TOAST Documentation

Release 2.3.10.dev2

Theodore Kisner, Reijo Keskitalo, Andrea Zonca

Nov 06, 2020

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

Change Log

2.1 2.3.10 (2020-10-15)

- Run serial unit tests without MPI. Fix bug in ground filter (PR #370).

2.2 2.3.9 (2020-10-15)

- Add stand-alone benchmarking tool (PR #365).
- Update wheels to use latest OpenBLAS and SuiteSparse (PR #368).
- Tweaks to atmosphere simulation based on calibration campaign (PR #367).
- Add support for 2D polynomial filtering across focalplane (PR #366).
- Ground scheduler support for elevation modulated scans (PR #364).
- Add better dictionary interface to Cache class (PR #363).
- Support simulating basic non-ideal HWP response (PR #362).
- Ground scheduler support for fixed elevations and partial scans (PR #361).
- Additional check for NULL plan returned from FFTW (PR #360).

2.3 2.3.8 (2020-06-27)

- Minor release focusing on build system changes to support packaging
- Update bundled pybind11 and other fixes for wheels and conda packages (PR #359).

2.4 2.3.7 (2020-06-13)

- Documentation updates and deployment of pip wheels on tags (PR #356).
- Cleanups of conviqt polarization support (PR #347).
- Support elevation nods in ground simulations (PR #355).
- Fix a bug in parallel writing of Healpix FITS files (PR #354).
- Remove dependence on MPI compilers. Only mpi4py is needed (PR #350).
- Use the native mapmaker by default in the example pipelines (PR #352).
- Updates to build system for pip wheel compatibility (PR #348, #351).
- Switch to github actions instead of travis for continuous integration (PR #349).
- Updates to many parts of the simulation and filtering operators (PR #341).
- In the default Healpix pointing matrix, support None for HWP angle (PR #345).
- Add support for HWP in conviqt beam convolution (PR #343).
- Reimplementation of example jobs used for benchmarks (PR #332).
- Apply atmosphere scaling in temperature, not intensity (PR #328).
- Minor bugfix in binner when running in debug mode (PR #325).
- Add optional boresight offset to the scheduler (PR #329).
- Implement helper tools for parsing mapmaker options (PR #321).

2.5 2.3.6 (2020-01-19)

- Overhaul documentation (PR #320).
- Small typo fix for conviqt operator (PR #319).
- Support high-cadence ground scan strategies and fix a bug in turnaround simulation (PR #316).
- Fix BLAS / LAPACK name mangling detection (PR #315).
- Allow disabling sky sim in example pipeline (PR #313).

2.6 2.3.5 (2019-11-19)

- Documentation updates (PR #310).

2.7 2.3.4 (2019-11-17)

- Disabling timing tests during build of conda package.

2.8 2.3.3 (2019-11-16)

- Change way that the MPI communicator is passed to C++ (PR #309).

2.9 2.3.2 (2019-11-13)

- Convert atmosphere simulation to new libaatm package (PR #307).
- Improve vector math unit tests (PR #296).
- Updates to conviqt operator (PR #304).
- Satellite example pipeline cleanups.
- Store local pixel information in the data dictionary (PR #306).
- Add elevation-dependent noise (PR #303).
- Move global / local pixel lookup into compiled code (PR #302).
- PySM operator changes to communicator (PR #301).
- Install documentation updates (PR #300, #299).

2.10 2.3.1 (2019-10-14)

- Fix bug when writing FITS maps serially.
- Improve printing of Environment and TOD classes (PR #294).
- Fix a race condition (PR #292).
- Control the Numba backend from TOAST (PR #283, #291).
- Functional TOAST map-maker (PR #288).
- Large improvements to ground sim example (PR #290).
- Overhaul examples to match 2.3.0 changes (PR #286).
- Handle small angles and improve unit tests for healpix.

2.11 2.3.0 (2019-08-13)

- Rewrite of internal compiled codebase and build system.
- Move common pipeline configuration to a new module (PR #280).
- Add scan synchronous simulation operator (PR #278).

CHAPTER 3

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.

3.1 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 one 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*.

3.1.1 Pip Binary Wheels

If you already have a newer Python3 (>= 3.6), then you can install pre-built TOAST packages from PyPI. You should always use virtualenv or similar tools to manage your python environments rather than pip-installing packages as root.

On Ubuntu Linux, you should install these minimal packages:

```
apt update
apt install python3 python3-pip python3-venv
```

On Redhat / Centos we need to take extra steps to install a recent python3:

```
yum update
yum install centos-release-scl
yum install rh-python36
scl enable rh-python36 bash
```

On MacOS, you can use homebrew or macports to install a recent python3. Now verify that your python is at least 3.6:

```
python3 --version
```

Next create a virtualenv (name it whatever you like):

```
python3 -m venv ${HOME}/cmb
```

Now activate this environment:

```
source ${HOME}/cmb/bin/activate
```

Within this virtualenv, update pip to the latest version. This is needed in order to install more recent wheels from PyPI:

```
python3 -m pip install --upgrade pip
```

Next, use pip to install toast and its requirements (note that the name of the package is "toast-cmb" on PyPI):

```
pip install toast-cmb
```

At this point you have toast installed and you can use it from serial scripts and notebooks. If you want to enable effective parallelism with toast (useful if you computer has many cores), then you need to install the mpi4py package. This package requires MPI compilers (usually MPICH or OpenMPI). Your system may already have some MPI compilers installed- try this:

```
which mpicc
mpicc -show
```

If the mpicc command is not found, you should use your OS package manager to install the development packages for MPICH or OpenMPI. Now you can install mpi4py:

```
pip install mpi4py
```

For more details about custom installation options for mpi4py, read the documentation for that package. You can test your TOAST installation by running the unit test suite:

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

And test running in parallel with:

```
mpirun -np 2 python -c 'import toast.tests; toast.tests.run()'
```

The runtime configuration of toast can also be checked with an included script:

```
toast_env_test.py
```

3.1.2 Conda Packages

If you already use the conda python stack, then you can install TOAST and all of its optional dependencies with the conda package manager. The conda-forge ecosystem allows us to create packages that are built consistently with all their dependencies. If you already have Anaconda / miniconda installed **and** it is a recent version, then skip ahead to "activating the root environment below".

If you are starting from scratch, 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:

```
conda install python=3 toast
```

Assuming this isthe 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.

If you want to use PySM with TOAST for sky simulations, you should install the `pysm3` and `libsharp` packages. For example:

```
conda install pysm3 libsharp
```

If you want to enable effective parallelism with toast, then you need to install the mpi4py package:

```
conda install mpi4py
```

As mentioned previously, you can test your TOAST installation by running the unit test suite:

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

And test running in parallel with:

```
mpirun -np 2 python -c 'import toast.tests; toast.tests.run()'
```

3.1.3 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".

3.2 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.

3.2.1 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 \
```

(continues on next page)

(continued from previous page)

```
    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 if you wish:

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

3.2.2 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)
```

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

```
pip install \
    numpy \
    scipy \
    matplotlib \
    healpy \
    astropy \
    pyephem \
    mpi4py
```

Then install libaatm as discussed in the previous section.

3.2.3 OS X with MacPorts

Todo: Document using macports to get gcc and installing optional dependencies.

3.2.4 OS X with Homebrew

Todo: Document installing compiled dependencies and using a virtualenv.

3.2.5 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.

3.2.6 Installing TOAST with CMake

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

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

AATM_ROOT The install prefix of the libaatm 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.

3.2.7 Installing TOAST with Pip / setup.py

The setup.py that comes with TOAST is just a wrapper around the cmake build system. You can pass options to the underlying cmake call by setting environment variables prefixed with "**TOAST_BUILD_**". For example, if you want to pass the location of the libaatm installation to cmake when using setup.py, you can set the "TOAST_BUILD_AATM_ROOT" environment variable. This will get translated to "-DAATM_ROOT" when cmake is invoked by setup.py

3.3 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()"
```

3.4 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 4

Data

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).

A TOAST workflow consists of one or more (distributed) observations representing the data, and a series of *operators* that "do stuff" with this data. The following sections detail the classes that represent TOAST data sets and how that data is distributed among many MPI processes.

4.1 Related Classes

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.

4.1.1 TOD Base Class

The base class is not used directly, but provides the interfaces required by all derived classes. The inputs to the TOD base 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 should 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 velocity.

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. This *cache* member is where alternate flavors of the timestream data are stored.

4.1.2 Cache Class

A *Cache* class behaves like a dictionary of N-dimensional numpy arrays. A restricted set of basic dtypes are supported. The memory for each buffer is allocated outside of Python in flat-packed and aligned storage. This means that it can be explicitly managed / freed, and can also be used directly in routines that require aligned memory for SIMD operations.

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)

4.1.3 Noise Model

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, running 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** (*str*) – 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)

4.1.4 Intervals

Within each TOD object, a process contains some local set of detectors and range of samples. That range of samples may contain one or more contiguous "chunks" that were used when distributing the data. Separate from this data distribution, TOAST has the concept of valid data "intervals". This list of intervals applies to the whole observation, and all processes have a copy of this list. This list of intervals is useful to define larger sections of data than what can be specified with per-sample flags. A single interval looks like this:

class toast.tod.**Interval**(*start=None*, *stop=None*, *first=None*, *last=None*)
Class storing a time and sample range.

Parameters

- **start** (*float*) – The start time of the interval in seconds.

- **stop** (*float*) – The stop time of the interval in seconds.
- **first** (*int*) – The first sample index of the interval.
- **last** (*int*) – The last sample index (inclusive) of the interval.

first
the first sample of the interval.

Type (int)

last
the first sample of the interval.

Type (int)

range
the number seconds in the interval.

Type (float)

samples
the number samples in the interval.

Type (int)

start
the start time of the interval.

Type (float)

stop
the start time of the interval.

Type (float)

4.1.5 The Data Class

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.

4.2 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*.

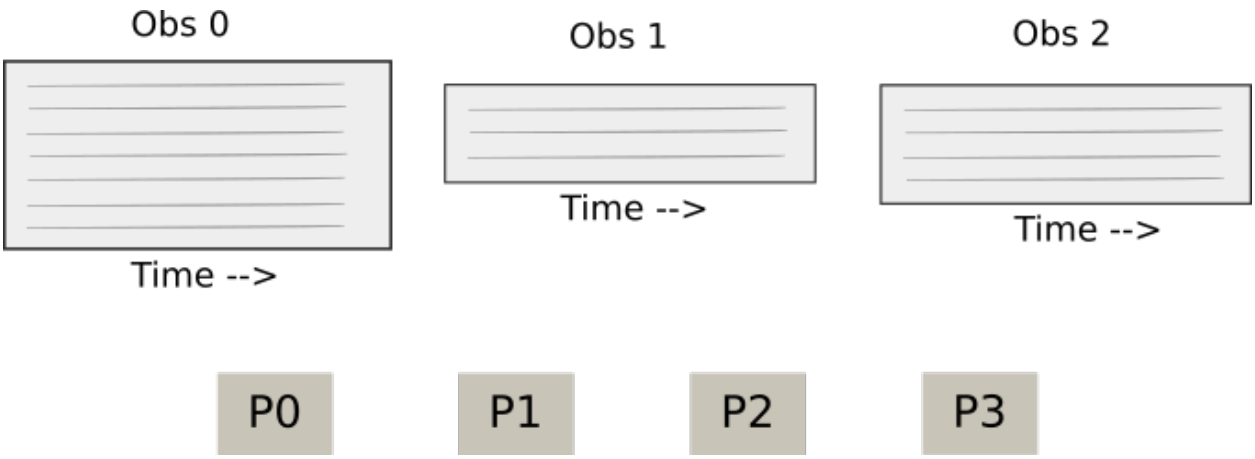
What about the data *within* an observation? A single observation is owned by exactly one of the process groups. The MPI communicator passed to the TOD constructor is the group communicator. Every process in the group will store some piece of the observation data. The division of data within an observation 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.

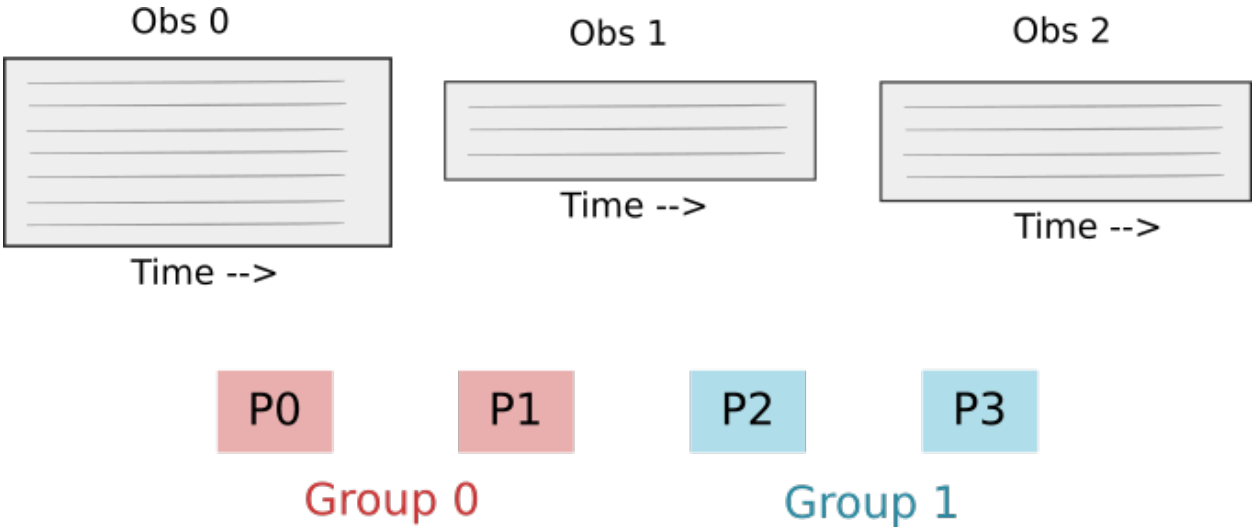
4.2.1 Examples

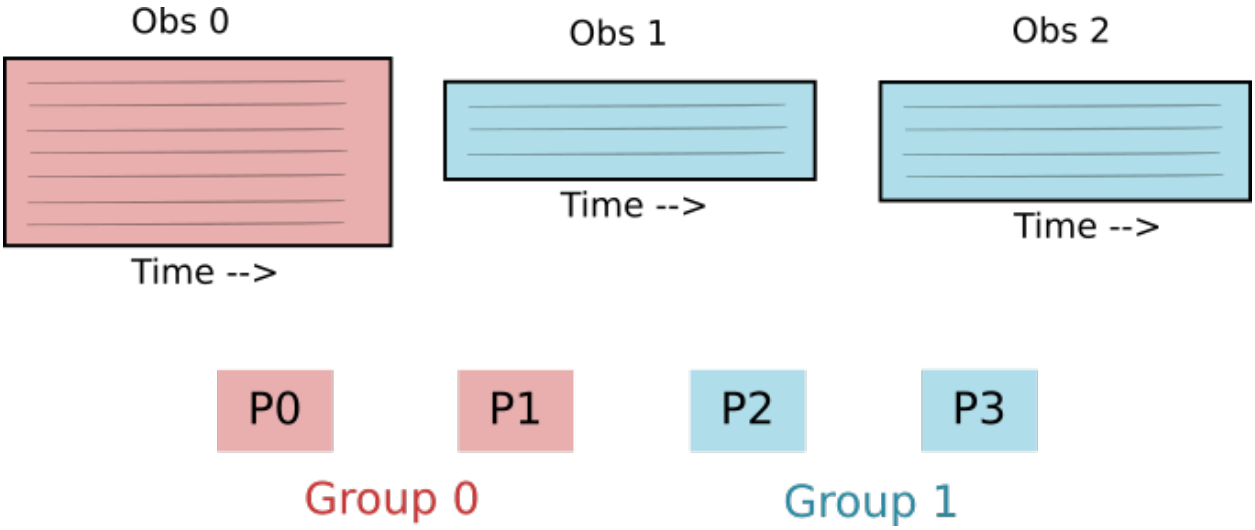
It is useful to walk through the process of how data is distributed for a simple case. We have some number of observations in our data, and we also have some number of MPI processes in our world communicator:



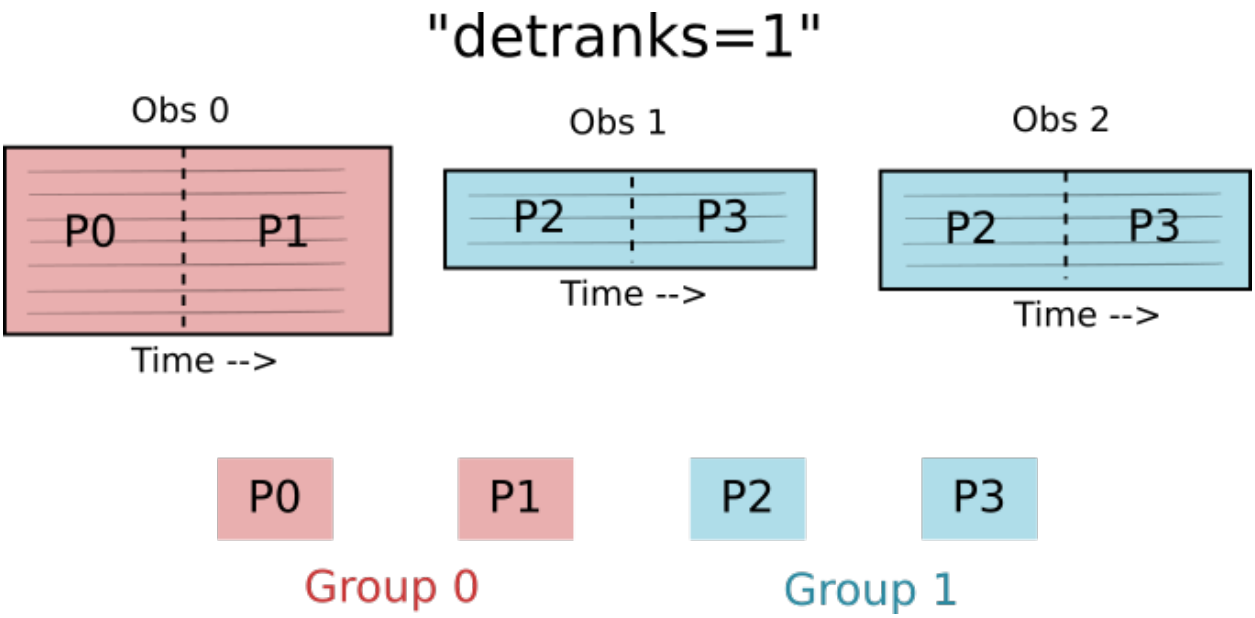
Starting point: Observations and MPI Processes.

Defining the process groups: We divide the total processes into equal-sized groups.

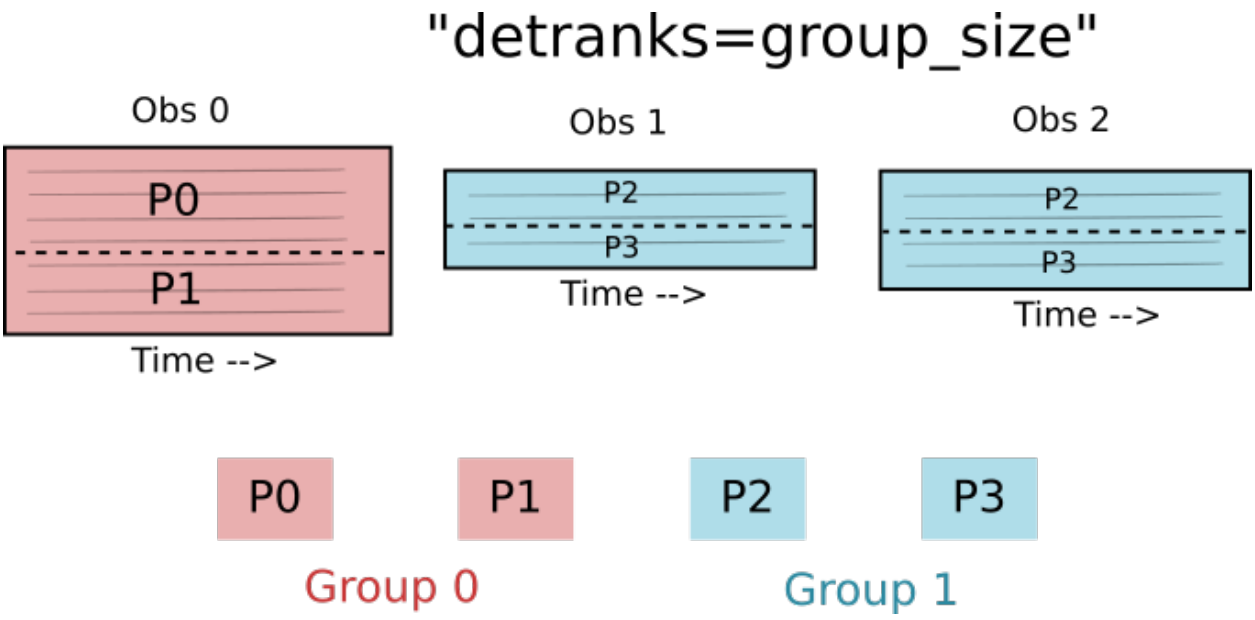

Obs 0
Time -->
Obs 1
Time -->
Obs 2
Time -->
P0
P1
P2
P3
Group 0
Group 1


Obs 0
Time -->
Obs 1
Time -->
Obs 2
Time -->
P0
P1
P2
P3
Group 0
Group 1

Assign observations to groups: Each observation is assigned to exactly one group. Each group has one or more observations.

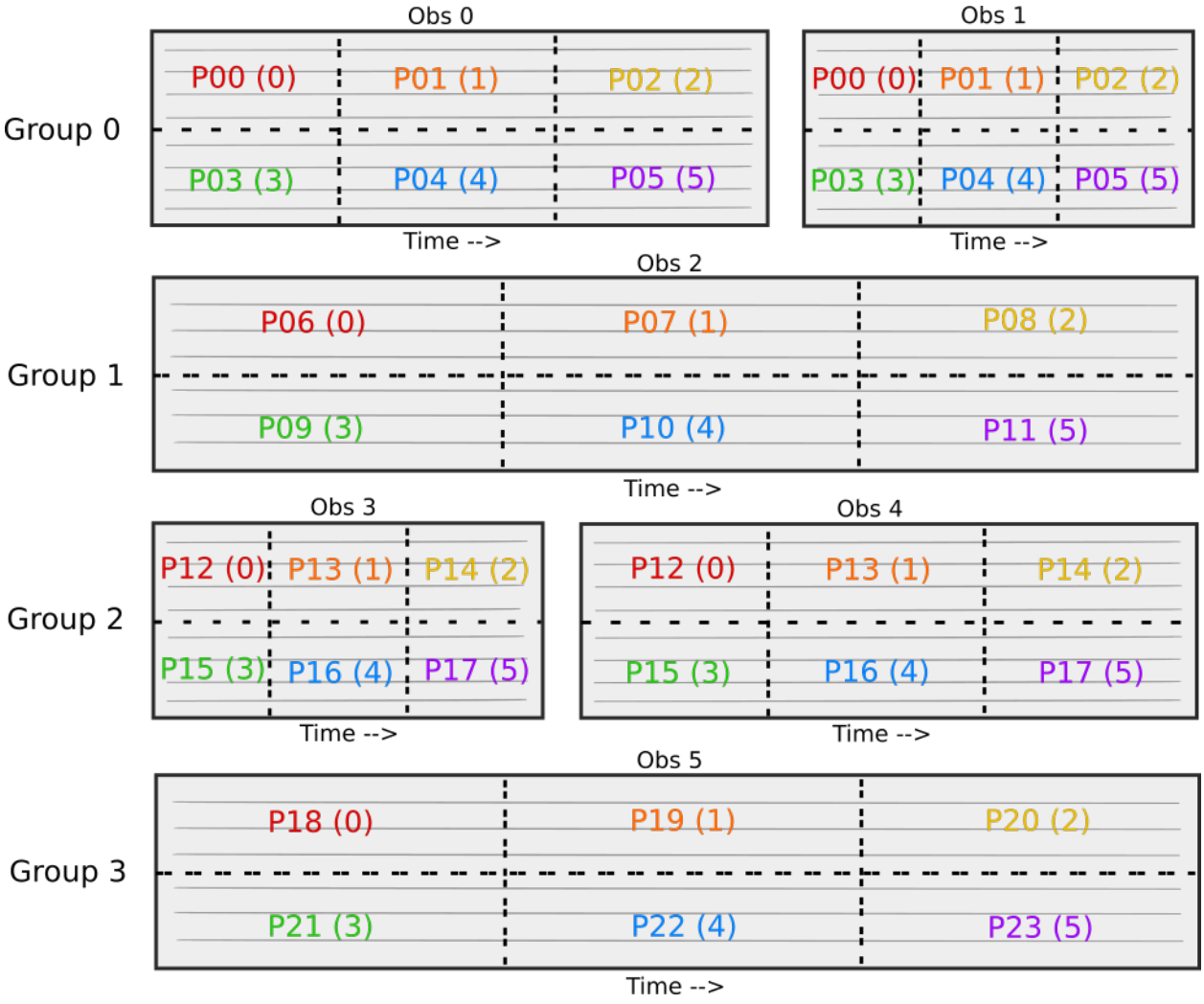


The *detranks* TOD constructor argument specifies how data **within** an observation is distributed among the processes in the group. The value sets the dimension of the process grid in the detector direction. In the above case, *detranks = 1*, so the process group is arranged in a one-dimensional grid in the time direction.



In the above case, the *detranks* parameter is set to the size of the group. This means that the process group is arranged in a one-dimensional grid in the process direction.

Now imagine a more complicated case (not currently used often if at all) where the process group is arranged in a two-dimensional grid. This is useful as a visualization exercise. Let's say that MPI.COMM_WORLD has 24 processes. We split this into 4 groups of 6 procesess. There are 6 observations of varying lengths and every group has one or 2 observations. For this case, we are going to use *detranks = 2*. 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:



4.3 Built-in Data Support

In most cases of "real" experiments / telescopes, there will be custom TOAST classes to represent the data. However, TOAST comes with support for some (mainly simulated) data types. These are useful when simulating proposed experiments that do not yet have detailed specifications. Once a project reaches the level of having detailed hardware specifications (either designed or measured), then it should implement custom classes for that instrument description and data I/O or simulation.

4.3.1 Simulated TOD Classes

Recall that a `TOD` class represents the properties of some detectors from a telescope for one observation. This includes its physical location / motion through space as well as geometric locations of the detectors and (in the case of real data) methods to read detector data. For simulated TOD classes, we simulate the telescope boresight pointing and detector properties. We do **not** simulate detector data as part of the TOD class- that is done by various simulation operators that accumulate signal and noise from various sources. For generic satellite simulations, we have:

For generic ground-based simulations we have:

4.3.2 Simulated Noise Properties

One common noise model that is frequency used in simulations is a 1/f spectrum with white noise.

class toast.tod.**AnalyticNoise**(*, *detectors*, *rate*, *fmin*, *fknee*, *alpha*, *NET*, *indices=None*)
Class representing an analytic noise model.

This generates an analytic PSD for a set of detectors, given input values for the knee frequency, NET, exponent, sample rate, minimum frequency, etc.

Parameters

- **detectors** (*list*) – List of detectors.
- **rate** (*dict*) – Dictionary of sample rates in Hertz.
- **fmin** (*dict*) – Dictionary of minimum frequencies for high pass
- **fknee** (*dict*) – Dictionary of knee frequencies.
- **alpha** (*dict*) – Dictionary of alpha exponents (positive, not negative!).
- **NET** (*dict*) – Dictionary of detector NETs.

NET(*det*)
(float): the NET.

alpha(*det*)
(float): the (positive!) slope exponent.

fknee(*det*)
(float): the knee frequency in Hz.

fmin(*det*)
(float): the minimum frequency in Hz, used as a high pass.

rate(*det*)
(float): the sample rate in Hz.

4.3.3 Simulated Intervals

Simulating regular data intervals is common task for many simulations:

toast.tod.**regular_intervals**(*n*, *start*, *first*, *rate*, *duration*, *gap*)
Function to generate simulated regular intervals.

This creates a list of intervals, given a start time/sample and time span for the interval and the gap in time between intervals. The length of the interval and the total interval + gap are rounded down to the nearest sample and all intervals in the list are created using those lengths.

If the time span is an exact multiple of the sampling, then the final sample is excluded. The reason we always round down to the whole number of samples that fits inside the time range is so that the requested time span boundary (one hour, one day, etc) will fall in between the last sample of one interval and the first sample of the next.

Example: you want to simulate science observations of length 22 hours and then have 4 hours of down time (e.g. a cooler cycle). Specifying a duration of 22*3600 and a gap of 4*3600 will result in a total time for the science + gap of a fraction of a sample less than 26 hours. So the requested 26 hour mark will fall between the last sample of one regular interval and the first sample of the next. Note that this fraction of a sample will accumulate if you have many, many intervals.

This function is intended only for simulations- in the case of real data, the timestamp of every sample is known and boundaries between changes in the experimental configuration are already specified.

Parameters

- **n** (*int*) – the number of intervals.
- **start** (*float*) – the start time in seconds.
- **first** (*int*) – the first sample index, which occurs at "start".
- **rate** (*float*) – the sample rate in Hz.
- **duration** (*float*) – the length of the interval in seconds.
- **gap** (*float*) – the length of the gap in seconds.

Returns a list of Interval objects.

Return type (list)

CHAPTER 5

Operators

TOAST workflows ("pipelines") 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 modifies it in place.

Parameters **None** –

exec(*data*)
Perform operations on a Data object.

Parameters **data** (*toast.Data*) – The distributed data.

Returns 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. TOAST ships with many built-in operators that are detailed in the rest of this section. Operator constructors frequently require many options. Most built-in operators have helper functions in the *toast.pipeline_tools* module to ease parsing of these options from the command line or argparse input file.

Todo: Document the pipeline_tools functions for each built-in operator.

5.1 Pointing Matrices

A "pointing matrix" in TOAST terms is the sparse matrix that describes how sky signal is projected to the timestream. In particular, the model we use is

$$d_t = \mathcal{A}_{tp} s_p + n_t$$

where we write s_p as a column vector having a number of rows given by the number of pixels in the sky. So the $\mathcal{A}_{tp}$ matrix has a number of rows given by the number of time samples and a column for every sky pixel. In practice, the

pointing matrix is sparse, and we only store the nonzero elements in each row. Also, our sky model often includes multiple terms (e.g. I, Q, and U). This is equivalent to having a set of values at each sky pixel. In TOAST we represent the pointing matrix as a vector of pixel indices (one for each sample) and a 2D array of "weights" whose values are the nonzero values of the matrix for each sample. TOAST includes a generic HEALPix operator to generate a pointing matrix.

5.1.1 Generic HEALPix Representation

Each experiment might create other specialized pointing matrices used in solving for instrument-specific signals.

5.2 Simulated Detector Signal

Every experiment is different, and a realistic simulation will likely require some customized code to model the instrument. However some kinds of simulations are generic, especially when doing trade studies in the early design phase of a project. TOAST comes with several built-in operators for simulating detector signals.

5.2.1 Sky Model

"Sky" In this case are sources of signals coming from outside of the Earth. The most simplistic model of the sky is just an input map at the same pixelization that is used for analysis (to avoid pixelization effects). There can be one map per detector or one map for a set of detectors (for example if simulating all detectors in a band without including bandpass variability):

The cosmological and orbital dipole can be simulated with this operator:

TOAST can also use some external packages for more complicated sky simulations. One of these is PySM, which supports generating bandpass-integrated sky maps in memory for each detector from a set of component maps. It can also do basic smoothing of the input signal:

For studies of far side lobes and full 4Pi beams, TOAST can use the external libconviqt package to convolve beam a_lm's with a sky a_lm at every sample:

5.2.2 Atmosphere Simulation

Although many ground experiments have traditionally modelled atmosphere as "correlated noise", in TOAST we take a different approach. For each observation, a realization of the physical atmosphere slab is created and moved according to the wind speed. Every detector sample is formed by a line of sight integral through this slab. This sort of real-space modelling naturally leads to correlated signal in the time domain that depends on atmospheric parameters and focalplane geometry.

5.2.3 Other Signals

Any systematic contaminant that is constant (for an observation) in the scanning reference frame (as opposed to sky coordinates), can be simulated as "Scan Synchronous Signal":

This next operator can be used to apply random gain errors to the timestreams:

class toast.tod.**OpGainScrambler**(*center=1*, *sigma=0.001*, *pattern='.*'*, *name=None*, *realization=0*, *component=234567*)

Apply random gain errors to detector data.

This operator draws random gain errors from a given distribution and applies them to the specified detectors.

Parameters

- **center** (*float*) – Gain distribution center.
- **sigma** (*float*) – Gain distribution width.
- **pattern** (*str*) – Regex pattern to match against detector names. Only detectors that match the pattern are scrambled.
- **name** (*str*) – Name of the output signal cache object will be <name_in>_<detector>. If the object exists, it is used as input. Otherwise signal is read using the tod read method.
- **realization** (*int*) – if simulating multiple realizations, the realization index.
- **component** (*int*) – the component index to use for this noise simulation.

exec(*data*)
Scramble the gains.

Parameters **data** (*toast.Data*) – The distributed data.

5.3 Simulated Detector Noise

TOAST includes an operator that simulates basic detector noise (1/f spectrum plus white noise), and also supports generating correlated noise by specifying a mixing matrix to combine individual detector streams with common mode sources.

class toast.tod.**OpSimNoise**(*out='noise'*, *realization=0*, *component=0*, *noise='noise'*, *rate=None*)
Operator which generates noise timestreams.

This passes through each observation and every process generates data for its assigned samples. The dictionary for each observation should include a unique 'ID' used in the random number generation. The observation dictionary can optionally include a 'global_offset' member that might be useful if you are splitting observations and want to enforce reproducibility of a given sample, even when using different-sized observations.

Parameters

- **out** (*str*) – accumulate data to the cache with name <out>_<detector>. If the named cache objects do not exist, then they are created.
- **realization** (*int*) – if simulating multiple realizations, the realization index.
- **component** (*int*) – the component index to use for this noise simulation.
- **noise** (*str*) – PSD key in the observation dictionary.

exec(*data*)
Generate noise timestreams.

This iterates over all observations and detectors and generates the noise timestreams based on the noise object for the current observation.

Parameters **data** (*toast.Data*) – The distributed data.

Raises

- KeyError – If an observation in data does not have noise object defined under given key.
- RuntimeError – If observations are not split into chunks.

simulate_chunk(**, tod, nse, curchunk, chunk_first, obsindx, times, telescope, global_offset*)
Simulate one chunk of noise for all detectors.

Parameters

- **tod** (toast.tod.TOD) – TOD object for the observation.
- **nse** (toast.tod.Noise) – Noise object for the observation.
- **curchunk** (*int*) – The local index of the chunk to simulate.
- **chunk_first** (*int*) – First global sample index of the chunk.
- **obsindx** (*int*) – Observation index for random number stream.
- **times** (*int*) – Timestamps for effective sample rate.
- **telescope** (*int*) – Telescope index for random number stream.
- **global_offset** (*int*) – Global offset for random number stream.

Returns Number of simulated samples

Return type chunk_samp (int)

5.4 Timestream Processing

Many timestream manipulations done prior to map-making are very specific to the instrument. However there are a few operations that are generically useful.

5.4.1 Filtering

This operator is used to build a template in azimuth bins of signal that is fixed in the scanning reference frame. This template is then subtracted from the timestream.

This next operator fits a polynomial to each scan and subtracts it.

class toast.tod.**OpPolyFilter**(*order=1*, *pattern='.*'*, *name=None*, *common_flag_name=None*, *common_flag_mask=255*, *flag_name=None*, *flag_mask=255*, *poly_flag_mask=1*, *intervals='intervals'*)

Operator which applies polynomial filtering to the TOD.

This applies polynomial filtering to the valid intervals of each TOD.

Parameters

- **order** (*int*) – Order of the filtering polynomial.
- **pattern** (*str*) – Regex pattern to match against detector names. Only detectors that match the pattern are filtered.
- **name** (*str*) – Name of the output signal cache object will be <name_in>_<detector>. If the object exists, it is used as input. Otherwise signal is read using the tod read method.
- **common_flag_name** (*str*) – Cache name of the output common flags. If it already exists, it is used. Otherwise flags are read from the tod object and stored in the cache under common_flag_name.
- **common_flag_mask** (*byte*) – Bitmask to use when flagging data based on the common flags.
- **flag_name** (*str*) – Cache name of the output detector flags will be <flag_name>_<detector>. If the object exists, it is used. Otherwise flags are read from the tod object.
- **flag_mask** (*byte*) – Bitmask to use when flagging data based on the detector flags.

- **poly_flag_mask** (*byte*) – Bitmask to use when adding flags based on polynomial filter failures.
- **intervals** (*str*) – Name of the valid intervals in observation.

exec(*data*)
Apply the polynomial filter to the signal.

Parameters **data** (*toast.Data*) – The distributed data.

5.4.2 Calibration

This operator applies a set of gains to the timestreams:

class toast.tod.**OpApplyGain**(*gain*, *name=None*)
Operator which applies gains to timelines.

Parameters

- **gain** (*dict*) – Dictionary, key "TIME" has the common timestamps, other keys are channel names their values are the gains
- **name** (*str*) – Name of the output signal cache object will be <name_in>_<detector>. If the object exists, it is used as input. Otherwise signal is read using the tod read method.

exec(*data*)
Apply the gains.

Parameters **data** (*toast.Data*) – The distributed data.

5.4.3 Utilities

These operators are used to manipulate cached data or perform other helper functions.

class toast.tod.**OpFlagGaps**(*common_flag_name=None*, *common_flag_value=1*, *intervals='intervals'*)
Operator which applies common flags to gaps between valid intervals.

Parameters

- **common_flag_name** (*str*) – the name of the cache object to use for the common flags. If None, use the TOD.
- **common_flag_value** (*int*) – the integer bit mask (0-255) that should be bitwise ORed with the existing flags.
- **intervals** (*str*) – Name of the valid intervals in observation.

exec(*data*)
Flag samples between valid intervals.

This iterates over all observations and flags samples which lie outside the list of intervals.

Parameters **data** (*toast.Data*) – The distributed data.

class toast.tod.**OpFlagsApply**(*name=None*, *common_flags=None*, *flags=None*, *common_flag_mask=1*, *flag_mask=1*)
This operator sets the flagged signal values to zero

exec(*data*)
Perform operations on a Data object.

Parameters **data** (*toast.Data*) – The distributed data.

Returns None

class toast.tod.**OpMemoryCounter**(**other_caching_objects*, *silent=False*)
Compute total memory used by TOD objects.

Operator which loops over the TOD objects and computes the total amount of memory allocated.

Parameters

- **silent** (*bool*) – Only count and return the memory without printing.
- ***other_caching_objects** – Additional objects that have a cache member and user wants to include in the total counts (e.q. DistPixels objects).

exec(*data*)
Count the memory

Parameters **data** (*toast.Data*) – The distributed data.

5.5 Map Making Tools

CMB map-making codes and algorithms have a long history going back more than 20 years. Early work focused on finding the maximum likelihood solution to the noise-weighted least squares expression for the pixelized map (and other parameters) given the data and an estimate of the time domain noise. Later efforts took other approaches, such as the destriping formalism used in Planck and agressive filtering and binning used in some ground experiments. In terms of algorithms built in to TOAST, there are 2 primary tools. The first is a wrapper around the external libmadam destriping map-maker. The second is a set of native mapmaking tools which are already quite usable for many common cases.

5.5.1 Libmadam Wrapper

This operator has some basic options and can also take a dictionary of parameters to pass to the underlying Fortran code for complete control over the options.

5.5.2 Native Tools

Todo: Document these more fully once the communication performance bottleneck is fixed.

CHAPTER 6

Pipelines

TOAST pipelines are a top-level python script which instantiates a *toast.Data* object as well as one or more *toast.Operator* classes which are applied to the data.

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

The details of how the global data object is created will depend on a particular project and likely use classes specific to that experiment. Here we look at several examples using built-in classes.

6.1 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

    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)
```

(continues on next page)

(continued from previous page)

```
    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
                # 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"
            )
```

(continues on next page)

(continued from previous page)

```
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 7

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.

7.1 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.

disable_function_timers(*self: toast._libtoast.Environment*) → None
Explicitly disable function timers.

enable_function_timers(*self: toast._libtoast.Environment*) → None
Explicitly enable function timers.

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.

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

7.2 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

verbose (*self: toast._libtoast.Logger*, *msg: str*) → None
Print a VERBOSE 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

7.3 Timing

TOAST includes some utilities for setting arbitrary timers on individual processes and also gathering timer data across MPI processes. A basic manual timer is implemented in the `Timer` class:

class toast.timing.**Timer**
Simple timer class.

This class is just a timer that you can start / stop / clear and report the results. It tracks the elapsed time and the number of times it was started.

calls (*self: toast._libtoast.Timer*) → int
Return the number of calls.

Returns The number of calls (if timer is stopped) else 0.

Return type (int)

clear (*self: toast._libtoast.Timer*) → None
Clear the timer.

elapsed_seconds (*self: toast._libtoast.Timer*) → float
Return the elapsed seconds from a running timer without modifying the timer state.

Returns The elapsed seconds (if timer is running).

Return type (float)

is_running (*self: toast._libtoast.Timer*) → bool
Is the timer running?

Returns True if the timer is running, else False.

Return type (bool)

report (*self: toast._libtoast.Timer*, *message: str*) → None
Report results of the timer to STDOUT.

Parameters **message** (*str*) – A message to prepend to the timing results.

Returns None

report_clear (*self: toast._libtoast.Timer*, *message: str*) → None
Report results of the timer to STDOUT and clear the timer.

If the timer was running, it is stopped before reporting and clearing and then restarted. If the timer was stopped, then it is left in the stopped state after reporting and clearing.

Parameters **message** (*str*) – A message to prepend to the timing results.

Returns None

report_elapsed(*self: toast._libtoast.Timer*, *message: str*) → None
Report results of a running timer to STDOUT without modifying the timer state.

Parameters **message** (*str*) – A message to prepend to the timing results.

Returns None

seconds(*self: toast._libtoast.Timer*) → float
Return the elapsed seconds.

Returns The elapsed seconds (if timer is stopped) else -1.

Return type (float)

start(*self: toast._libtoast.Timer*) → None
Start the timer.

stop(*self: toast._libtoast.Timer*) → None
Stop the timer.

A single `Timer` instance is only valid until it goes out of scope. To start and stop a global timer that is accessible anywhere in the code, use the `get()` method of the `GlobalTimers` singleton to get a handle to the single instance and then use the class methods to start and stop a named timer:

class toast.timing.**GlobalTimers**
Global timer registry.

This singleton class stores timers that can be started / stopped anywhere in the code to accumulate the total time for different operations.

clear_all(*self: toast._libtoast.GlobalTimers*) → None
Clear all global timers.

collect(*self: toast._libtoast.GlobalTimers*) → dict
Stop all timers and return the current state.

Returns A dictionary of Timers.

Return type (dict)

get() → toast._libtoast.GlobalTimers
Get a handle to the singleton class.

is_running(*self: toast._libtoast.GlobalTimers*, *name: str*) → bool
Is the specified timer running?

Parameters **name** (*str*) – The name of the global timer.

Returns True if the timer is running, else False.

Return type (bool)

names(*self: toast._libtoast.GlobalTimers*) → List[str]
Return the names of all currently registered timers.

Returns The names of the timers.

Return type (list)

report (*self: toast._libtoast.GlobalTimers*) → None
Report results of all global timers to STDOUT.

seconds (*self: toast._libtoast.GlobalTimers*, *name: str*) → float
Get the elapsed time for a timer.

The timer must be stopped.

Parameters **name** (*str*) – The name of the global timer.

Returns The elapsed time in seconds.

Return type (float)

start (*self: toast._libtoast.GlobalTimers*, *name: str*) → None
Start the specified timer.

If the named timer does not exist, it is first created before being started.

Parameters **name** (*str*) – The name of the global timer.

Returns None

stop (*self: toast._libtoast.GlobalTimers*, *name: str*) → None
Stop the specified timer.

The timer must already exist.

Parameters **name** (*str*) – The name of the global timer.

Returns None

stop_all (*self: toast._libtoast.GlobalTimers*) → None
Stop all global timers.

The `GlobalTimers` object can be used to automatically time all calls to a particular function by using the `@function_timer` decorator when defining the function.

To gather timing statistics across all processes you can use this function:

`toast.timing.`**gather_timers** (*comm=None*, *root=0*)
Gather global timer information from across a communicator.

Parameters

- **comm** (*MPI.Comm*) – The communicator or None.
- **root** (*int*) – The process returning the results.

Returns The timer stats on the root process, otherwise None.

Return type (dict)

And the resulting information can be written on the root process with:

`toast.timing.`**dump** (*results*, *path*)
Write out timing results to a format suitable for further processing.

Parameters

- **results** (*dict*) – The results as returned by compute_stats().
- **path** (*str*) – File root name to dump.

Returns None

7.4 Random Number Generation

The following functions define wrappers around an internally-built version of the Random123 package for streamed random number generation. This generator is fast and can return reproducible values from any location in the stream specified by the input key and counter values.

toast.rng.**random**(*samples*, *key=(0, 0)*, *counter=(0, 0)*, *sampler='gaussian'*, *threads=False*)
Generate random samples from a distribution for one stream.

This returns values from a single stream drawn from the specified distribution. The starting state is specified by the two key values and the two counter values. The second value of the "counter" is used to represent the sample index. If the serial option is enabled, only a single thread will be used. Otherwise the stream generation is divided equally between OpenMP threads.

Parameters

- **samples** (*int*) – The number of samples to return.
- **key** (*tuple*) – Two uint64 values which (along with the counter) define the starting state of the generator.
- **counter** (*tuple*) – Two uint64 values which (along with the key) define the starting state of the generator.
- **sampler** (*string*) – The distribution to sample from. Allowed values are "gaussian", "uniform_01", "uniform_m11", and "uniform_uint64".
- **threads** (*bool*) – If True, use OpenMP threads to generate the stream in parallel. NOTE: this may actually run slower for short streams and many threads.

Returns The random values of appropriate type for the sampler.

Return type (Aligned array)

If generating samples from multiple different streams you can use this function:

toast.rng.**random_multi**(*samples*, *keys*, *counters*, *sampler='gaussian'*)
Generate random samples from multiple streams.

Given multiple streams, each specified by a pair of key values and a pair of counter values, generate some number of samples from each. The number of samples is specified independently for each stream. The generation of the streams is run with multiple threads.

NOTE: if you just want threaded generation of a single stream, use the "threads" option to the random() function.

Parameters

- **samples** (*list*) – The number of samples to return for each stream
- **keys** (*list*) – A tuple of integer values for each stream, which (along with the counter) define the starting state of the generator for that stream.
- **counters** (*list*) – A tuple of integer values for each stream, which (along with the key) define the starting state of the generator for that stream.
- **sampler** (*string*) – The distribution to sample from. Allowed values are "gaussian", "uniform_01", "uniform_m11", and "uniform_uint64".

Returns The random samples for each stream.

Return type (list)

7.5 Vector Math Operations

The following functions are just simple wrappers vendor math libraries or the default libm. They exist mainly just to provide a common interface to architecture-specific math libraries.

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

toast.utils.**vcos**(*in: buffer*, *out: buffer*) → None
Compute the Cosine 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

toast.utils.**vsincos**(*in: buffer*, *sinout: buffer*, *cosout: buffer*) → None
Compute the sine and cosine for an array of float64 values.

The results are stored in the output buffers. 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.
- **sinout** (*array_like*) – 1D array of float64 values.
- **cosout** (*array_like*) – 1D array of float64 values.

Returns None

toast.utils.**vatan2**(*y: buffer*, *x: buffer*, *ang: buffer*) → None
Compute the arctangent of the y and x 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

- **y** (*array_like*) – 1D array of float64 values.
- **x** (*array_like*) – 1D array of float64 values.
- **ang** (*array_like*) – output angles as 1D array of float64 values.

Returns None

toast.utils.**vsqrt**(*in: buffer*, *out: buffer*) → None
Compute the sqrt 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

toast.utils.**vrsqrt**(*in: buffer*, *out: buffer*) → None
Compute the inverse sqrt 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

toast.utils.**vexp**(*in: buffer*, *out: buffer*) → None
Compute e^x 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

toast.utils.**vlog**(*in: buffer*, *out: buffer*) → None
Compute the natural log of 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

CHAPTER 8

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.

8.1 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.

8.2 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.

8.3 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}"
}
```

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

```
load_toast
```

CHAPTER 9

Tutorial

The TOAST tutorial is based on a series of notebooks that were presented in past workshops. You can find them inside the `tutorial` directory in the top level of the source tree. These notebooks are designed to be run interactively from a laptop or workstation. To avoid adding binary objects to the git repository, these notebooks do not have any outputs saved. You can either run them to view the output or you can browse a copy of the generated output here.

CHAPTER 10

Developer Guidelines

Everyone contributing code to TOAST should aim to follow these guidelines in order to keep the code consistent. Whenever you make changes and before opening a pull request, run the script `src/format_source.sh` to apply the standard formatting rules to the Python and C++ code. If you use an editor with some other automatic code formatting, you should disable it unless you can configure it identically to the action of this script.

10.1 Python Code

We aim to follow PEP8 style guidelines whenever possible. There are some reasonable exceptions to this. In particular, import statements might not always be placed at the top of the code if:

- The import is an optional feature that has large performance impacts and which is only used infrequently. For example matplotlib, astropy, etc.
- Some initialization code is needed prior to the import statement. For example, setting the matplotlib backend.

Other style choices:

- Double quotes for strings unless the string contains double quotes, resulting in excessive backslash escaping. This should be handled automatically by the code formatter.

We use the "black" command line tool to format our source. This needs to be installed on your system before running the `format_source.sh` script.

10.2 Compiled Code

For consistency with python, class names follow python CamelCase convention. Function names follow python_underscore_convention. Formatting is set by uncrustify with a custom config file and this is run by the `format_source.sh` script.

All code that is exposed through pybind11 is in a single toast namespace. Nested namespaces may be used for code that is internal to the C++ code.

The "using" statement is allowed for aliasing a specific class or type:

```
using ShapeContainer = py::detail::any_container<ssize_t>;
```

But should **not** be used to import an entire namespace:

```
using std;
```

Header files included with "#include" should use angle brackets ("<>") for the header file name. If this fails for some reason then that indicates a problem with the build system and its header file search paths. Using double quotes for "#include" statements is OK when including raw source files in the same directory (for example, when including raw *.cpp contents generated by external scripts).

When including C standard header files in C++, use the form:

```
#include <cstdio>
```

Rather than:

```
#include <stdio.h>
```

Pointer / reference declarations: this allows reading from right to left as "a pointer to a constant double" or "a reference to a constant double":

```
double const * data
double const & data
```

Not:

```
const double * data
const double & data
```

When indexing the size of an STL container, the index variable should be either of the size type declared in the container class or size_t.

When describing time domain sample indices or intervals, we using int64_t everywhere for consistency. This allows passing, e.g. "-1" to communicate unspecified intervals or sample indices.

Single line conditional statements:

```
if (x > 0) y = x;
```

Are permitted if they fit onto a single line. Otherwise, insert braces.

Internal toast source files should not include the main "toast.hpp". Instead they should include the specific headers they need. For example:

```
#include <toast/sys_utils.hpp>
#include <toast/math_lapack.hpp>
#include <toast/math_qarray.hpp>
```

If attempting to vectorize code with OpenMP simd constructs, be sure to check that any data array used in the simd region are aligned (see toast::is_aligned). Otherwise this can result in silent data corruption.

Documentation: sphinx is used. All python code should have docstrings. All C++ code exposed through pybind11 should also have docstrings defined in the bindings. C++ code that is not exposed to python is considered internal, expert-level code that does not require formal documentation. However, such code should have sufficient comments to describe the algorithm and design choices.

10.3 Testing Workflow

We are using a github workflow to pull docker containers with our dependencies and run our unit tests. Those docker containers are re-generated whenever a new tag is made on the cmbenv git repository. So if there are dependencies that need to be updated, open a PR against cmbenv which updates the version or build in the package file. After merging and tagging a new cmbenv release the updated docker images will be available in an hour or two and be used automatically.

10.4 Release Process

There are some github workflows that only run when a new tag is created. Unless you are sure everything works, create a "release candidate" tag first. Before making a tag, ensure that the *docs/changes.rst* file contains all pull requests that have been merged since the last tag. Also edit the *src/toast/RELEASE* file and set the version to a PEP-440 compatible string. Next go onto the github releases page and create a new release from the master branch. Briefly, the format for a stable release, a release candidate or alpha version is:

2.6.9 2.6.7rc2 2.6.8a1

After tagging the release, verify that the github workflow to deploy pip wheels runs and uploads these to PyPI. The conda-forge bots will automatically detect the new tag and open a PR to update the feedstock. Also, after the release is complete, update the *RELEASE* file to be at the "alpha" of the next release. For example, after tagging version 3.4.5, set the version in the RELEASE file to 3.4.6a1. This version is **only** used when building pip wheels. Anyone installing locally with setup.py or with pip running on the local source tree will get a version constructed from the number of commits since the last git tag.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Index

H

I

K

L

M

N

O

P

R

S

T

V

W