TOAST Documentation

Release 2.2.0.dev50

Theodore Kisner, Reijo Keskitalo

Feb 11, 2019

Contents

Contents:

CHAPTER 1

Introduction

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 and 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 2.0 evolved as a re-implementation (in Python) of an earlier codebase written in C++. This was a pragmatic choice given the need to interface better with instrument scientists, and was made possible by improving support for Python on HPC systems.

TOAST is a Python package with tools for:

- Distributing data among many processes
- Performing operations on the local pieces of the data
- Creating customized processing operations

This package comes with a set of basic operations that will expand as development continues. All of the experiment-specific classes and pipeline scripts are kept in separate git repositories. Currently repositories exist for:

- Planck
- LiteBIRD
- CMB "Stage 4"

This list will grow over time.

1.1 Data Organization and Terminology

Example: Satellite

Example: Ground-Based

Example: Balloon

CHAPTER 2

Installation

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

2.1 Compiled Dependencies

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

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

2.2 Python Dependencies

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

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

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

There are obviously several ways to meet these python requirements.

2.2.1 Option #0

If you are using machines at NERSC, see *Using at NERSC*.

2.2.2 Option #1

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

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

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

2.2.3 Option #2

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

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

2.2.4 Option #3

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

```
%> conda install numpy scipy matplotlib mpi4py
```

Then install healpy and pyephem with pip:

```
%> pip install healpy pyephem
```

2.3 Using Configure

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

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

Now run configure:

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

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

```
%> make install
```

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

2.4 Testing the Installation

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

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

CHAPTER 3

Pipelines

Before delving into the structure of the toast package, it is sometimes useful to look at (and use!) an example. One such program is the simple script below which simulates a fake satellite scanning strategy with a focalplane of detectors and then makes a map.

3.1 Simple Satellite Simulation

The current version of this tool simulates parameterized boresight pointing and then uses the given focalplane (loaded from a pickle file) to compute the detector pointing. Noise properties of each detector are used to simulate noise timestreams.

In order to create a focalplane file, you can do for example:

```
import pickle
import numpy as np

fake = {}
fake['quat'] = np.array([0.0, 0.0, 1.0, 0.0])
fake['fwhm'] = 30.0
fake['fknee'] = 0.05
fake['alpha'] = 1.0
fake['NET'] = 0.000060
fake['color'] = 'r'
fp = {}
fp['bore'] = fake

with open('fp_lb.pkl', 'wb') as p:
    pickle.dump(fp, p)
```

Note that until the older TOAST mapmaking tools are ported, this script requires the use of libmadam (the –madam option).

```
usage: toast_satellite_sim.py [-h] [--samplerate SAMPLERATE]
                              [--spinperiod SPINPERIOD]
                              [--spinangle SPINANGLE]
                              [--precperiod PRECPERIOD]
                              [--precangle PRECANGLE] [--hwprpm HWPRPM]
                              [--hwpstep HWPSTEP] [--hwpsteptime HWPSTEPTIME]
                              [--obs OBS] [--gap GAP] [--numobs NUMOBS]
                              [--obschunks OBSCHUNKS] [--outdir OUTDIR]
                              [--debug] [--nside NSIDE] [--baseline BASELINE]
                              [--noisefilter] [--madam] [--fp FP]

Simulate satellite boresight pointing and make a noise map.

optional arguments:
  -h, --help            show this help message and exit
  --samplerate SAMPLERATE
                        Detector sample rate (Hz)
  --spinperiod SPINPERIOD
                        The period (in minutes) of the rotation about the spin
                        axis
  --spinangle SPINANGLE
                        The opening angle (in degrees) of the boresight from
                        the spin axis
  --precperiod PRECPERIOD
                        The period (in minutes) of the rotation about the
                        precession axis
  --precangle PRECANGLE
                        The opening angle (in degrees) of the spin axis from
                        the precession axis
  --hwprpm HWPRPM       The rate (in RPM) of the HWP rotation
  --hwpstep HWPSTEP     For stepped HWP, the angle in degrees of each step
  --hwpsteptime HWPSTEPTIME
                        For stepped HWP, the the time in seconds between steps
  --obs OBS             Number of hours in one science observation
  --gap GAP             Cooler cycle time in hours between science obs
  --numobs NUMOBS       Number of complete observations
  --obschunks OBSCHUNKS
                        Number of chunks to subdivide each observation into
                        for data distribution
  --outdir OUTDIR       Output directory
  --debug               Write diagnostics
  --nside NSIDE         Healpix NSIDE
  --baseline BASELINE   Destriping baseline length (seconds)
  --noisefilter         Destripe with the noise filter enabled
  --madam               If specified, use libmadam for map-making
  --fp FP               Pickle file containing a dictionary of detector
                        properties. The keys of this dict are the detector
                        names, and each value is also a dictionary with keys
                        "quat" (4 element ndarray), "fwhm" (float, arcmin),
                        "fknee" (float, Hz), "alpha" (float), and "NET"
                        (float). For optional plotting, the key "color" can
                        specify a valid matplotlib color string.
```

3.2 Example: Proposed CoRE Satellite Boresight

Here is one example using this script to generate one day of scanning with a single boresight detector, and using one proposed scan strategy for a LiteCoRE satellite:

```
toast_satellite_sim.py --samplerate 175.86 --spinperiod 1.0 --spinangle 45.0
--precperiod 5760.0 --precangle 50.0 --hwprpm 0.0 --obs 23.0 --gap 1.0
--obschunks 24 --numobs 1 --nside 1024 --baseline 5.0 --madam --noisefilter
--fp fp_core.pkl --outdir out_core_nohwp_fast
```

3.3 Example: Proposed LiteBIRD Satellite Boresight

Here is how you could do a similar thing with a boresight detector and one proposed lightbird scanning strategy for a day:

```
toast_satellite_sim.py --samplerate 23.0 --spinperiod 10.0 --spinangle 30.0
--precperiod 93.0 --precangle 65.0 --hwprpm 88.0 --obs 23.0 --gap 1.0
--obschunks 24 --numobs 1 --nside 1024 --baseline 60.0 --madam --fp fp_lb.pkl
--debug --outdir out_lb_hwp
```

3.4 Creating Your Own Pipeline

TOAST is designed to give you tools to piece together your own data processing workflow. Here is a slightly modified version of the pipeline script above. This takes a boresight detector with 1/f noise properties, simulates a sort-of Planck scanning strategy, generates a noise timestream, and then generates a fake signal timestream and adds it to the noise. Then it uses madam to make a map.

```
#!/usr/bin/env python3

import mpi4py.MPI as MPI

import os
import re

import numpy as np

import quaternionarray as qa

import toast
import toast.tod as tt
import toast.map as tm


# scanning parameters

samplerate = 25.0 # Hz
spinperiod = 1.0 # minutes
spinangle = 85.0 # degrees
precperiod = 0 # minutes
precangle = 0 # degrees
```

(continues on next page)

(continued from previous page)

```
# we add a HWP here, since we have one detector for just a couple days

hwprpm = 60.0

# map making

nside = 256
baseline = 60.0

# make a fake focalplane

fake = {}
fake['quat'] = np.array([0.0, 0.0, 1.0, 0.0])
fake['fwhm'] = 30.0
fake['fknee'] = 0.1
fake['alpha'] = 1.5
fake['NET'] = 0.0002
fp = {}
fp['bore'] = fake

# Since madam only supports a single observation, we use
# that here so that we can use the same data distribution whether
# or not we are using libmadam.  Normally we would have multiple
# observations with some subset assigned to each process group.

# This is the 2-level toast communicator.  By default,
# there is just one group which spans MPI_COMM_WORLD.

comm = toast.Comm()

# construct the list of intervals.  We'll use 55 minute intervals
# with a 5 minute gap. and observe for 4 days.

numobs = 4 * 24
science = 55 * 60
gap = 5 * 60

intervals = tt.regular_intervals(numobs, 0.0, 0, samplerate, science, gap)

# how many samples in one interval (plus the gap)?

interval_samples = intervals[1].first - intervals[0].first

# when we distribute the data, we don't want to split these intervals
# between processes.  So make a list of the samples in each interval
# and pass that list when constructing the TOD class.  This way it
# distributes the data as evenly as possible among the processes.

distsizes = []
for it in intervals:
    distsizes.append(interval_samples)

# how many total samples do we have now?

totsamples = np.sum(distsizes)

# create the single TOD for this observation
```

(continues on next page)

(continued from previous page)

```
detectors = sorted(fp.keys())
detquats = {}
for d in detectors:
    detquats[d] = fp[d]['quat']

tod = tt.TODSatellite(
    comm.comm_group,
    detquats,
    totsamples,
    firsttime=0.0,
    rate=samplerate,
    spinperiod=spinperiod,
    spinangle=spinangle,
    precperiod=precperiod,
    precangle=precangle,
    sampsizes=distsizes
)

# Create the noise model for this observation

fmin = 2.0 / samplerate
fknee = {}
alpha = {}
NET = {}
for d in detectors:
    fknee[d] = fp[d]['fknee']
    alpha[d] = fp[d]['alpha']
    NET[d] = fp[d]['NET']

noise = tt.AnalyticNoise(rate=samplerate, fmin=fmin, detectors=detectors, fknee=fknee,
↪ alpha=alpha, NET=NET)

# The distributed timestream data

data = toast.Data(comm)

# Create the (single) observation

ob = {}
ob['name'] = 'mission'
ob['tod'] = tod
ob['intervals'] = intervals
ob['baselines'] = None
ob['noise'] = noise
ob['id'] = 0

data.obs.append(ob)

# Constantly slewing precession axis, so that it makes a circle in one year

degday = 360.0 / 365.25

precquat = tt.slew_precession_axis(nsim=tod.local_samples[1],
    firstsamp=tod.local_samples[0], samplerate=samplerate, degday=degday)

# we set the precession axis now, which will trigger calculation
```

(continues on next page)

(continued from previous page)

```
# of the boresight pointing.

tod.set_prec_axis(qprec=precquat)

# simulate noise

nse = tt.OpSimNoise(out='simdata')
nse.exec(data)

# make a Healpix pointing matrix.

pointing = tt.OpPointingHpix(nside=nside, nest=True, mode='IQU', hwprpm=hwprpm)
pointing.exec(data)

# Simulate a gradient signal and accumulate it to the same output
# as the noise simulation.

grad = tt.OpSimGradient(out='simdata', nside=nside, min=-1.0, max=1.0, nest=True)
grad.exec(data)

# Mapmaking.  For purposes of this simulation, we use detector noise
# weights based on the NET (white noise level).  If the destriping
# baseline is too long, this will not be the best choice.

detweights = {}
for d in detectors:
    net = fp[d]['NET']
    detweights[d] = 1.0 / (samplerate * net * net)

# Set up MADAM map making.  By setting purge=True, we will
# purge all data after copying it into the madam
# buffers.  This is ok, as long as madam is the last step of
# the pipeline.

outdir = 'out_example_customize'
if not os.path.isdir(outdir):
    os.mkdir(outdir)

pars = {}

cross = int(nside / 2)
submap = int(nside / 8)

pars[ 'temperature_only' ] = 'F'
pars[ 'force_pol' ] = 'T'
pars[ 'kfirst' ] = 'T'
pars[ 'base_first' ] = baseline
pars[ 'nside_map' ] = nside
pars[ 'nside_cross' ] = cross
pars[ 'nside_submap' ] = submap
pars[ 'write_map' ] = 'T'
pars[ 'write_binmap' ] = 'T'
pars[ 'write_matrix' ] = 'T'
pars[ 'write_wcov' ] = 'T'
pars[ 'write_hits' ] = 'T'
pars[ 'kfilter' ] = 'F'
pars[ 'run_submap_test' ] = 'T'
```

(continues on next page)

(continued from previous page)

```
pars[ 'fsample' ] = samplerate
pars[ 'path_output' ] = outdir

madam = tm.OpMadam(params=pars, detweights=detweights, name='simdata', purge=True)
madam.exec(data)
```

CHAPTER 4

Data Distribution

The toast package is designed for data that is distributed across many processes. When passing the data to toast processing routines, you can either use pre-defined base classes as a container and copy your data into them, or you can create your own derived classes that provide a standard interface.

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

The Data class below is essentially just a list of observations for each process group.

4.1 Example

```
#!/usr/bin/env python

import mpi4py.MPI as MPI

import toast
import toast.tod as tt

# Split COMM_WORLD into groups of 4 processes each
cm = toast.Comm(world=MPI.COMM_WORLD, groupsize=4)

# Create the distributed data object
dd = toast.Data(comm=cm)

# Each process group appends some observations.
# For this example, each observation is going to have the same
# number of samples, and the same list of detectors.  We just
# use the base TOD class, which contains the data in memory.

obs_samples = 100
```

(continues on next page)

(continued from previous page)

```
obs_dets = ['detA', 'detB', 'detC']

for i in range(10):
    tod = tt.TOD(cm.comm_group, obs_dets, obs_samples)
    indx = cm.group * 10 + i
    ob = {}
    ob['name'] = '{}'.format(indx)
    ob['tod'] = tod
    ob['intervals'] = None
    ob['noise'] = None
    ob['id'] = indx
    dd.obs.append(ob)

# Now at the end we have 4 process groups, each of which is assigned
# 10 observations.  Each of these observations has 3 detectors and 100
# samples.  So the Data object contains a total of 40 observations and
# 12000 samples.
```

CHAPTER 5

Telescope TOD

The TOD base class represents the timestream information associated with a telescope. The base class enforces a minimal set of methods for reading and writing detector data and flags, detector pointing, and timestamps. The base class also provide methods for returning information about the data distribution, including which samples are local to a given process.

The base TOD class contains a member which is an instance of a Cache object. This is similar to a dictionary of arrays, but by default the memory used in these arrays is allocated in C, rather than using the python memory pool. This allows us to do aligned memory allocation and explicitly manage the lifetime of the memory.

CHAPTER 6

Data 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 TOD sample range, and all processes have a copy of this list.

CHAPTER 7

Noise Model

The Noise base class represents the time domain noise covariance for all detectors for an entire TOD length.

CHAPTER 8

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. PyTOAST includes a generic HEALPix operator to generate a pointing matrix.

8.1 Generic HEALPix Representation

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

CHAPTER 9

Simulations

There are several classes included in pytoast that can simulate different types of data.

9.1 Simulated Telescope

9.2 Simulated Noise Model

9.3 Simulated Intervals

9.4 Simulated Detector Data

This operator uses an externally installed libconviqt.

CHAPTER 10

Map-making Tools

This broad class of operations include anything that generates pixel-space data products.

10.1 Distributed Pixel-space Data

10.2 Diagonal Noise Covariance

10.3 Native Mapmaking

Using the distributed diagonal noise covariance tools, one can make a simple binned map. Porting the old TOAST map-maker to this version of TOAST is still on the to-do list.

10.4 External Madam Interface

If the MADAM library is installed and in your shared library search path, then you can use it to make maps.

CHAPTER 11

Using at NERSC

To use TOAST at NERSC, you need to have a Python3 software stack with all dependencies installed. There is already such a software stack installed on edison and cori.

11.1 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/${NERSC_HOST}/contrib/hpcosmo/modulefiles
```

You can safely put the above line in your ~/.bashrc.ext inside the sections for edison and cori.

11.2 Load Dependencies

In order to load a full python-3.5 stack, and also all dependencies needed by toast, do:

```
%> module load toast-deps
```

11.3 Install TOAST

If you are using a stable release of TOAST, there may already be an installation you can use. Do:

```
%> module avail toast
```

to see if something is already built meets your needs. If not, then you will have to install TOAST yourself. When installing *any* software at NERSC, we need to keep several things in mind:

- The home directories are small.
- The performance of the home directories when accessed by many processes is very bad.

- The scratch space has lots of room, and very good performance.
- Any file on the scratch space that has not be accessed for some number of weeks will be deleted.

So unfortunately there is no location which has good performance and also persistent file storage. For this example, we will install software to scratch and assume that we will be using the software frequently enough that it will never be purged. If you have not used the tools for a month or so, you should probably reinstall just to be sure that everything is in place.

First, we pick a location to install our software. For this example, we will be installing to a "software" directory in our scratch space. First make sure that exists:

```
%> mkdir -p ${SCRATCH}/software
```

Now we will create a small shell function that loads this location into our search paths for executables and python packages. Add this function to ~/.bashrc.ext and you can rename it to whatever you like:

```
toast () {
    pref=${SCRATCH}/software/toast
    export PATH=${pref}/bin:${PATH}
    export PYTHONPATH=${pref}/lib/python3.5/site-packages:${PYTHONPATH}
}
```

Log out and back in to make this function visible to your shell environment. Now checkout the toast source in your home directory somewhere:

```
%> cd
%> git clone https://github.com/hpc4cmb/toast.git
```

Then configure and build the software. Unless you know what you are doing, you should probably use the platform config example for the machine you are building for:

```
%> cd toast
%> ./autogen.sh
%> ./platforms/edison_gnu_mkl.sh --prefix=${SCRATCH}/software/toast
```

Now we can run our function to load this installation into our environment:

```
%> toast
```

On NERSC systems, MPI is not allowed to be run on the login nodes. In order to run our unittests, we first get an interactive compute node:

```
%> salloc
```

and then run the tests:

```
%> srun python -c "import toast; toast.test()"
```

You should read through the many good NERSC webpages that describe how to use the different machines. There are pages for edison and pages for cori.

11.4 Install Experiment Packages

If you are a member of Planck, Core, or LiteBIRD, you can get access to separate git repos with experiment-specific scripts and tools. You can install these to the same location as toast. All of those packages currently use distutils, and you will need to do the installation from a compute node (since importing the toast python module will load MPI):

```
%> cd toast-<experiment>
%> salloc
%> srun python setup.py clean
%> srun python setup.py install --prefix=${SCRATCH}/software/toast
```

CHAPTER 12

Developer's Guide

TOAST aims to follow best practices whenever reasonably possible. If you submit a pull request to contribute C++ code, try to match the existing coding style (indents are 4 spaces, not tabs, curly brace placement, spacing, etc). If you are contributing python code follow PEP-8. When documenting python classes and methods, we use google-style docstrings. The C++ code in TOAST uses the google test framework for unit tests. Python code uses the standard built in unittest classes. When contributing new code, please add unit tests as well. Even if we don't have perfect test coverage, that should be our goal. When actively developing the codebase, you can run the C++ unit tests without installation by doing:

```
%> make check
```

In order to run the python unit tests, you must first do a "make install".

CHAPTER 13

Indices and tables

- genindex
- modindex
- search