular "process grid" along the detector (as opposed to time) direction. Common values of *detranks* are:

- "1" (processes in the group have all detectors for some slice of time)
- Size of the group communicator (processes in the group have some of the detectors for the whole time range of the observation)

The detranks parameter must divide evenly into the number of processes in the group communicator.

As a concrete example, imagine that MPI.COMM_WORLD has 24 processes. We split this into 4 groups of 6 processs. There are 6 observations of varying lengths and every group has one or 2 observations. Here is a picture of what data each process would have. The global process number is shown as well as the rank within the group:



In either case the full dataset is divided into one or more observations, and each observation has one TOD object (and optionally other objects that describe the noise, valid data intervals, etc). The toast "Comm" class has two levels of MPI communicators that can be used to divide many observations between whole groups of processes. In practice this is not always needed, and the default construction of the Comm object just results in one group with all processes.

CHAPTER 4

Pipelines

TOAST workflows are usually called "pipelines" and consist of a `toast.Data` object that is passed through one or more "operators":

class toast.**Operator**

Base class for an operator that acts on collections of observations.

An operator takes as input a toast.dist.Data object and returns a new instance of the same size. For each observation in the distributed data, an operator may pass some data types forward unchanged, or it may replace or modify data.

Parameters **None** –

There are very few restrictions on an "operator" class. It can have arbitrary constructor arguments and must define an *exec()* method which takes a *toast.Data* instance.

Each operator might take many arguments. There are helper functions in *toast.pipeline_tools* that can be used to create an operator in a pipeline. Currently these helper functions add arguments to *argparse* for control at the command line. In the future, we intend to support loading operator configuration from other config file formats.

4.1 Example: Simple Satellite Simulation

TOAST includes several "generic" pipelines that simulate some fake data and then run some operators on that data. One of these is installed as *toast_satellite_sim.py*. There is some "set up" in the top of the script, but if we remove the timing code then the *main()* looks like this:

```
def main():
    env = Environment.get()
    log = Logger.get()

    mpiworld, procs, rank, comm = pipeline_tools.get_comm()
    args, comm, groupsize = parse_arguments(comm, procs)

    # Parse options
```

(continues on next page)

(continued from previous page)

```
if comm.world_rank == 0:
    os.makedirs(args.outdir, exist_ok=True)

focalplane, gain, detweights = load_focalplane(args, comm)

data = create_observations(args, comm, focalplane, groupsize)

pipeline_tools.expand_pointing(args, comm, data)

signalname = None
skyname = pipeline_tools.simulate_sky_signal(
    args, comm, data, [focalplane], "signal"
)
if skyname is not None:
    signalname = skyname

skyname = pipeline_tools.apply_conviqt(args, comm, data, "signal")
if skyname is not None:
    signalname = skyname

diponame = pipeline_tools.simulate_dipole(args, comm, data, "signal")
if diponame is not None:
    signalname = diponame

# Mapmaking.

if not args.use_madam:
    if comm.world_rank == 0:
        log.info("Not using Madam, will only make a binned map")

    npp, zmap = pipeline_tools.init_binner(args, comm, data, detweights)

    # Loop over Monte Carlos

    firstmc = args.MC_start
    nmc = args.MC_count

    for mc in range(firstmc, firstmc + nmc):
        outpath = os.path.join(args.outdir, "mc_{:03d}".format(mc))

        pipeline_tools.simulate_noise(
            args, comm, data, mc, "tot_signal", overwrite=True
        )

        # add sky signal
        pipeline_tools.add_signal(args, comm, data, "tot_signal", signalname)

        if gain is not None:
            op_apply_gain = OpApplyGain(gain, name="tot_signal")
            op_apply_gain.exec(data)

        if mc == firstmc:
            # For the first realization, optionally export the
            # timestream data.  If we had observation intervals defined,
            # we could pass "use_interval=True" to the export operators,
            # which would ensure breaks in the exported data at
```

(continues on next page)

(continued from previous page)

```
                # acceptable places.
                pipeline_tools.output_tidas(args, comm, data, "tot_signal")
                pipeline_tools.output_spt3g(args, comm, data, "tot_signal")

            pipeline_tools.apply_binner(
                args, comm, data, npp, zmap, detweights, outpath, "tot_signal"
            )

    else:

        # Initialize madam parameters

        madampars = pipeline_tools.setup_madam(args)

        # Loop over Monte Carlos

        firstmc = args.MC_start
        nmc = args.MC_count

        for mc in range(firstmc, firstmc + nmc):
            # create output directory for this realization
            outpath = os.path.join(args.outdir, "mc_{:03d}".format(mc))

            pipeline_tools.simulate_noise(
                args, comm, data, mc, "tot_signal", overwrite=True
            )

            # add sky signal
            pipeline_tools.add_signal(args, comm, data, "tot_signal", signalname)

            if gain is not None:
                op_apply_gain = OpApplyGain(gain, name="tot_signal")
                op_apply_gain.exec(data)

            pipeline_tools.apply_madam(
                args, comm, data, madampars, outpath, detweights, "tot_signal"
            )

            if comm.comm_world is not None:
                comm.comm_world.barrier()
```

CHAPTER 5

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.

5.1 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.

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.

5.2 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

5.3 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

5.4 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.

CHAPTER 6

Using TOAST at NERSC

A recent version of TOAST is already installed at NERSC, along with all necessary dependencies. You can use this installation directly, or use it as the basis for your own development.

6.1 Module Files

To get access to the needed module files, add the machine-specific module file location to your search path:

```
module use /global/common/software/cmb/${NERSC_HOST}/default/modulefiles
```

The *default* part of this path is a symlink to the latest stable installation. There are usually several older versions kept here as well.

You can safely put the above line in your ~/.bashrc.ext inside the section for cori. It does not actually load anything into your environment.

6.2 Loading the Software

To load the software do the following:

```
module load cmbenv source cmbenv
```

Note that the "source" command above is not "reversible" like normal module operations. This is required in order to activate the underlying conda environment. After running the above commands, TOAST and many other common software tools will be in your environment, including a Python3 stack.

6.3 Installing TOAST (Optional)

The cmbenv stack contains a recent version of TOAST, but if you want to build your own copy then you can use the cmbenv stack as a starting point. Here are the steps:

1. Decide on the installation location. You should install software either to one of the project software spaces in */global/common/software* or in your home directory. If you plan on using this installation for large parallel jobs, you should install to */global/common/software*.
2. Load the cmbenv stack.
3. Go into your git checkout of TOAST and make a build directory:

   cd toast mkdir build cd build

4. Use the cori-intel platform file to build TOAST and install:

   ../platforms/cori-intel.sh -DCMAKE_INSTALL_PREFIX=/path/to/somewhere make -j 4 install

5. Set up a shell function in *~/.bashrc.ext* to load this into your environment search paths before the cmbenv stack:

   load_toast () { dir=/path/to/your/install export PATH="${dir}/bin:${PATH}" pysite=$(python3 –version 2>&1 | awk '{print $2}' | sed -e "s#(.*).(.*)..*#1.2#") export PYTHONPATH="${dir}/lib/python${pysite}/site-packages:${PYTHONPATH}"

   Definition list ends without a blank line; unexpected unindent.

   }

Now whenever you want to override the cmbenv TOAST installation you can just do:

load_toast

#intervals.rst #noise.rst #pointing.rst #sim.rst #maptools.rst #timing.rst #dev.rst

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index