Commander Documentation

Release 0.1

D. S. Seljebotn

November 01, 2013

CONTENTS

The main purpose of Commander is to be the best CMB Gibbs sampler around. However, doing that depends on a lot of building blocks, and it is also an aim to package those building blocks nicely enough so that Commander (as a library) is applicable to other settings as well.

Contents:

CHAPTER

ONE

THE COMMANDER USER'S GUIDE

1.1 How to compute constrained realizations

To set up a Commander run, use a Python script such as this one: https://gist.github.com/4065322

Some explanation. First, the usual imports:

```
from __future__ import division
import numpy as np
import os
import commander as cm
```

Then, set some Python variables. These are simply reused below and have *no* meaning by themselves, so we'll cover them when they are used:

```
output_filename = 'newmonodip-em7-%d.h5' % os.getpid()
lmax = 500
lprecond = 50
eps = 1e-7
cache_path = 'cache'
seed = None # integer, or None for getting random seed from OS
```

First we must define our source of data, by constructing *cm.SkyObservation* objects. Constructing these objects does nothing but basically keep track of a bunch of filenames, in particular, *no data is loaded at this stage*. To keep things brief we use a Python loop but nothing stops you from unrolling the loop yourself to get Commander 1-style configuration. Again, simply constructing these objects have no value by itself – it's how they are used later that matters.

Note that columns in FITS tables are referenced by a tuple `(fits_filename, extension_no, column)`, where *column* can be either an integer or a name. Also note that while in this case we opt for having Commander do the necesarry data processing to get an RMS map you can of course provide an RMS map yourself by passing `TT_rms_map`.

Environment variables ("$DATA") and home directories ("~dagss") are always expanded in any filenames.

```
observations = []
for name, da_list in [('K', [('K1', '1.437 mK')]),
                      ('Ka', [('Ka1', '1.470 mK')]),
                      ('Q', [('Q1', '2.254 mK'), ('Q2', '2.140 mK')]),
                      ('V', [('V1', '3.319 mK'), ('V2', '2.955 mK')]),
                      ('W', [('W1', '5.906 mK'), ('W2', '6.572 mK'),
                             ('W3', '6.941 mK'), ('W4', '6.778 mK')])]:
    obs = cm.AverageSkyObservations(name, [
        cm.SkyObservation(
```

```
            name=da_name,
            description='Raw WMAP r9 data, from http://lambda.gsfc.nasa.gov',
            T_map=('$DATA/wmap/n512/wmap_da_imap_r9_7yr_%s_v4.fits' % name, 1, 'TEMPERATURE'),
            TT_nobs_map=('$DATA/wmap/n512/wmap_da_imap_r9_7yr_%s_v4.fits' % name, 1, 'N_OBS'),
            TT_sigma0=da_sigma0,
            beam_transfer='$DATA/wmap/n512/wmap_%s_ampl_bl_7yr_v4.txt' % name,
            mask_map=('$DATA/wmap/n512/wmap_processing_mask_r9_7yr_v4.fits', 1, 0),
            lmax_beam=lmax)
        for da_name, da_sigma0 in da_list])
```

Then, model the components and foregrounds. Currently the way to get mixing maps is by loading output from Commander 1; we must then create a dict that maps observations to mixing map descriptors (by which we mean, "FITS-tuples"). Create a function that does that (i.e. maps observations to the channel number that was used in the Commander 1 run):

```
def get_mixing_maps(comp_num):
    # Create a mapping from observation to Commander 1 output file
    d = {}
    for idx, obs in enumerate(observations):
        fitsfile = '$DATA/wmap/mixing/mixmat_comp%02d_band%02d_k03999.fits' % (comp_num, idx + 1)
        d[obs] = (fitsfile, 1, 0)
    return d
```

Set up some prior power spectrums for synchrotron and dust, using NumPy arrays. We refuse them to pick up and monopole and dipole to avoid degeneracies in the system:

```
ls = np.arange(lmax + 1)
ls[0] = ls[1] # avoid NaN
prior_synch = 3e5 * ls**-2.1
prior_dust = 1e3 * ls**-1.8
prior_synch[:2] = 0
prior_dust[:2] = 0
```

Then set up our model; note that we can pass either a FITS file or a NumPy array as `power_spectrum`:

```
cmb = cm.IsotropicGaussianCmbSignal(
    name='cmb',
    power_spectrum='$DATA/wmap/wmap_lcdm_sz_lens_wmap5_cl_v3.fits',
    lmax=lmax
    )

dust = cm.MixingMatrixSignal(
    name='dust',
    lmax=lmax,
    power_spectrum=prior_dust,
    mixing_maps=get_mixing_maps(1)
    )

synchrotron = cm.MixingMatrixSignal(
    name='synch',
    lmax=lmax,
    power_spectrum=prior_synch,
    mixing_maps=get_mixing_maps(2)
    )

monodipole = cm.MonoAndDipoleSignal('monodipole', fix_at=[])

signal_components = [cmb, dust, synchrotron, monodipole]
```

Again, constructing the *..Signal* objects has no meaning by itself, they must be used; the `signal_components` list is what is really important here.

Finally, construct a *CommanderContext*, which is responsible for figuring out the parallelization scheme to use and so on:

```
ctx = cm.CommanderContext(cache_path=cache_path, seed=seed)
```

The `cache_path` should be a directory where results that are likely to be reused in other runs will be cached. The `seed` is the seed for the random number generator which you can leave as `None` if you want a new one each time.

Finally:

```
realizations = cm.app.constrained_realization(
    ctx,
    output_filename,
    observations,
    signal_components,
    wiener_only=False, # want samples
    lprecond=lprecond,
    eps=eps,
    filemode='w')
```

This will do the actual work. Note that we pass in the lists of `observations` and `signal_components`, if we didn't, none of the above would do anything!

The resulting HDF file contains some information about what went into the run etc., plus the results:

- `/power/<comp_name>`: The power spectrum of the component
- `/samples/<comp_name>`: The coefficients in m-major ordering

The latter can be dealt with by using routines in `commander.sphere.mmajor`, or by using the command-line tool to do an `alm2map` and dump to FITS file:

```
$ cmdr dumpmap --nside=512 myresults.h5 /samples/cmb 0 cmb.fits
$ cmdr dumpmap --nside=512 myresults.h5 /samples/synch 0 synch.fits
```

1.2 Joint Bayesian component separation and model estimation

The main goal of Commander is to perform joint Bayesian component separation and model estimation of the CMB data (see http://arxiv.org/abs/0709.1058). Here we describe everything that can go into a Commander analysis of CMB data; of course, one is free to sample only a subset of the parameters in a given run.

1.2.1 Input

Data

The input is a number of maps, indexed by i, each containing the following information:

Temperature and polarization maps $\mathbf{d}_i$: In the HEALPix pixelization.

Noise properties $\mathbf{N}_i$: Currently supported: RMS maps (independent noise between pixels)

Mask $\mathbf{m}_i$: Internally the mask is treated as part of $\mathbf{N}_i^{-1}$

Beams $\mathbf{B}_i$: Currently supported: Symmetric beams; input the spherical harmonic transfer function. Todo: FEBeCOP beam

Frequency band information $f(\nu)$: Either $[\nu_{\min}, \nu_{max}]$ or $F_i(\nu)$

Priors

1.2.2 Output

ξ^2 map: ...

Instrument parameters

Monopole, dipole: One per detector map i. Choose to arbitrarily fix monopole at some frequencies.

Bandpass shift:

Gain g_i:

Noise calibration factor τ_i: Multiplicative factor with τ_i

Foreground parameters

Dust $A_p^d f(\nu; T, e) g_\nu$: Priors on all parameters.

Synchrotron, $A_p^s (\frac{\nu}{\nu_{\text{ref}}}^{\beta + C \log(\frac{\nu}{\nu_{\text{ref}}})}$:

Free-free, $A_p^f (\frac{\nu}{\nu_{\text{ref}}}^{\beta_f}$ Prior: $N(-2.15, 0.02^2)$

CO $A_p^c \alpha_\nu$ Specifically, $\alpha^{217}, \alpha^{353}$

CMB model parameters

Currently only power spectrum estimation is supported:

CMB power spectrum C_ℓ, σ_ℓ : Standard inverse-Wishart

1.3 Build and installation of Commander

1.3.1 Dependencies

- NumPy
- SciPy
- Cython
- PyFITS
- libpsht and psht4py
-

1.4 Modelling signal components

Every signal component on the sky (CMB, synchrotron, free-free, etc.) comes with two classes to implement them. The first one, with the suffix *Signal*, is a *model description*:

```
synchrotron_info = cm.SynchrotronSignal(
    nu_ref=nu_ref, lmax=lmax,
    beta_prior=(-2.0, 1.0), # mean, std. dev
    )
```

The model description is immutable (carries no state), serializable, hashable, and in general does no work on construction.

Given a model description of a signal, one can create the *Sampler* for it, typically attached to a *Chain* instance:

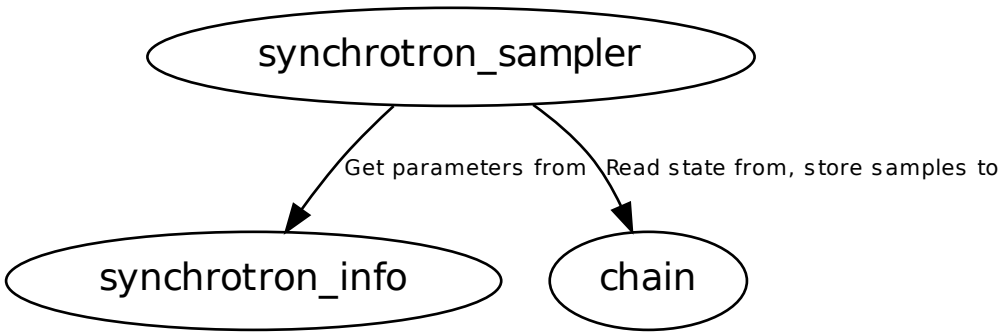
```
# Get descriptions of parameters to sample. This can depend on resolution
# of the observations
synchrotron_params = synchrotron_info.get_parameters([obs1, obs2])

# Now, we can construct a chain involving these parameters
chain = cm.Chain([obs1, obs2], synchrotron_params)

# Finally, create an instance of cm.signal.synchrotron.SynchrotronSampler
# attached to the given chain
synchrotron_sampler = cm.create_sampler(chain, synchrotron_info)
```

The sampler is stateful (may store precomputations necesarry for the sampling process etc.). On construction the sampler may read out parameters that are relevant for it from the *chain* argument passed in.

The dependencies between the constructed objects are as follows:



CHAPTER

TWO

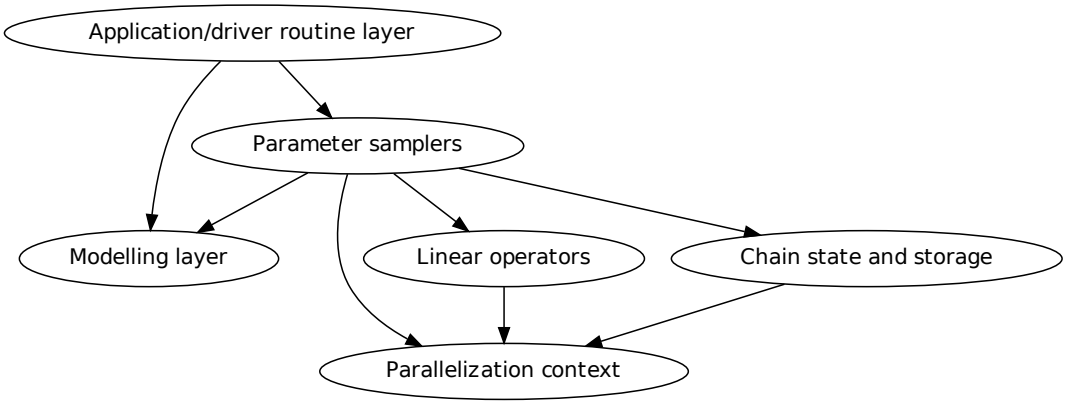
THE COMMANDER LIBRARY GUIDE

2.1 Architecture

Various aspects of Commander, ordered roughly from high-level to low-level layers.

A design goal of Commander is that the lower-level layers should be useful in their own right without involving the abstractions in the higher layers in any way.

This illustration is **not** finished:



2.1.1 Application/driver routine layer

The package `commander.apps` contain command-line applications and utilities. The most basic of these can be used directly, and are also available through scripts in `/bin`.

However, some applications require much more configuration than what can be conveyed on the command-line. In these cases, one calls the application routines while passing in something from the modelling layer which effectively acts as configuration (described below). Assuming `[V1, V2]` and `[cmb, synchrotron, <...>]` are objects from the modelling layer, one might invoke the CMB Gibbs sampler like this:

```
cm.apps.cmb_gibbs_sampler_command_line(
    observations=[V1, V2],
    signal_components=[cmb, synchrotron, free_free, monodipole],
```

```
    # <snip more options>
    chain_count=10,
    sample_count=5000,
    output_filename='wmap-9r-analysis.h5'
)
```

These *application routines* typically have two versions: One with and one without a `...command_line` suffix. The `command_line` version will in addition to the provided options parse `sys.argv` for additional options, and in general provide more defaults.

Full example: See `scripts/wmap_gibbs_sampler_runner.py`.

2.1.2 Modelling layer

This layer is the primary API to set up the problem that Commander should solve. The objects are all entirely descriptive and take few or no actions on their own. An example:

```
V1 = cm.SkyObservation(
    name='V1',
    T_map=('$WMAPPATH/wmap_da_imap_r9_7yr_V1_v4.fits', 1, 'TEMPERATURE'),
    ...
    )
cmb = cm.IsotropicGaussianCmbComponent(
    power_spectrum='$WMAPPATH/wmap_lcdm_sz_lens_wmap7_cl_v4.dat',
    lmax=1500)
synchrotron = cm.SynchrotronComponent(nu_ref=60, lmax=1500)
```

Here, *V1* and *cmb* are *descriptors*, their role is to carry around information/configuration, not to do anything. The constructors will resolve the filenames to absolute paths (so that a later `os.chdir` does not alter the contents), but no attempt is made to load any data.

While the constructor is not allowed to do anything intensive, model objects are allowed to provide methods that do loading or computation. The loading/computation will typically depend on the parallelization and so require a *CommanderContext*. E.g.:

```
ctx = cm.CommanderContext(my_mpi_communicator)

# load single map
map = V1.load_map(ctx)
# load many maps in parallel, more efficient
map_lst, rms_lst, mask_lst = cm.load_map_rms_mask(ctx, [V1, V2])

# Constructing the mixing operator of synchrotron requires some
# computation (integrating over the bandpass of the observation several times
# and compute a spline)
M_V1_sync = synchrotron.get_mixing_operator_sh(ctx, V1, beta=-2.2, C=2.12)
```

2.1.3 MCMC chains

The *Chain* and *MemoryChainStore* / *HDF5ChainStore*' classes cooperate to provide the abstractions for running an MCMC/Gibbs chain. They don't do any sampling by themselves, but primarily allow a) persistence of chain, and b) passing around information about current chain state. By using these classes the boilerplate is greatly reduced from a typical MCMC loop, while still allowing for programming the chain in the good old imperative manner. See *Chain* documentation (TODO).

2.1.4 Linear operators

Commander uses the `oomatrix` library to deal with linear algebra. The library allows for plugging in your own linear operators, and many such operators have to be provided by Commander, e.g., spherical harmonic transforms and efficient component mixing. See `scripts/simple_constrained_realization.py` for an example of constrained realization being quickly implemented using only the linear operators.

2.1.5 Parallelization context

The *CommanderContext* (typically named `ctx` in code) is responsible for storing the parallelization scheme (and provide good defaults), and is conventionally passed as the first argument to many routines. Parallelization is coordinated by passing around the *CommanderContext* object. E.g., in the following code, it is assumed that the call is done collectively *with the same arguments*, and after the call *map* only contains the rings of the local rank:

```
map = cm.load_map(ctx, V1)
```

Note that it is **not** allowed to alter the arguments to load different bands on each rank:

```
map = cm.load_map(ctx, observations[comm.Get_rank()])
```

Rather, it is the responsibility of the *CommanderContext* to coordinate which map lives where:

```
# THIS WAY OF PARTITIONING IS NOT IMPLEMENTED, and may not be, the
# point is just for now to provide an example of the
# responsibilities of CommanderContext
ctx = CommanderContext(comm=comm_with_two_ranks,
                       observations=[V1, V2],
                       partition_by_observation=True)
map_lst, rms_lst, mask_lst = cm.load_map_rms_mask(ctx, [V1, V2])
# Now, rank 0 contains all V1 data and rank 1 contains all V2 data
```

2.2 `commander.sphere`: Working with spherical data

The `commander.sphere` package contains utilities for working with spherical data. The tools here are independent of the rest of Commander; the rest of Commander is heavily based on the data format and conventions present here.

2.2.1 Spherical harmonic conventions

Ordering scheme

Across Commander we use m-major ordering with interleaved positive and negative m; i.e., the coefficients (ℓ, m) are ordered first by $|m|$, then by ℓ, and finally by ordering the positive m before the negative m.

Examples: For all m and $0 \leq \ell \leq 2$, the coefficient order is

$$\left[a_{0,0}, a_{1,0}, a_{2,0}, a_{1,1}, a_{1,-1}, a_{2,1}, a_{2,-1}, a_{2,2}, a_{2,-2}\right],$$

whereas for storing only $3 \leq \ell \leq 4$ one gets

$$\left[a_{3,0}, a_{4,0}, a_{3,1}, a_{3,-1}, a_{4,1}, a_{4,-1}, a_{3,2}, a_{3,-2}, \ldots a_{4,4}, a_{4,-4}\right].$$

The routines `commander.sphere.mmajor.lm_to_idx()` and `commander.sphere.mmajor.idx_to_lm()` can translate between array indices and (ℓ, m).

Spherical harmonic transforms process the coefficients one m at the time, making m-major ordering the most efficient one for SHTs. Memory locality aside, it is crucial that all coefficients for a given m reside on the same node.

In various computations $a_{\ell,m}$ and $a_{\ell,-m}$ are often related and treated together, so it makes sense to keep them together. When interleaving in this particular way with positive m first, converting between *half-complex* and *real* spherical harmonic formats is just a matter of in-place coefficient scaling with no reordering (except possibly for $m=0$). In the half-complex format, $a_{\ell,m}$ is seens as the real component and $a_{\ell,-m}$ the imaginary.

We avoid any padding. Padding could have made for more convenient indexing, but is impractical in linear algebra, as one is forced to also pad matrices accordingly. Also it becomes easy to mis-count the number of degrees of freedom for vectors and so on.

Real vs. complex spherical harmonics

We deal with real fields on the sphere, and with the traditional complex definition of spherical harmonics we have $a_{\ell m} = (-1)^m a_{\ell -m}$ and the negative m are redundant. *However*, they must still be included in many computations (you can **not** stack only the non-negative m in a vector and do linear algebra with the result).

Therefore, to make sure that linear algebra with spherical harmonic vectors always work out well (and to avoid a very significant speed penalty), we mainly use *real spherical harmonics*, related to complex spherical harmonics by the equations

$$
\begin{aligned}
a^R_{\ell,0} &= a^C_{\ell,0} && (2.1)\\
a^R_{\ell,m} &= \sqrt{2}\mathrm{Re}(a^C_{\ell,m}) && \text{for } m > 0 \quad (2.2)\\
a^R_{\ell,m} &= \sqrt{2}\mathrm{Im}(a^C_{\ell,-m}) && \text{for } m < 0 \quad (2.3)\\
& && (2.4)
\end{aligned}
$$

Algebraically, the transformation from complex to real spherical harmonics is an orthonormal transformation.

Warning: While the conversion of *vectors* is trivial, the conversion of *matrices* between the real and complex spherical harmonic bases is a bit less trivial, since one needs to apply the basis change operator on both sides of the matrix. In the case of sparse matrices it may even sometimes be better to stick with the full expanded complex vectors, as the number of non-zero elements are smaller.

When it comes to storage, we do not define any *half-complex* format (i.e., where one stores only non-negative m), as one can simply read the real coefficients and correct for the $\sqrt{2}$ on the fly. Both *real* and *complex* storage formats are in use, both using the m-major ordering described above with either real or complex coefficients.

2.2.2 Reference

2.3 `commander.memoize`: Reusing results

Perhaps the primary reason to use memoization is the convenience during debugging to save computational results to disk (even if memoization is often turned off for cluster runs).

But another reason for memoization is to structure the program. If a temporary result is memoized, there's no need to explicitly pass it around:

```
V = SkyObservation(name='V', source='WMAP7yr', ...)
x = compute_frobnification(V) # uses square of RMS map inside
y = compute_bartification(V) # also uses square of RMS map inside
```

In "traditional" programming, one *should* (and do, if the computation time is large) figure out everything that the two functions share in terms of temporary result, compute that in the caller, and pass it in. But that can become unwieldy, and when it does, memoization is your friend.

Note: A big part of what's memoziation is used for in Commander is simply to read in the input data.

Warning: Use memoization sparingly. (Unless until we get a better handle on it.) The problem doesn't really disappear, because one now needs to figure out when to release the results from the memoization cache, which can't be done perfectly by any heuristic (and at the time of writing we don't have any heuristics but just fill up the store forever).

2.3.1 Core idea

Memoized results are always associated with an explicit `MemoContext` (of which `CommanderContext` is a subclass). You annotate a function/method with `@memoize`/`@memoize_method`:

```
from commander.memoize import memoize, memoize_method

class MyContext(MemoContext):
    @memoize_method('description_of_result', tags=['disk'])
    def method(self, ctx, arg): ...

@memoize() # default name of result is 'compute_foo'
def compute_foo(ctx, arg): ...
```

The `ctx` argument is special, and is where the cached results are stored.

For this to work,

- all input arguments must support hashing
- the result must support being made immutable

The tags can be arbitrary strings, which are then picked up on by the memoization policies. The only one currently supported is `"disk"` which causes it to be stored to disk cache (if disk cache is enabled).

Note: The reason for the separate `@memoize_method` is that the context is the second argument rather than the first. In subclasses of `MemoContext` one should use `@memoize_in_self`.

2.3.2 Hashing protocol

TODO

2.3.3 Immutabilification

The results from a memoized function will be converted to read-only (or if this is not possible, such as a with a dict, an exception will be raised). The conversion process is recursive through lists.

If the type is not known, the memoization will try to call an `as_immutable(self)` method to convert the result.

2.3.4 Comparison with joblib

The Joblib library is intended to enter your own (presumably dynamic, interactive) workflow with a minimum of intervention. Therefore it, e.g., checks that the source code has changed (and if so invalidates the cache for the function), tries to hash almost any input (by running it through a hashing "pickler"), and so on. A typical joblib example is:

```
@cache
def f(arr): return fft(arr**2) # arr is a NumPy array
```

On the other hand, `commander.memoize` is very explicit. It will not try to hash NumPy arrays for instance, instead you typically use descriptors which you use to fetch the data . Typical example:

```
@memoize(, tags=['disk'])
def f(ctx, map_descriptor):
    arr = ctx.get_foo(map_descriptor)
    return fft(arr)
```

CHAPTER

THREE

THE COMMANDER DEVELOPER'S GUIDE

3.1 Conventions

- Every module should start with `from __future__ import division`
- Commander imported as `cm`, NumPy as `np`, SciPy as `sp`, oomatrix as `om`, matplotlib.pyplot as `plt`

3.2 Random number generation

Each program typically need two RNGs: One that is guaranteed to be different between ranks (given the variable name `nc_rng`, for non-collective RNG), and one that has the same seed for every rank and is always called collectively (called `c_rng`).

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*