CosmoSIS Documentation

Release 1.4

The CosmoSIS Team

Jun 07, 2021

Contents:

1 CosmoSIS Overview 3

2 Installing CosmoSIS 7

3 Tutorials: Getting Started 13

4 Debugging 25

5 Reference 27

6 FAQ 67

7 Indices and tables 73

Documentation Under Construction!

CosmoSIS is a **cosmological parameter estimation code**. It is now at version 1.4.

It is a framework for structuring cosmological parameter estimation in a way that eases re-usability, debugging, verifiability, and code sharing in the form of calculation modules.

It consolidates and connects together existing code for predicting cosmic observables, and makes mapping out experimental likelihoods with a range of different techniques much more accessible

CosmoSIS is described in *Zuntz et al <http://arxiv.org/abs/1409.3409>* - if you make use of it in your research, please cite that paper and include the URL of this repository in your acknowledgments. Thanks!

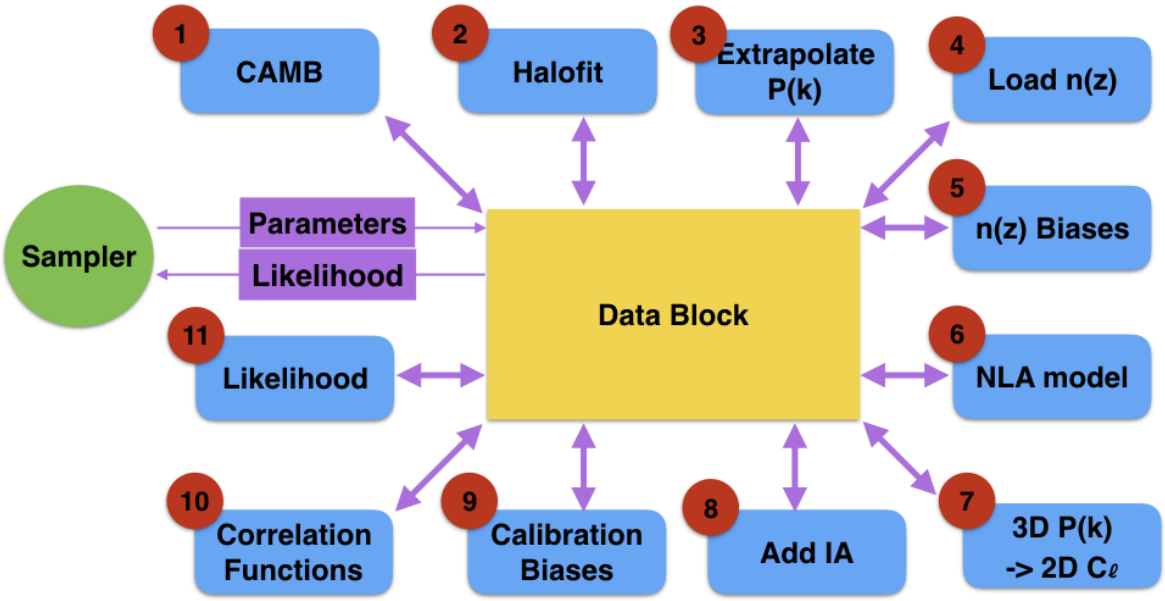
CHAPTER 1

CosmoSIS Overview

1.1 What CosmoSIS does

CosmoSIS connects together **samplers**, which decide how to explore a cosmological parameter space, with **pipelines** made from a sequence of **modules**, which calculate the steps needed to get a likelihood functions. You have to decide on what calculations your likelihood function should consist of and choose or write modules that perform them.

Here is an example schematic of a CosmoSIS run of a weak lensing analysis:



- The green sampler generates parameters and sends them to the pipeline. At the end it gets a total likelihood back.
- The blue modules are independent codes run in the numbered sequence. They each perform one step in the calculation of the likelihood.
- The purples connections show each module reading inputs from the datablock, and then saving their results back to it.
- The yellow DataBlock acts like a big lookup table for data. It stores the initial parameters and then the results from each module.
- The blue Likelihood module (number 11) has a special output - it computes a final likelihood value.

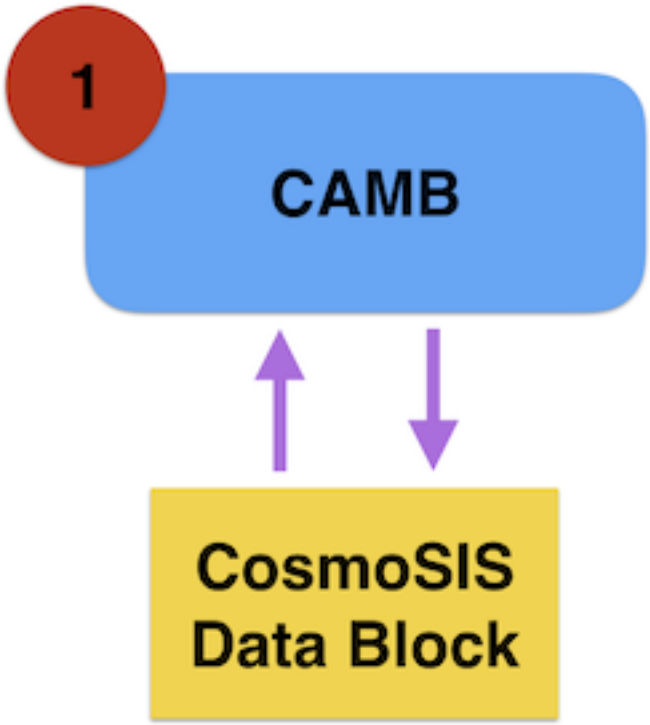
1.2 Modules

For many cosmological models, calculating the theoretical prediction of a model is a long and complicated process with many steps. CosmoSIS organizes each of those steps as a separate piece of code, called a module. The CosmoSIS Standard Library comes with a large collection of popular cosmology codes packaged as modules, and it is easy to make your own modules.

For example, we package the Boltzmann code CAMB as a module - it takes in cosmological parameters as inputs and calculates cosmological power spectra as outputs.

To become a CosmoSIS module, a piece of code in python, C, C++, or Fortran just needs to have two specially named

functions in it: `setup` and `execute`. The setup function is run once, a the start of the CosmoSIS process. The execute function is run again each time there are new input parameters for which the sampler wants the likelihood.



1.3 DataBlocks

CosmoSIS modules do not send their calculations on to each other directly. Instead they communicate only with CosmoSIS, via a DataBlock. A DataBlock is a look-up table that takes a pair of strings as keys (a section and a name) and maps them to a value, which can be either a single scalar value (integer, real, or complex number, or a string) or a (multi-dimensional) array of them.

1.4 Samplers

In CosmoSIS a *sampler* is a code that generates a series of sample sets of parameters on which the likelihood must be run. Some of these are trivial, like the *test* sampler, which just generates a single sample. Others are much more complex and explore the parameter space dynamically. Most of them are what statisticians would genuinely consider to be a *sampler* - they generate samples from the posterior distribution.

There are many different samplers packaged with the code that are useful in different circumstances. They are all interact with the rest of CosmoSIS in very similar ways - they are configured from options in a parameter file, and they build a likelihood pipeline which they run whenever they have a new sample whose posterior they want to evaluate.

1.5 User Interface

CosmoSIS decides what analysis to perform using three configuration files that you write (one of them optional). Theyt are all in the "ini" format, and are:

- The parameter file, which describes the pipeline of the model you want to run and the sampler you want to run on it. The `setup` functions for modules look in this file.
- The values file, which describes the numerical input parameters, some of which will probably be varied throughout the run. The samplers decide how to vary the parameter within the ranges it gives and modules look for these parameters in their `execute` functions.
- The priors file, which optionally describes additional priors on the input parameters. There is always an implicit Uniform prior on the value; adding a prior here creates an additional prior. If you do not set a priors file then there will be no additional priors.

When you execute CosmoSIS you tell it the parameter file on the command line. The parameter file tells CosmoSIS where to find the other two files.

The "ini" format splits a file into sections, which are named using square brackets, and keys and values within those sections. For example:

```
[section_name]
parameter_name = value
```

Values can be integers, doubles, complex, or strings.

CHAPTER 2

Installing CosmoSIS

2.1 Bootstrap Installation

2.1.1 Platforms

We have an installation script which installs cosmosis and all its dependencies that works on 64 bit **Mac OSX Mavericks** (10.9) and **Scientific Linux 6** or its equivalents (**CentOS, Redhat Enterprise**).

Everything this script installs is put in a single directory which can be cleanly deleted and will not interfere with your other installed programs.

2.1.2 Before starting on all platforms

You need `git` and `curl` to run the bootstrap installer. You can check with these commands:

```
curl --help
git --help
```

If either gives you a message like "Command not found" then you can install them from the git website or the curl website.

You also need to be running the bash shell. Check with:

```
echo ${SHELL}
```

If it does not say "bash" then you can either change your shell to bash permanently (search for instructions for your operating system) or manually run `bash` before each time you use CosmoSIS.

2.1.3 Before starting on MacOS

You need the XCode developer tools on a Mac. First check if you have them by running:

```
xcode-select --install
```

If the message includes the text `command line tools are already installed` then you can skip to the next section.

Otherwise, first check if you have XCode by running:

```
which -a gcc
```

if it includes `/usr/bin/gcc` then you have it already. Otherwise, install XCode from the apple website or from the the Mac App Store (it is free).

Next, install the XCode command line tools. This requires `sudo` powers:

```
sudo xcode-select --install
```

2.1.4 Before starting on Red Hat 6.x, Scientific Linux 6.x, and other derivatives

On RHEL 6.x derivatives, you need to have the following set of RPM packages installed:

```
redhat-lsb-core
libpng-devel
freetype-devel
lapack-devel
git
```

You can check this by running:

```
yum list redhat-lsb-core libpng-devel freetype-devel lapack-devel git
```

If any of them appear underneat the "Available packages" section of the output, then you will need to have someone with superuser (admin) privileges install the packages listed using this command:

```
su - -c "yum install redhat-lsb-core libpng-devel freetype-devel lapack-devel git"`
```

2.1.5 Running the bootstrap

Run these lines to get and run the script:

```
curl -L --remote-name https://bitbucket.org/mpaterno/cosmosis-bootstrap/raw/master/
↪cosmosis-bootstrap-linux
chmod u+x cosmosis-bootstrap
./cosmosis-bootstrap cosmosis
cd cosmosis
```

2.1.6 Setting up the environment

Each time you use CosmoSIS you need to first do this:

```
source config/setup-cosmosis
```

2.1.7 Compiling

Build the code like this:

```
make
```

You need to do this again whenever you modify C, C++, or Fortran code. If you just change python code you don't need to do it again.

2.1.8 Note

A note for people who understand git: this procedure leaves you in a "detached head" state at the latest version. You can get to the bleeding edge with: git checkout master

2.2 Docker Installation

Docker is a virtual machine-like system that lets us package up all the cosmosis dependencies in a known system, called a "Container". Working in this container is slightly different to what you may be used to, but is simple enough.

The main advantage here for docker is that it gives a very specific installation environment so any problems you have we can instantly reproduce. The main disadvantages are disk space - you have to download an *image* of the OS you are using - and the fact that you are working on a different file and operating system than you are used to inside docker.

The docker documentation is terrible for typical scientist users - it mostly assumes you want to run web applications.

You need admin rights to install docker, but not to use it, so if you have a system administrator it is a one-off task for them.

1. Install and start docker for your system
    - Installation on MacOS (get the stable version)
    - Installation on Windows (get the stable version)
    - On Linux, docker is available in most package managers like apt and yum. You can also try the more opaque instructions on the docker site.
2. `git clone https://bitbucket.org/joezuntz/cosmosis-docker` This will download the cosmosis-docker installer
3. `cd cosmosis-docker`
4. `./get-cosmosis-and-vm ./cosmosis` This download cosmosis. Wait a little while for the download to complete.
5. `./start-cosmosis-vm ./cosmosis` #This starts you inside docker. Read what it says on the screen.
6. First time only: `update-cosmosis --develop` to get the development version which has some fixes in.
7. First time only, or when you change any C/C++/F90 code: `make`

2.3 Manual Installation

We recommend using one of the automatic installation methods described above. They install everything neatly in a single directory and will not mess up your current installation of anything. Even if you're an expert it really is much easier.

If you can't or don't want to use the bootstrap method then you can manually install CosmoSIS.

2.3.1 Library dependencies

You need to install several libraries and programs before installing CosmoSIS:

- CAMB
- gcc/g++/gfortran 4.8 or above
- gsl 1.16 or above
- cfitsio 3.30 or above
- FFTW 3
- lapack (except on MacOS)
- git
- python 2.7

2.3.2 Python dependencies

You also need to install several python packages before installing CosmoSIS. These are also listed in *config/requirements.txt*:

- astropy
- Cython
- matplotlib
- numpy
- PyYAML
- scikit-learn
- scipy
- CosmoloPy
- emcee
- fitsio
- kombine

2.3.3 Download

Download CosmoSIS and the standard library using git:

```
git clone http://bitbucket.org/joezuntz/cosmosis
cd cosmosis
git clone http://bitbucket.org/joezuntz/cosmosis-standard-library
cd cosmosis-standard-library
cd ..
```

2.3.4 Setup script

From the cosmosis directory make a copy of the manual setup script:

```
cp config/manual-install-setup setup-cosmosis
```

Edit the new file setup-my-cosmosis and replace all the places where it says `/path/to/XXX` in this file with correct paths based on how you installed things.

2.3.5 Build

Source the setup script and make:

```
source setup-cosmosis
make
```

If you get any errors then check your setup-cosmosis script for errors, and whether there are any environment variables like `LD_LIBRARY_PATH` set before you start.

2.3.6 Usage

Each time you start a new terminal shell then you need to repeat this step:

```
source setup-cosmosis
```

Then test your install is working by *following our first tutorial*.

CHAPTER 3

Tutorials: Getting Started

The CosmoSIS demos show you the basics of running cosmosis to do different things. They assume a basic understanding of cosmological models.

3.1 Tutorial 1: Computing a single cosmology likelihood

This tutorial shows you how to build a pipeline to evaluate a single cosmology likelihood, in this case the Planck satellite's 2015 measurements of the cosmic microwave background.

3.1.1 Installation

Before you start, *follow the instructions to install cosmosis*. Once you are complete, you should be able to run:

```
cosmois --help
```

and see a usage message.

3.1.2 Parameter files

Have a look at the file `demos/demo2.ini`. CosmoSIS is always run on a single parameter file like this. It specifies how to construct a pipeline that generates a likelihood, and what to do with that likelihood once it is calculated.

CosmoSIS parameter files use the `ini` format, which has section names with square brackets around them and parameters specified with an equals sign. This example says there is a section called `runtime` with a parameter named `sampler` with the values "test":

```
[runtime]
sampler = test
```

Each CosmoSIS run uses at least two ini files, this one, called a parameter file, and a second *values* file, specifying the cosmological and other varied parameters used in the pipeline. In this case the values file is `demos/values2.ini`.

3.1.3 Running CosmoSIS on a parameter file

Run CosmoSIS on this parameter file with this command:

```
cosmosis demos/demo2.ini
```

You will see a lot of output showing:

- What parameters are used in this pipeline, e.g.

```
Parameter Priors
----------------
planck--a_planck                      ~ delta(1.0)
cosmological_parameters--h0           ~ delta(0.6726)
cosmological_parameters--omega_m      ~ delta(0.3141)
...
```

- The set-up phase for each step (module) in the calculation, e.g.:

```
-- Setting up module camb --
 camb mode  =            1
 camb cmb_lmax =         2650
 camb FeedbackLevel =            2
 accuracy boost =    1.1000000000000001
 HighAccuracyDefault =  T
 ...
```

- The sampler that is being run:

```
****************************
* Running sampler 1/1: test
****************************
```

- The output of each module, e.g.:

```
Running camb ...
Reion redshift       =  11.751
Integrated opt depth =  0.0800
Om_b h^2             =  0.018096
Om_c h^2             =  0.123356
Om_nu h^2            =  0.000644
...
```

3.1.4 Defining a sampler

The first lines in the parameter file `demos/demo2.ini` are:

```
[runtime]
sampler = test

[test]
save_dir=demo_output_2
fatal_errors=T
```

The first option, `sampler`, tells CosmoSIS what it should do with the likelihood that we will construct - how the parameter space should be *sampled*.

CosmoSIS has lots of different samplers in it, designed to move around parameter spaces in different ways. The `test` sampler is the simplest possible one: it doesn't move around the parameter space at all - it just computes a likelihood (runs the pipeline) for a single set of values. These tutorials will discuss several samplers; the full list is described in *the samplers page*.

Once you have chosen a sampler you configure that sampler with the second section shown in above, which has the name of the sampler, in this case `test`.

3.1.5 Defining a pipeline

Cosmological analyses use a *Likelihood Function* - a calculation of the probability of the observed data given some cosmological model. In toy problems these likeliood functions are often just simple analytic functions. In realistic cosmological problems they are usually long calculations with many parts.

In CosmoSIS you build up a likelihood function from a sequence of steps called *modules*. Each module does a different piece of the calculation, often modelling different pieces of physics and different observed data sets. You need to understand the calculation you are trying to do to build a CosmoSIS pipelines, and then put together the ingredients that it needs.

The pipeline is defined in the parameter file like this:

```
[pipeline]
modules = consistency camb planck bicep2
...
likelihoods = planck2015
```

This tells CosmoSIS to run four modules, and to expect a likelihood called "planck2015" at the end. The names of modules are not fixed - they refer to section names in the rest of the parameter file. For example, the `planck` module is specified futher down like this:

```
[planck]
file = cosmosis-standard-library/likelihood/planck2015/planck_interface.so
data_1 = ${COSMOSIS_SRC_DIR}/cosmosis-standard-library/likelihood/planck2015/data/
↪plik_lite_v18_TT.clik
data_2 = ${COSMOSIS_SRC_DIR}/cosmosis-standard-library/likelihood/planck2015/data/
↪commander_rc2_v1.1_l2_29_B.clik
```

The first option, which all modules must have, tells CosmoSIS where to find the file containing the code of this module. The other two options, `data_1` and `data_2` are passed to the module. In general it can do whatever it likes with them, but in this case the Planck module uses them to decide which data sets to generate the likelihood for.

The modules in this example are all part of the CosmoSIS Standard Library. For your own analyses you could mix standard library modules with your own steps. We have a list of all the standard library modules and their options, inputs, and outputs in the standard library reference.

3.1.6 Defining input values

The pipeline we have built is a machine for turning a collection of numerical parameters into a single total likelihood. We need some initial input values for the first module to take in:

```
[pipeline]
...
values = demos/values2.ini
```

This option points to the values file, the second cosmosis ini file. The values file contains all the inputs that are passed to the pipeline. For example:

```
[cosmological_parameters]
h0 = 0.6726       ;H0 (km/s/Mpc)/100.0km/s/Mpc
omega_m = 0.3141  ;density fraction for matter today
omega_b = 0.04    ;density fraction for baryons today
omega_k = 0.0     ;spatial curvature
```

This creates a category of parameters called `cosmological_parameters` and within that a collection of named values. The semi-colons begin comments.

Parameters can either have a fixed value, like the ones above, or they can have a range, like this:

```
[cosmological_parameters]
h0 = 0.6   0.6726   0.8
```

This doesn't make any difference for the test sampler, because it just uses the one central value. But if you are sampling, as in the next tutorial, then that is the range that the parameters can take.

3.2 Tutorial 2: Running an MCMC sampler

In the first tutorial we generated a single likelihood. In this tutorial we we run an MCMC analysis to explore a parameter space and put constraints on some parameters. There are lots of different MCMC algorithms available through CosmoSIS; in this example we will use one called emcee, which is popular in astronomy.

3.2.1 Running an MCMC

Have a look at `demos/demo5.ini` and its values file `demos/demo5.ini`.

Let's try using MPI parallelism to speed up this analysis. Run this command:

```
mpirun -n 4 cosmosis --mpi demos/demo5.ini
```

If that fails straight away then you may not have MPI installed (MPI should work automatically with the bootstrap and docker installation methods - let us know if not). If it fails then you can fall back to serial mode:

```
cosmosis demos/demo5.ini
```

Demo 5 runs a supernova likelihood, using the JLA supernova sample, which measures the redshift-distance relation. The code will take a few minutes to run, and will generate a file called `demo5.txt` as output. This file will contain a Monte Carlo Markov Chain (MCMC) which you can use as samples from the posterior probability distribution of the model given the data.

3.2.2 Demo 5 Parameter File

This time the parameter file contains these lines:

```
[runtime]
; The emcee sampler, which uses the Goodman & Weare algorithm
sampler = emcee

[emcee]
walkers = 64
samples = 400
nsteps = 100
```

This tells CosmoSIS to use the emcee sampler, and configures it to use 64 walkers (points exploring the parameter space). It tells it to generate 400 samples per walker, and to save results to disc every 100 steps.

3.2.3 The output file

The parameter file also contains these lines:

```
[output]
filename = demo5.txt
format = text
verbosity= debug
```

This tells the code to generate an output file called `demo5.txt`, in text format. You could also specify `fits` to get a FITS format file. Our first demo in tutorial 1 didn't produce an output chain, so it didn't need this section.

Whichever sampler you use, CosmoSIS output files always have the same format. Comment lines are all preceded with a #, so that chains can be read easily with most tools. The first line is a header which tells you what the different columns mean:

```
#cosmological_parameters--omega_m   supernova_params--deltam    supernova_params--
↪alpha supernova_params--beta  supernova_params--m post
```

The first entries are the varied parameters from the values file. They are shown in the form: `section_name--parameter_name`. After this any parameters generated by the sampler are shown. In this case that just means `post` - the log-posterior of this row. Other samplers might also generate other outputs such as weights.

The next lines are metadata and show the name of the sampler, the number of varied parameters, the pipeline that was run, papers you should cite for the given pipeline, and options passed to the sampler. Finally, the three parameter files are all copied into the output file so you can check later exactly what you ran.

NB: The verbosity settings are a little bit confused in the code right now, so this option might not do that much.

All the samplers except the test sampler produce this chain file. Some produce other files too.

3.3 Tutorial 3: Making plots and statistics using postprocess

The `cosmosis` command runs the CosmoSIS samplers and generates either an output chain file (for most samplers) or a directory (for the test sampler).

The `postprocess` command knows how to take these files or directories and generate a collection of image and text files describing the chains.

Different samplers produce different kinds of outputs - for example, some produce samples with weights per sample, some require an initial burn-in to be removed, and some like the Fisher Matrix sampler don't produce samples at all. The postprocess command knows how to deal with each different kind of output.

3.3.1 Running postprocess

The postprocess command should be run on the chain file or test directory that is output by the sampler. In tutorial 2 you ran the `emcee` sampler and made an output file `demo5.txt`. Now we can process that command to generate some plots and statistics. Run this command:

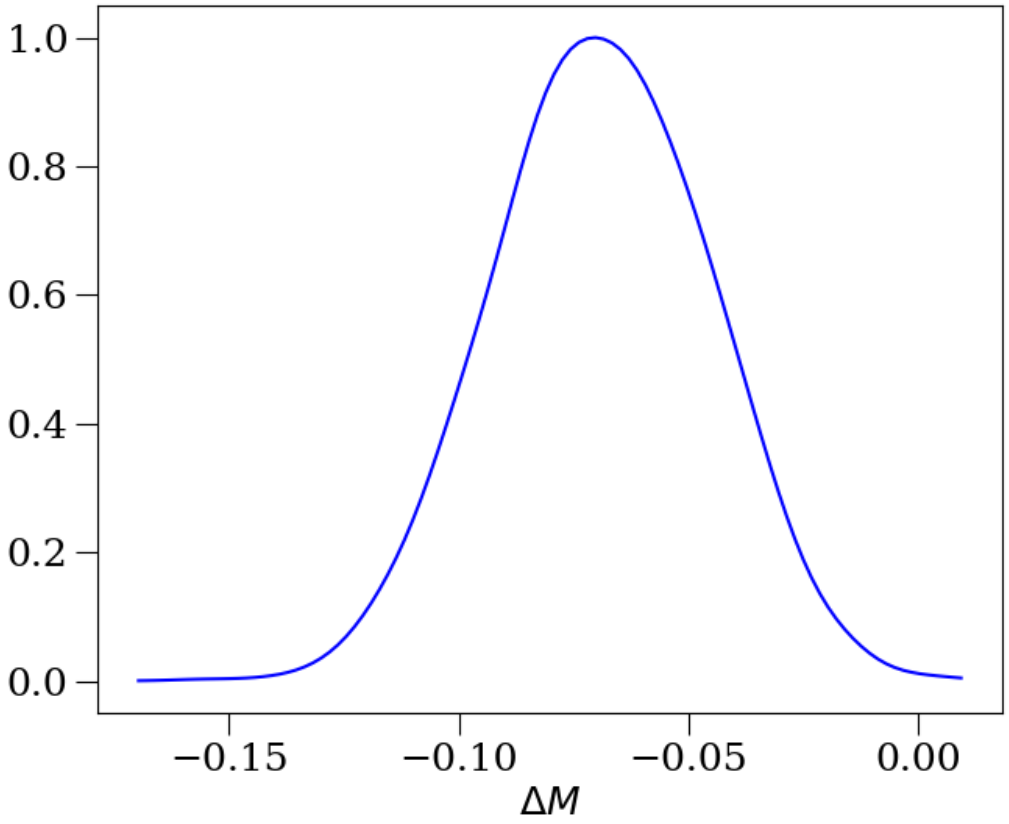
```
postprocess -o plots -p demo5 --burn 5000 demo5.txt
```

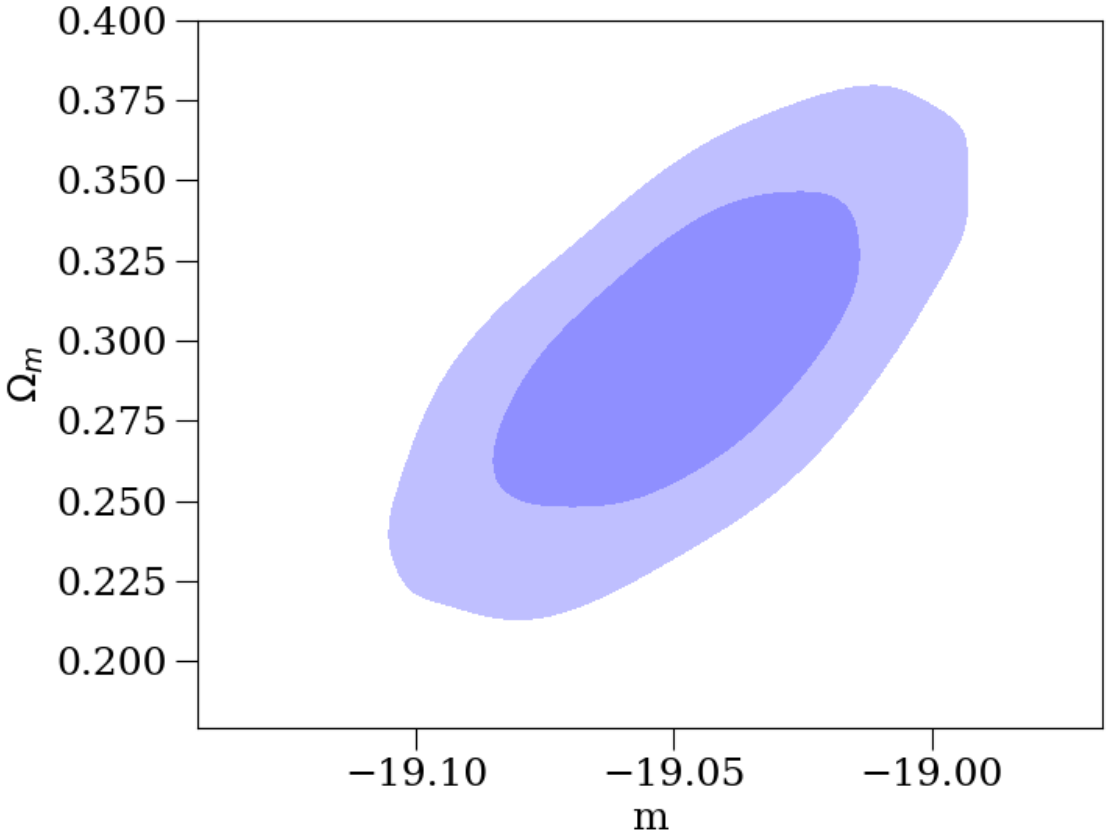
This will create a group of files in the `plots` directory with names starting `demo5_`

The `--burn` flag specifies the number of samples to remove from the start of the chain - this is required for many MCMC samplers, including emcee. You should examine your chain and cut off any initial portion where the samples have not reach a steady state.

3.3.2 Postprocess outputs: plots

Postprocess produces a collection of plots showing the 1D and 2D parameter contours. Here are some examples:





These are estimates of the posterior distribtion of the parameters, smoothed using a Kernel Density Estimation process.

The 2D plots show 68% and 95% credible intervals.

3.3.3 Postprocess outputs: statistics

Postprocess also produces text files containing

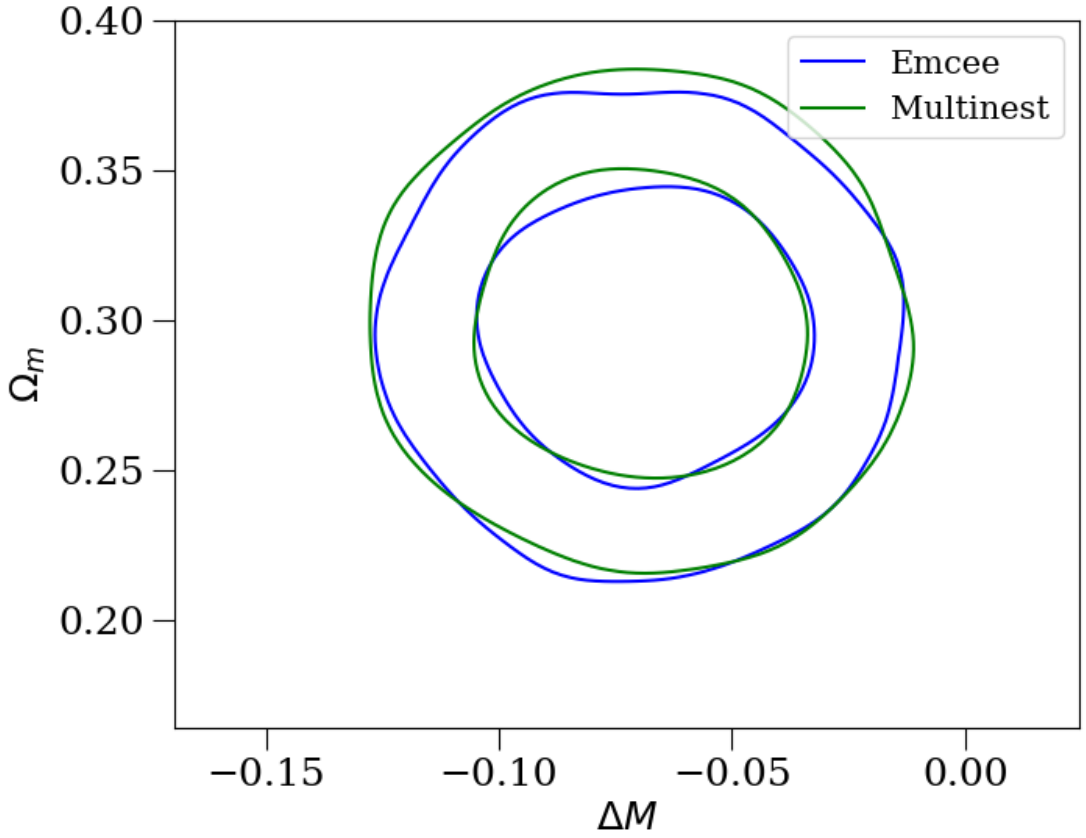
3.3.4 Multiple plots

You can give the postprocess command two or more chain files and it will produce plots with multiple sets of contours.

In this case you probably want to set the flag `--no-fill` to make the 2D contours easier to read, and the flag `--legend="Chain 1 Name|Chain 2 Name"` to add a legend to the plot.

Here is an example comparing chains from the multinest and emcee samplers, which I made using the command:

```
postprocess demo5.txt  demo9.txt -o plots -p compare_multinest --burn 5000 --no-fill -
↪-legend="Emcee|Multinest"
```



3.3.5 Controlling plot output

The 2D plots made using KDE can take some time to generate, especially for large chains. You can avoid making 2D plots altogether using the postprocess flag `--no-2d` or any plots using `--no-plots`. The latter can be useful on systems where matplotlib is not available.

You can also select subsets of plots to make using the flags `--only=xxx` and `--either=yyy`, which restrict 2D plots to only make plots where both of the parameter names start with `xxx` and either of them start with `yyy`.

See the reference page for the postprocess command line for lots more details.

3.4 Tutorial 4: Building new pipelines

In CosmoSIS a *pipeline* is a sequence of *modules* that computes one or more *likelihoods*.

You can build a new pipeline or modify an existing one by choosing which modules to include in the pipeline, and by selecting their configuration.

3.4.1 Extending a calculation by adding modules

To know which module to add you have to understand the calculation you want to perform, and make sure that all the necessary calculations for it are done at some point in the pipeline. You can add modules into the middle of an existing

pipeline, or at the end.

3.4.2 Adding the new module

In the `[pipeline]` section of `demos/demo2.ini` you will find:

```
[pipeline]
modules = consistency camb planck bicep2
```

Let's modify this pipeline, by removing the old BICEP2 likelihood, and adding a more recent and accurate one, the BOSS DR12 Baryon Acoustic Oscillation measurement. We can consult the list of standard library modules. on the CosmoSIS wiki to find out what we will need.

The BOSS DR12 page gives some basic details about the likelihood

It tells us the file we need to use for the likelihood:

```
File: cosmosis-standard-library/likelihood/boss_dr12/boss_dr12.py
```

This tells us to make a new section in the parameter file that we are using, with this information in. We can add new text to the bottom of `demos/demo2.ini`, like this:

```
[boss]
file = cosmosis-standard-library/likelihood/boss_dr12/boss_dr12.py
```

3.4.3 Configuring the module

The wiki page also tells us what parameters we can use to configure the pipeline, and what inputs the likelihood will need. The only mandatory parameter is described like this:

```
mode    0 for BAO only, 1 for BAO+FS measurements
```

Let's use both the BAO and full-shape information, so we can set it to 1. So our new parameter file section becomes this:

```
[boss]
file = cosmosis-standard-library/likelihood/boss_dr12/boss_dr12.py
mode = 1
```

3.4.4 Running the module

Right now, nothing will change if we run this pipeline, because we have not told CosmoSIS to use this new module. We can do so by changing the modules option from above to this:

```
[pipeline]
modules = consistency camb planck boss
```

This tells CosmoSIS to look for a section called "boss" in the parameter file and configure a module based on it.

It's fine to include unused modules in the parameter file - it can be useful later when you run different variations of a similar pipeline.

3.4.5 The missing growth function

If we run this pipeline with `cosmosis demos/demo2.ini` then we will get this error:

```
cosmosis.datablock.cosmosis_py.errors.BlockSectionNotFound: 3: Could not find section␣
↪called growth_parameters (name was z)
```

This is because the logic of our pipeline didn't add up - we need the growth rate of cosmic structure to calculate this likelihood, but we never calculated it. In fact the wiki page above showed us a table of inputs we needed for the pipeline, but never supplied.

A quick search of the list of standard modules shows us several modules that calculate the growth factor. They make different assumptions about physics, so which we should use will depend on our science case. In this case, let's use the one called growth_factor which solves the growth factor differential equation to calculate it. You'll notice that the missing `growth_parameters` section is listed as one of the outputs it generates.

So let's add this module to the bottom of the parameter file:

```
[growth]
file = cosmosis-standard-library/structure/extract_growth/extract_growth.py
```

and change our pipeline like this:

```
[pipeline]
modules = consistency camb planck growth boss
```

3.4.6 The total likelihood

There's one final change we need to make - we need to tell CosmoSIS to add the BOSS likelihood to the total:

```
likelihoods = planck2015 boss_dr12
```

(in fact if we had just left out the likelihoods line altogether CosmoSIS would have done this by default. It's only because we explicitly listed planck that the issue arose).

If we do this and run then the pipeline will calculate both our likelihoods, and their total:

```
    Likelihood planck2015 = -1726.9467027368746
    Likelihood boss_dr12 = -4.313138812597072
Likelihood total = -1731.2598415494717
```

So we have successfully extended our pipeline!

3.5 Tutorial 5: Writing new modules

So far we've used modules from the CosmoSIS standard library to build our pipeline. This works fine for many projects, where using or modifying the standard library will be enough. But we can go further by creating entirely new modules to calculate new observables or include new physics effects in them.

3.6 Pipelines & Modules

CosmoSIS modules are isolated from each other - all the calculations done by a module are stored in a DataBlock, which is passed through the list of modules in the pipeline. Each module reads what previous modules have done,

performs its own calculations, and then adds these to the pipeline.

Typically, a pipeline is run many times on different sets of input parameters, for an example in an MCMC.

3.7 Module requirements

CosmoSIS modules can be written in C, C++, Fortran, Python, or (experimentally) Julia.

In each of these languages the structure of the file you write is the same: you write functions with specific names. Two of these are required, and one is optional:

```
setup(options)
# Takes the information from the parameter file as a DataBlock and configures the␣
↪module.
# This is called once, when the pipeline is created.

execute(block, config)
# Takes a DataBlock (see below) from the values file and any preivous pipeline, and␣
↪runs the module.
# This is called for each set of parameters the pipeline is run on.

cleanup(config)
# (Optional) frees memory allocated in the setup function
# This is run once, when the pipeline is deallocated.
# We will skip it in this tutorial, as it is rarely needed in python.
```

3.8 Writing our new module

Let's write our new module in Python, since that's usually much easier.

Make a new file. You can put it anywhere, but to keep things organized it's usally better to keep your work separate from the existing standard library.

In that file, put these lines, which represent an empty but runnable CosmoSIS module:

```
from cosmosis.datablock import names, option_section

def setup(options):
    return {}

def execute(block, config):
    return 0
```

CHAPTER 4

Debugging

CosmoSIS has a number of tools to help you debug problems with your analyses.

4.1 The Test Sampler

The first thing to try when debugging an MCMC failure is the test sampler - try to see if you can recreate the problem when running on a single parameter set. Then you can switch on the options below for more information about the error.

You can set the parameter:

```
[test]
fatal_errors = T
```

to get more output on any error in the test sampler.

4.2 Parameter file options

You can set several options in the CosmoSIS parameter file to find problems more easily:

```
[pipeline]
quiet=F
debug=T
```

Setting `quiet=F` will print when each stage is reached so you can tell where errors happen.

Setting `debug=T` will, after an error, print out a log of all the values saved to the CosmoSIS data block - if you are getting an error saying that some value has not been found in the block then this can be useful to figure out why.

Note that this setting prints out a lot of text *after* any other errors are reported, so you may need to scroll up past it to look for any other messages.

4.3 Python errors

If you are finding a crash in a Python function then you can use the `--pdb` flag on the command line to enter the Python Debugger. When the program crashes you will be dropped into an interactive debugger shell and you can print the values of parameters or run functions to try to determine the source of the error:

```
cosmosis my_params.ini --pdb
```

To do this in the test sampler you may need to set the parameter `fatal_errors=T` in the `[test]` section of the parameter file.

This option doesn't work when running in parallel.

4.4 Compiling in debug mode

You can compile all CosmoSIS code in debug mode to make finding errors in compiled code easier. To do so, run these commands in the main directory:

```
export COSMOSIS_DEBUG=1
make clean
make
```

4.5 Crashes in C/C++/Fortran code

If you are seeing a crash in C/C++/Fortran code you can use the command line flag `--experimental-fault-handling` to try to track it down. If you are using python 2 you first need to install a package to support this:

```
pip install faulthandler # python 2 only; it comes with python3
```

Then:

```
cosmosis my_params.ini --experimental-fault-handling
```

On a crash this should print out a traceback revealing in which function the error ocurred.

4.6 Special Samplers

CosmoSIS has several samplers that explore the parameter space in ways that help debug or compare to other code.

The *apriori* sampler samples points from throughout the prior. It can be useful if your error is only triggered by extreme parameters.

The *list* sampler takes a list of parameter sets to evaluate one by one. It can be useful if you have a set of possible parameters that might cause errors, or which you want to compare to another code.

The *star* sampler samples a set of 1D parameters slices in each parameter direction through the central point. It can be useful to compare to the output of another code.

CHAPTER 5

Reference

5.1 Samplers

Samplers are the different methods that CosmoSIS uses to choose points in parameter spaces to evaluate.

Some are designed to actually explore likelihood spaces; others are useful for testing and understanding likelihoods.

5.1.1 The Abcpmc sampler

Approximate Bayesian Computing (ABC) Population Monte Carlo (PMC)

Name	abcpmc
Version	0.1.1
Author(s)	Joel Akeret and contributors
URL	http://abcpmc.readthedocs.org/en/latest/
Citation(s)	Akeret, J., Refregier, A., Amara, A, Seehars, S., and Hasner, C., JCAP (submitted 2015), Beaumont et al. 2009 arXiv:0805.2256, Fillippi et al 2012 arXiv:1106.6280
Parallelism	parallel

abcpmc is a Python Approximate Bayesian Computing (ABC) Population Monte Carlo (PMC) implementation based on Sequential Monte Carlo (SMC) with Particle Filtering techniques. This likelihood free implementation estimates the posterior distribution using a model to simulate a dataset given a set of parameters. A metric rho is used to determine a distance between the model and the data and parameter values are retained if rho(model,data) < epsilon. This epsilon threshold can be fixed or linearly or exponentially modified every iteration in abcpmc. abcpmc uses a set of N particles to explore parameter space (theta), on the first iteration, t=0, these are chosen from the prior. On subsequent iterations, t, another N particles are selected with a perturbation kernal K(theta(t) | theta(t-1)) using twice the weighted covariance matrix. It is extendable with k-nearest neighbour (KNN) or optimal local covariance matrix (OLCM) pertubation kernels.

This implementation of abcpmc in CosmoSIS requires an understanding of how ABC sampling works and we recommend you contact the CosmoSIS team for specific implementaion questions; we would be very happy to help out!

Installation

pip install abcpmc #to install centrally, may require sudo

pip install abcpmc –user #to install just for you

Parameters

These parameters can be set in the sampler's section in the ini parameter file. If no default is specified then the parameter is required. A listing of "(empty)" means a blank string is the default.

Parameter	Type	Meaning	Default
niter	integer	T - number of iterations	2
distance_func	str	def func(x,y):nt do some calculationnt return dist_result	None
epimin	double	epsilon at t=T	1.0
ngauss	int	dimension of multigaussian if run_multigauss	4
epimax	double	epsilon at t=0	5.0
particle_prop	string	Particle proposal kernal, options = weighted_cov, KNN, OLCM	weighted_cov
npart	integer	number of particles	
run_multigauss	boolean	generate multigaussian data	F
metric_kw	str	mean, chi2 or other; if “other” then need to specify name of function “distance_func	“chi2”
num_nn	integer	number of neighbours if using particle_prop = KNN	10
threshold	string	Various different threshold implementations, options = LinearEps, ConstEps, ExpEps	LinearEps

5.1.2 The Apriori sampler

Draw samples from the prior and evaluate the likelihood

Name	apriori
Version	1.0
Author(s)	CosmoSIS Team
URL	http://dan.iel.fm/emcee/
Citation(s)	
Parallelism	parallel

This sampler draws samples from the prior and evaluates their likelihood. The main current use of this is to help test for misbehaviour in calculation modules in extreme regions of parameter space.

Installation

None required

Parameters

These parameters can be set in the sampler's section in the ini parameter file. If no default is specified then the parameter is required. A listing of "(empty)" means a blank string is the default.

Parameter	Type	Meaning	Default
save_name	string	If set, save sample data to directories save_name_0, save_name_1, etc.	(empty)
nsample	integer	number of samples to draw	

5.1.3 The Emcee sampler

Ensemble walker sampling

Name	emcee
Version	2.1.0
Author(s)	Dan Foreman-Mackey and contributors
URL	http://dan.iel.fm/emcee/
Citation(s)	PASP, 125, 925, 306-312
Parallelism	parallel

The emcee sampler is a form of Monte-Carlo Markov Chain that uses an ensemble of 'walkers' that explore the parameter space. Each walker chooses another walker at random and proposes along the line connecting the two of them using the Metropolis acceptance rule. The proposal scale is given by the separation of the two walkers.

It is parallel, so multiple processes can be used to speed up the running. It is also affine invariant, so that no covariance matrix or other tuning is required for the proposal.

The burn-in behavior of emcee can sometimes be poor; it is much better to start the chain as near to the maximum posterior as possible. The maxlike or similar samplers can help you find this.

The total number of samples taken is walkers*samples.

Installation

emcee is included in the cosmosis bootstrap, but if you are installing manually you can get emcee using the command:

pip install emcee #to install centrally, may require sudo

pip install emcee –user #to install just for you

Parameters

These parameters can be set in the sampler's section in the ini parameter file. If no default is specified then the parameter is required. A listing of "(empty)" means a blank string is the default.

Parameter	Type	Meaning	Default
random_start	bool	whether to start the walkers at random points in the prior instead of near the start. Usually a bad idea	N
covmat	string	a file containing a covariance matrix for initializing the walkers.	(empty)
samples	integer	number of jumps to attempt per walker	
start_points	string	a file containing starting points for the walkers. If not specified walkers are initialized randomly from the prior distribution.	(empty)
nsteps	integer	number of sample steps taken in between writing output	
walkers	integer	number of walkers in the space	

5.1.4 The Fisher sampler

Fisher matrix calculation

Name	fisher
Version	1.0
Author(s)	CosmoSIS Team
URL	https://bitbucket.org/joezuntz/cosmosis
Citation(s)	
Parallelism	embarrassing

The Fisher information matrix characterizes the curvature of a distribution, typically at its peak. For multi-dimensional parameter spaces it can be given in matrix form, with each element representing the curvature between two parameters.

Fisher matrices can be used to approximate the full likelihood shape one would derive from a sampling process, but much faster as only a handful of likelihood evaluations are needed. This approximation is exact for perfectly Gaussian posteriors but not otherwise; in particular for distributions with "Banana-like" degeneracies or cut-offs near the peak it is a poor approximation.

For a general distribution the Fisher matrix can be rather fiddly to calculate; this sampler does not do that. Instead it assumes a Gaussian likelihood, in which case the fisher matrix can be computed from to the derivatives of the observable quantities v with respect to the parameters p, and their covariance matrix C :

$$F_{ij} = \sum_{mn} \frac{\partial v_m}{\partial p_i} [C^{-1}]_{mn} \frac{\partial v_n}{\partial p_j}$$

There is an additional term that arises when the covariance matrix C depends on the parameters p, which we do not currently calculate here, as it is usually fixed in cosmology. We plan to implement this shortly, however.

The CosmoSIS fisher sampler is calculated around the central value provided in the values file; no optimization is done before running.

Unlike most other CosmoSIS samplers which depend only likelihood of a parameter set, the fisher sampler requires the predicted observables from a pipeline too. They are expected to be saved to a section of the data block, called "data_theory", with keys, name+"_theory" and name+"_inverse_covariance" where "name" is the name of the likelihood, for example, to save a cmb data vector you could save "cmb_theory" and "cmb_inverse_covariance" in the "data_vector" section.

Gaussian likelihoods implemented in CosmoSIS save these sections automatically. Your own likelihoods can use the CosmoSIS gaussian likelihood superclass to do the same, or you can manually save name+"_theory" and name+"_inverse_covariance" for any data that you add.

Installation

No special installation required; everything is packaged with CosmoSIS

Parameters

These parameters can be set in the sampler's section in the ini parameter file. If no default is specified then the parameter is required. A listing of "(empty)" means a blank string is the default.

Parameter	Type	Meaning	Default
step_size	float	The size, as a fraction of the total parameter range, of steps to use in the derivative calculation. You should investigate stability wrt this.	0.01

5.1.5 The Grid sampler

Simple grid sampler

Name	grid
Version	1.0
Author(s)	CosmoSIS Team
URL	https://bitbucket.org/joezuntz/cosmosis
Citation(s)	
Parallelism	embarrassing

Grid sampling is the simplest and most brute-force way to explore a parameter space. It simply builds an even grid of points in parameter space and samples all of them.

As such it scales extremely badly with the number of dimensions in the problem: n_sample = grid_size ^ n_dim and so quickly becomes unfeasible for more than about 4 dimensions.

It is extremely useful, though, in lower dimensions, where it is perfectly parallel and provides a much smoother and more exact picture of the parameter space than MCMC methods do. It is also useful for taking lines and planes through the parameter space.

The main parameter for the grid sampler is the number of sample points per dimension (grid_size above).

Installation

No special installation required; everything is packaged with CosmoSIS

Parameters

These parameters can be set in the sampler's section in the ini parameter file. If no default is specified then the parameter is required. A listing of "(empty)" means a blank string is the default.

Parameter	Type	Meaning	Default
nsample_dimension	integer	The number of grid points along each dimension of the space	
save	string	If set, a base directory or .tgz name for saving the cosmology output for every point in the grid	(empty)

5.1.6 The Gridmax sampler

Naive grid maximum-posterior

Name	gridmax
Version	1.0
Author(s)	CosmoSIS Team
URL	https://bitbucket.org/joezuntz/cosmosis
Citation(s)	
Parallelism	parallel

This sampler is a naive and experimental attempt to try a parallel ML/MAP sampler.

It samples through the dimensions of the space one-by-one, and draws a line of evenly space samples through that dimension, and finds the best fit along that line.

It then changes that parameter value to the best fit, leaving all the others fixed. The idea is that at moves in on a "square spiral" towards the best fit.

The sampling is parallel along the line.

This sampler is experimental and should probably only be used for testing purposes.

Installation

No special installation required; everything is packaged with CosmoSIS

Parameters

These parameters can be set in the sampler's section in the ini parameter file. If no default is specified then the parameter is required. A listing of "(empty)" means a blank string is the default.

Parameter	Type	Meaning	Default
output_ini	string	if present, save the resulting parameters to a new ini file with this name	(empty)
tolerance	real	tolerance for Delta Log Like along one complete loop through the dimensions	0.1
nsteps	integer	The number of sample points along each line through the space	

5.1.7 The Importance sampler

Importance sampling

Name	importance
Version	1.0
Author(s)	CosmoSIS Team
URL	https://bitbucket.org/joezuntz/cosmosis
Citation(s)	
Parallelism	embarrassing

Importance sampling is a general method for estimating quantities from one distribution, P', when what you have is samples from another, similar distribution, P. In IS a weight is calculated for each sample that depends on the difference between the likelihoods under the two distributions.

IS works better the more similar the two distributions are, but can also be useful for adding additional constraints to an existing data set.

There's a nice introduction to the general idea in Mackay ch. 29: http://www.inference.phy.cam.ac.uk/itila/book.html

Installation

No special installation required; everything is packaged with CosmoSIS

Parameters

These parameters can be set in the sampler's section in the ini parameter file. If no default is specified then the parameter is required. A listing of "(empty)" means a blank string is the default.

Parameter	Type	Meaning	Default
input_filename	string	cosmosis-format chain of input samples	
nstep	integer	number of samples to do between saving output	128
add_to_likelihood	bool	include the old likelihood in the old likelihood; i.e. P'=P*P_new	N

5.1.8 The Kombine sampler

Clustered KDE

Name	kombine
Version	0.01
Author(s)	Benjamin Farr
URL	https://github.com/bfarr/kombine
Citation(s)	Farr, B. and Farr, W.M., "kombine: a kernel-density-based, embarrassingly parallel ensemble sampler", in preparation., http://arxiv.org/abs/1309.7709
Parallelism	parallel

kombine is an ensemble sampler that uses a clustered kernel-density-estimate proposal density, which allows it to efficiently sample multimodal or non-gaussian posteriors. In between updates to the proposal density estimate, each member of the ensemble is sampled independently, allowing for massive parallelization.

The total number of samples generated will be walkers * samples.

Installation

kombine needs to be installed separately: it can be installed from github using pip:

pip install –user git+git://github.com/bfarr/kombine

Parameters

These parameters can be set in the sampler's section in the ini parameter file. If no default is specified then the parameter is required. A listing of "(empty)" means a blank string is the default.

Parameter	Type	Meaning	Default
update_interval	integer	number of steps taken in between updating the posterior	
nsteps	integer	number of sample steps taken in between writing output	
walkers	integer	number of independent walkers in the ensemble	
samples	integer	total sample steps taken	
start_file	string	a file containing starting points for the walkers. If not specified walkers are initialized randomly from the prior distribution.	(empty)

5.1.9 The List sampler

Re-run existing chain samples

Name	list
Version	1.0
Author(s)	CosmoSIS Team
URL	https://bitbucket.org/joezuntz/cosmosis
Citation(s)	
Parallelism	embarrassing

This is perhaps the second simplest sampler - it simply takes all its samples from a list in a file and runs them all with the new pipeline.

This could probably be replaced with an importance sampler, and may be merged into it in future.

Installation

No special installation required; everything is packaged with CosmoSIS

Parameters

These parameters can be set in the sampler's section in the ini parameter file. If no default is specified then the parameter is required. A listing of "(empty)" means a blank string is the default.

Parameter	Type	Meaning	Default
save	string	if present the base-name to save the cosmology output from each sample	(empty)
filename	string	cosmosis-format chain of input samples	

5.1.10 The Maxlike sampler

Find the maximum likelihood using various methods in scipy

Name	maxlike
Version	1.0
Author(s)	SciPy developers
URL	http://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.optimize.minimize.html
Citation(s)	
Parallelism	serial

This sampler attempts to find the single point in parameter space with the highest likelihood. It wraps a variety of samplers from the scipy.minimize package.

These methods are all iterative and local, not global, so they can only find the nearest local maximum likelihood point from the parameter starting position.

Maximum likelihood using these kinds of methods can be something of a dark art. Results can be quite sensitive to the starting position and exact parameters used, so if you need precision ML then you should carefully explore the robustness of your results.

These samplers are wrapped in the current version of scipy:

- Nelder-Mead
- Powell
- CG
- BFGS
- Newton-CG
- Anneal (deprecated by scipy)
- L-BFGS-B
- TNC
- COBYLA
- SLSQP
- dogleg

- trust-ncg

Each has different (dis)advantages, and which works best will depend on your particular application. The default in CosmoSIS is Nelder-Mead. See the references on the scipy URL above for more details.

Some methods can also output an estimated covariance matrix at the likelihood peak.

Installation

Requires SciPy 0.14 or above. This is installed by the CosmoSIS bootstrap, but if you are installing manually you can get it with the command:

pip install scipy #to install centrally, may require sudo

pip install scipy –user #to install just for you

Parameters

These parameters can be set in the sampler's section in the ini parameter file. If no default is specified then the parameter is required. A listing of "(empty)" means a blank string is the default.

Parameter	Type	Meaning	Default
output_ini	string	if present, save the resulting parameters to a new ini file with this name	(empty)
maxiter	integer	Maximum number of iterations of the sampler	1000
tolerance	real	The tolerance parameter for termination. Meaning depends on the sampler - see scipy docs.	1e-3
output_covmat	string	if present and the sampler supports it, save the estimated covariance to this file	(empty)
method	string	The minimization method to use.	Nelder-Mead

5.1.11 The Metropolis sampler

Classic Metropolis-Hastings sampling

Name	metropolis
Version	1.0
Author(s)	CosmoSIS Team
URL	https://bitbucket.org/joezuntz/cosmosis
Citation(s)	Journal of Chemical Physics 21 6 1087-1092 (1953)
Parallelism	multi-serial

Metropolis-Hastings is the classic Monte-Carlo Markov Chain method for sampling from distributions.

MH as a Markov process where from each point in chain there is a process for choosing the next point in such a way that the distribution of chain points tends to the underlying distribution.

In MH a proposal function is defined that suggests a possible next point in the chain. The posterior of that point is evaluated and if: P_new / P_old > U[0,1] where U[0,1] is a random number from 0-1, then the new point is 'accepted' and becomes the next chain element. Otherwise the current point is repeated.

The choice of proposal function strongly determines how quickly the sampler converges to the underlying distribution. In particular a covariance matrix approximately describing the distribution provides a significant speed up.

The CosmoSIS metropolis sampler tries to mirror the CosmoMC MH implementation (though we do not yet have the fast-slow split).

Metropolis-Hastings is intrinsically a serial (non-parallel) algorithm. Like CosmoMC, CosmoSIS parallelizes it by running several independent chains in parallel and comparing them to assess convergence using the Gelman-Rubin test.

Installation

No special installation required; everything is packaged with CosmoSIS

Parameters

These parameters can be set in the sampler's section in the ini parameter file. If no default is specified then the parameter is required. A listing of "(empty)" means a blank string is the default.

Parameter	Type	Meaning	Default
random_start	bool	whether to start the chains at random points in the prior instead of the ini file start	N
Rconverge	float	when multiple chains are run, use this as the Gelman-Rubin statistic	-1.0
covmat	string	load a covariance matrix from this file. The parameter order should match the order of varied parameters in the ini file	(empty)
nsteps	integer	number of points between saving data and testing convergence	100
samples	integer	number of steps	

5.1.12 The Minuit sampler

Find the maximum posterior using the MINUIT library

Name	minuit
Version	1.0
Author(s)	SciPy developers
URL	https://seal.web.cern.ch/seal/MathLibs/Minuit2/html/
Citation(s)	
Parallelism	serial

This sampler attempts to find the single point in parameter space with the highest posterior probability. It is a wrapper around the powerful MINUIT2 library that is widely used in particle physics.

Minuit is one of the more robust optimizers, but you should still try starting the sampler from a few different points to make sure they converge to the same place.

By default this wrapper uses the MIGRAD algorithm, which is pretty robust unless there are sharp edges in the parameter space. It also re-parameterizes so that the formal parameter edges (the limits in your values file) are shifted to +- infinity.

At the end of the sampling a covariance estimate is also returned.

Note on parallelism: The minuit2 sampler can be used in parallel, but the version that is packaged with the CosmoSIS auto-installer does not support that yet, so at the moment we are only supporting serial sampling (no MPI).

Installation

Requires the Minuit2 library. the auto-installer includes Minuit2, but if you are installing manually you may need to download and install from your package manager or the URL above. You will also need to set the MINUIT2_INC and MINUIT2_LIB environment variables in your setup script to point to the directories of the minuit2 headers and libraries respectively. (The minuit2 headers directory has two subdirectories, called Minuit2 and Math. The MINUIT2_INC should point to the parent directory, not the subdir).

Parameters

These parameters can be set in the sampler's section in the ini parameter file. If no default is specified then the parameter is required. A listing of "(empty)" means a blank string is the default.

Parameter	Type	Meaning	Default
verbose	bool	Print more information to the command line.	F
algorithm	string	Choose from migrad, simplex, and fallback. Migrad is better unless there are strange parameter space cliffs. Fallback tries migrad first and if it fails tries simplex.	migrad
output_ini	string	if present, save the resulting parameters to a new ini file with this name	(empty)
save_dir	string	If set, save the data block containing the cosmology at the best-fit point to this directory name	(empty)
width_estimate	float	A guess of the parameter posterior widths as a fraction of their range. Can speed convergence the more accurate it is but does not need to be very exact.	0.05
maxiter	integer	Maximum number of likelihood calculations to do	1000
strategy	string	Choose from fast, medium, and safe. Safe mode means slower convergence but less chance of failure. Fast means the opposite.	medium

5.1.13 The Multinest sampler

Nested sampling

Name	multinest
Version	3.7
Author(s)	Farhan Feroz,Mike Hobson
URL	https://ccpforge.cse.rl.ac.uk/gf/project/multinest/
Citation(s)	arXiv:0809.3437, arXiv:0704.3704, arXiv:1306.2144
Parallelism	parallel

Nested sampling is a method designed to calculate the Bayesian Evidence of a distribution, for use in comparing multiple models to see which fit the data better.

The evidence is the integral of the likelihood over the prior; it is equivalent to the probability of the model given the data (marginalizing over the specific parameter values): B = P(D|M) = int P(D|Mp) P(p|M) dp

Nested sampling is an efficient method for evaluating this integral using members of an ensemble of live points and steadily replacing the lowest likelihood point with a new one from a gradually shrinking proposal so and evaluating the integral in horizontal slices.

Multinest is a particularly sophisticated implementation of this which can cope with multi-modal distributions using a k-means clustering algorithm and a proposal made from a collection of ellipsoids.

The output from multinest is not a set of posterior samples, but rather a set of weighted samples - when making histograms or parameter estimates these must be included.

The primary mulitnest parameter is the number of live points in the ensemble. If this number is too small you will get too few posterior samples in the result, and if it is too large the sampling will take a long time. A few hundred seems to be reasonable for typical cosmology problems.

One odd feature of the multinest output is that it doesn't save any results until it has done a complete run through the parameter space. It then starts again on a second run, and sometimes a third depending on the parameters. So don't worry if you don't see any lines in the output file for a while.

Installation

No special installation required; everything is packaged with CosmoSIS

Parameters

These parameters can be set in the sampler's section in the ini parameter file. If no default is specified then the parameter is required. A listing of "(empty)" means a blank string is the default.

Parameter	Type	Meaning	Default
live_points	integer	Number of live points in the ensemble	
random_seed	integer	Seed to use for random proposal; -1 to generate from current time. Allows re-running chains exactly	-1
feedback	bool	Print out progression information from multinest	T
resume	bool	If you previously set multinest_outfile_root you can restart an interrupted chain with this setting	F
wrapped_params	str	Space separated list of parameters (section–name) that should be given periodic boundary conditions. Can help sample params that hit edge of prior.	(empty)
multinest_outfile_root	str	In addition to CosmoSIS output, save a collection of multinest output files	(empty)
ins	boolean	Use Importance Nested Sampling (INS) mode - see papers for more info	True
efficiency	float	Target efficiency for INS - see papers	0.1

5.1.14 The Pmc sampler

Adaptive Importance Sampling

Name	pmc
Version	1.0
Author(s)	CosmoSIS Team
URL	https://bitbucket.org/joezuntz/cosmosis
Citation(s)	MNRAS 405.4 2381-2390 (2010)
Parallelism	embarrassing

Population Monte-Carlo uses importance sampling with an initial distribution that is gradually adapted as more samples are taken and their likelihood found.

At each iteration some specified number of samples are drawn from a mixed Gaussian distribution. Their posteriors are then evaluated and importance weights calculated. This approximate distribution is then used to update the Gaussian mixture model so that it more closely mirrors the underlying distribution.

Components are dropped if they are found not to be necessary.

This is a python re-implementation of the CosmoPMC alogorithm in the cited paper.

Installation

No special installation required; everything is packaged with CosmoSIS

Parameters

These parameters can be set in the sampler's section in the ini parameter file. If no default is specified then the parameter is required. A listing of "(empty)" means a blank string is the default.

Parameter	Type	Meaning	Default
components	integer	Number of components in the Gaussian mixture	5
iterations	integer	Number of iterations (importance updates) of PMC	30
student	boolean	Do not use this. It is a not yet functional attempt to use a Student t mixture.	F
final_samples	integer	Samples to take after the updating of the mixture is complete	5000
samples_per_iteration	integer	Number of samples per iteration of PMC	1000
nu	float	Do not use this. It is the nu parameter for the non-function Student t mode.	2.0

5.1.15 The Snake sampler

Intelligent Grid exploration

Name	snake
Version	1.0
Author(s)	CosmoSIS Team
URL	https://bitbucket.org/joezuntz/cosmosis
Citation(s)	ApJ 777 172 (2013)
Parallelism	parallel

Snake is a more intelligent version of the grid sampler that avoids taking large number of samples with a low likelihood, which the naive grid sampler nearly always does.

It does ultimately have the same bad behaviour as you go to a higher number of dimensions, though you can push it higher than with the straight grid.

The Snake sampler maintains a list of samples on the interior and surface of the parameter combinations it has explored. This allows it to first move gradually towards the maximum likelihood and then gradually diffuse outwards from that point in all the different dimensions.

Snake outputs can be postprocessed in exactly the same way as grid samples, with missing entries assumed to have zero posterior.

Installation

No special installation required; everything is packaged with CosmoSIS

Parameters

These parameters can be set in the sampler's section in the ini parameter file. If no default is specified then the parameter is required. A listing of "(empty)" means a blank string is the default.

Parameter	Type	Meaning	Default
threshold	float	Termination for difference betwen max-like and highest surface likelihood	4.0
nsample_dimension	integer	Number of grid points per dimension	10
maxiter	integer	Maximum number of samples to take	100000

5.1.16 The Star sampler

Simple star sampler

Name	star
Version	1.0
Author(s)	CosmoSIS Team
URL	https://bitbucket.org/joezuntz/cosmosis
Citation(s)	
Parallelism	embarrassing

Installation

No special installation required; everything is packaged with CosmoSIS

Parameters

These parameters can be set in the sampler's section in the ini parameter file. If no default is specified then the parameter is required. A listing of "(empty)" means a blank string is the default.

Parameter	Type	Meaning	Default
nsample_dimension	integer	The number of star points along each dimension of the space	
save	string	If set, a base directory or .tgz name for saving the cosmology output for every point in the star	(empty)

5.1.17 The Test sampler

Evaluate a single parameter set

Name	test
Version	1.0
Author(s)	CosmoSIS Team
URL	https://bitbucket.org/joezuntz/cosmosis
Citation(s)	
Parallelism	serial

This is the most trivial possible 'sampler' - it just runs on a single parameter sample. It is mainly useful for testing and for generating cosmology results for plotting.

The test sampler uses the starting position defined in the value ini file, and runs the pipeline just on that.

At the end of the run it will print out the prior and likelihood (if there is one), and can optionally also save all the data saved along the pipeline, so that you can make plots of the useful cosmological quantities.

Experimental: we have a new test feature where you can plot your pipeline and the data flow through it as a graphical diagram. This requires pygraphviz and the graphviz suite to make an image - use a command like: dot -Tpng -o graph.png graph.dot

Installation

No special installation required; everything is packaged with CosmoSIS. If you want to make a graphical diagram of your pipeline you need to have pygraphviz installed. You can get this with:

pip install pygraphviz #to install centrally, may require sudo

pip install pygraphviz –user #to install just for you

You also need graphviz to turn the result into an image.

Parameters

These parameters can be set in the sampler's section in the ini parameter file. If no default is specified then the parameter is required. A listing of "(empty)" means a blank string is the default.

Parameter	Type	Meaning	Default
fatal_errors	bool	Any errors in the pipeline trigger an immediate failure so you can diagnose errors	N
graph	string	Requires pygraphviz. Save a dot file describing the pipeline	(empty)
save_dir	string	Save all the data computed in the pipeline to this directory (can also end with .tgz to get a zipped form)	(empty)

5.2 CosmoSIS Parameter Files

5.2.1 Parameter files

CosmoSIS uses three configuration files to describe an analysis:

- The parameter file defines the sampler, output, and pipeline.
- The values file defines the input parameters and their allowed ranges.
- The priors file defines additional priors on the parameters.

Details about these files are given on these sub-pages. This page describes common features of the ini format that they all use.

Parameter Files

Options that are set in the parameter file but never used will just be silently ignored, so check your spelling carefully.

This can be very useful if you have some options that you only switch on sometimes but not others.

Runtime Options

A section called `[runtime]` must be present in the parameter file. It has one mandatory parameter, `sampler`, and one optional one, `root`.

The `sampler` parameter must be one of the CosmoSIS samplers. It determines how the parameter space is explored.

The `root` parameter changes how paths to modules in the parameter file are looked for. All paths are defined relative to the root. By default this is just the CosmoSIS directory:

```
[runtime]
sampler = metropolis
; This is the default value of root:
root = ${COSMOSIS_SRC_DIR}
```

Sampler Options

The specific sampler that you chose in the `[runtime]` section must also be configured in a section

Different samplers have different options - some mandatory and some which take a default value if you don't set them. The pages for the individual samplers describe what parameter they can take. For example, the emcee sampler might take these parameters:

```
[emcee]
walkers = 64
samples = 400
nsteps = 100
```

Output Options

All the samplers except the test sampler generate output chain files. You can choose where this should go and what format it should have in the `output` section. The only mandatory parameter is `filename`:

```
[output]
filename = my_output_chain.txt
; the default format is "text".  It can also be set to "fits".
format = text
verbosity = noisy
```

The verbosity parameter can be set to any of these values:

```
highest
debug
noisy
standard
gentle
quiet
silent
```

Or a value from 0 (silent) to 50 (highest). The verbosity settings in the code are a little mixed up right now, and the biggest effect can actually be had by modifying the "quiet" parameter described below in the `[pipeline]` section. This will be addressed in future releases.

Pipeline Options

The mandatory `pipeline` section defines the pipeline that you want to run to generate a likelihood.

It must have these parameters:

```
[pipeline]
modules = consistency camb jla riess11
values = demos/values5.ini
likelihoods = jla riess
quiet=T
debug=F
timing=F
extra_output = cosmological_parameters/Yhe
```

The **modules** parameter must contain a sequence of the modules that make up the pipeline. Each module is a single collection of code that performs a piece of the calculation. See also the overview section for more information on the concept modules, the list of CosmoSIS standard modules, and details on how to make your own modules. The names used in this option can be absolutely anything, as long as there is a correspondingly named section somewhere else in the file (see Module Options below).

The **values** parameter must point to the values file, which defines what parameters are put in to the start of the pipeline. The path is relative to the current working directory, not to the parameter file.

The **likelihoods** parameter defines which likelihoods are extracted from the pipeline. After running the pipeline CosmoSIS looks in the data block for parameters called `X_LIKE` for each X in the list given here. If this parameter is not set then all available likelihoods are used.

The **quiet** parameter tells the code to minimize the output it prints out if it is set to "T".

The **debug** parameter can be set to "T" to print out more detailed debugging output if the pipeline fails. It will print out the complete list of all things written to and read from the pipeline.

The **timing** parameter can be set to "T" to print out how long each stage in the pipeline takes to configure and run.

The **extra_output** parameter can be used to save derived parameters in the output chain. Set it to a sequence of strings each of the form `section/name`, and after the pipeline is run each these will be extracted from the output as a new column.

Module Options

Every entry found in the `modules` option in the `pipeline` section (see above) must have a section with the same name in the parameter file.

That section must have at least one mandatory parameter in it, `file`:

```
[pipeline]
; We must include two sections below with these names:
modules = my_theory_module   my_likelihood_module

[my_theory_module]
file = modules/path/to/theory/filename.so

[my_likelihood_module]
file = modules/path/to/likelihood/filename.py
```

The file option must be the path to either a shared library (.so) or a python (.py) file. The paths that you need for CosmoSIS standard library modules are described in the reference section for them. See the documentation on making modules for more information on creating your own new modules.

In addition to this mandatory parameter, you can also specify other options in the file. These options can be read in the setup phase of the module:

```
[my_likelihood_module]
file = modules/path/to/likelihood/filename.py
data_file = some_path_to_a_data_file.dat
xxx = 1
```

Values Files

The values file defines the parameters that the sampler varies over, and other fixed parameters that are put into the likelihood pipeline at the start. All the values in the values file will be added to the data block before any modules are run.

The values file is required to run cosmosis. The path to it must be specified in the `[pipeline]` section of the main parameter file, using the syntax `values = path/to/values.ini`.

Like the other parameter files it is in the `ini` format. All parameters should be in named `[section]s`.

Fixed parameters

Fixed parameters are just given a single value, for example:

```
[cosmological_parameters]
w = -1.0
```

Fixed parameters can be integer or double types - if your code is expecting a double then make sure you don't write it as an integer - for example, writing the above example as `w=-1` would cause an error.

Varied parameters

Parameters that should be varied in the run are given three values, representing, respectively, the parameter lower limit, a typical starting value, and an upper limit, for example:

```
[cosmological_parameters]
omega_m  =  0.15    0.3    0.4
```

Different samplers use these ranges in different ways - for example, the test sampler ignores everything except the starting point, the metropolis sampler starts a chain at the start point and won't let it stray outside the limits, and the grid sampler ignores the starting value and generates a linear grid between the lower and upper limits. See the documentation on the samplers for more information.

Creating a parameter like this gives it an implicit uniform prior between the lower and upper limits (0.15 to 0.4). You can add additiona priors in the priors file.

Priors Files

The priors file is optional, and if desired can be set in the `[pipeline]` section of the main parameter file. The path to it may be specified in the `[pipeline]` section of the main parameter file, using the syntax `priors = path/to/values.ini`.

Like the other parameter files it is in the `ini` format. All parameters should be in named `[section]s`.

Implicit Priors

Parameters that appear in the values file always have an implicit uniform prior between the lower and upper limits that you specify there.

Priors that you specify in the priors ini file act as *additional* priors - for example, if you specify a Gaussian prior here for a parameter then the overall priors will be a Gaussian, truncated at the lower and upper bounds.

Additional Priors

Priors to include in addition to the implicit ones are set in the priors ini file. Here is an example:

```
[cosmological_parameters]
omega_m = uniform 0.0 0.2
h0 = gaussian 0.72 0.08
tau = exponential 0.05
```

As in the values file all parameters must be in a corresponding section. Priors that are set here but refer to parameters not mentioned in the values file will be ignored.

Currently only independent priors can be specified in CosmoSIS.

Uniform Priors

Uniform priors specified in the priors file can be used to further restrict parameters compared to their allowed range in the values file. Otherwise they are a little pointless.

They are specified in the form: `param_name = uniform <lower_limit> <upper_limit>`.

Gaussian Priors

Gaussian priors specified here combine with the implicit priors in the values section to form truncated Gaussian priors. The normalization is correctly adjusted.

They are specified in the form: `param_name = gaussian <mean> <std_dev>`.

Exponential Priors

Again, exponential priors are truncated by the limits in the values file.

The exponential distribution has the form $P(X = x) = \frac{1}{\beta} \exp(-x/\beta)$ for the parameter β.

They are specified in the form: `param_name = exponential <beta>`.

5.2.2 The ini format

All the CosmoSIS parameter files use the standard `ini` format, with a few additional features. In this format the file is divided into named sections surrounded by square brackets, and has named parameters in each section:

```
[section_name]
param1 = 123
param2 = 1.4e14
param3 = potato potato

[another_section_name]
param1 = abc
```

The parameter name and section name can be any ascii string. CosmoSIS will try to interpret the parameter value first as an integer, then it will fall back to a double, then as a boolean, and finally if all else fails it will assume it is a string. Different sections can store parameters with the same name; they are completely separate.

True / False values can be specified using "T"/"F" or "Y"/"N".

You can "re-open" a section later to specify more parameters if you want, for example:

```
[section1]
a = 1
b = 2

[section2]
x = 4.5

[section1]
b = 3
c = 10.
```

If you specify a parameter twice then the second one will overwrite the first one, so in the example above the "section1" parameter "b" will have the value 3.

5.2.3 Case

CosmoSIS ini file parameter names are CASE-INSENSITIVE. The parameter "x" is the same as "X". The entries can be case-sensitive depending how you write your modules.

5.2.4 Comments

Comments in ini files can be marked with a semi-colon `; anything after this is a comment.`

You can also use hashes `#`, but we recommend sticking with semi-colons for consistency.

5.2.5 Include statements

The first feature that CosmoSIS adds to the `ini` format is that it allows files to include other files, so that you can have nested parameter files. For example, you might have one basic parameter file and a number of small variants.

You can tell a parameter file to include another file like this:

```
%include /path/to/other_file.ini
```

This has the effect of "pasting" in the other file into the current one, so if you `%include` file "A" at the start of file "B" then file B's parameters will take precedence and any repeated options will overwrite "A", whereas if you include it at the end file A will take precedence.

The path is looked up relative to the current working directory, not to the first parameter file.

5.2.6 Environment variables

A second feature that CosmoSIS adds to ini files is the use of environment variables. You can use environment variables within parameter files with this syntax:

```
[my_likelihood]
data_file = ${DATAFILE}
```

Then if before you run CosmoSIS you write this in the (bash) terminal:

```
export DATAFILE=my_data_file.dat
```

Then it will replace DATAFILE with this value in the parameters.

If the environment variable `$DATAFILE` is not set when you run the code then no replacement will be done and your pipeline will look for a presumably non-existent file called `$DATAFILE`.

5.2.7 Interpolation

An in-built feature of ini files is called interpolation - parameters in the file can reference other parameters using this syntax:

```
[section]
name=xxxx
data_dir: %(name)s/data
```

Then the parameter `data_dir` will have the value `xxxx/data`. This only works within the same section, or using the default section described below.

5.2.8 The default section

If you include a section called `[DEFAULT]` then the code will fall back to that section if a particular option isn't found elsewhere. This can be particularly useful in combination with the interpolation feature described above.

For example:

```
[DEFAULT]
NAME = v1
```

(continues on next page)

(continued from previous page)

```
[output]
filename = %(NAME)s.txt

[my_module]
model=model-%(NAME)s
```

Would make the code use the value "v1" for the parameter "model" in the "my_module" section, and output the chain to "v1.txt" and use the parameter `model=model-v1` in the my_module section.

Looking up the parameter "NAME" in the "my_module" section would also give the value `v1`.

5.3 Command line flags for cosmosis

You can view command line flags for cosmosis using the command:

```
cosmosis --help
```

5.3.1 Parallelization

The two flags `--mpi` and `--smp` run the code in parallel.

The MPI flag runs the code under MPI, which means different processes are launched and communicate with each other. This requires the MPI runtime to be installed, and you must run cosmosis with the `mpirun` command first. This will depend on the details of your system, but usually a command like:

```
mpirun -n 4 cosmosis --mpi params.ini
```

will run CosmoSIS using 4 processes. All the parallel cosmosis samplers can run under MPI, and it can be used on large machines like supercomputers across nodes.

The SMP flag runs with the python multiprocessing module, which starts one process and then forks new ones. You would use it like this:

```
cosmosis --smp 4 params.ini
```

Most of the cosmosis samplers can use SMP, but not the Multinest or Polychord samplers. It can only run on single nodes of larger machines.

5.3.2 Debugging

CosmoSIS has two commands that help with debugging, `--pdb` and `--experimental-fault-handling`.

If your code fails and crashes while running any python module (not C, C++, or Fortran) then the `--pdb` flag will mean rather than just crashing the code will stop at the point of the error and you will enter the python debugger, PDB. You can read about how to use this debugger here: https://docs.python.org/3/library/pdb.html

The `--experimental-fault-handling` requires the python module `faulthandler` to be installed on your system, for example with `pip install faulthandler`. This command means that if your code crashes during a C, C++, or Fortran module you will get a traceback listing which functions were being run at the time.

5.3.3 Overriding Parameter Files

It can be useful to override parameters specified in the configuration files on the command line - this can let you launch a variety of different runs with the same file. The `-p` and `-v` flags let you override parameters in the main (params.ini) and values files respectively.

You can override any number of parameters in the main parameter file like this:

```
cosmosis params.ini -p section1.name1=value  section2.name2=value ...
```

For example, this command would change the sampler being used to emcee instead of its current value:

```
cosmosis params.ini -p runtime.sampler=test
```

The `-v` command is used exactly the same way but for the values file, for example, this would change one of the parameter ranges in demo 5:

```
cosmosis demos/demo5.ini  -v cosmological_parameters.omega_m="0.2 0.3 0.5"
```

Note the quotations marks above, which are needed when there are spaces in the parameter value.

5.4 Command line flags for postprocess

You can view command line flags for postprocess using the command:

```
postprocess --help
```

Get help

CHAPTER 6

FAQ

6.1 Why does the bootstrap installer download so many things?

The auto-installation script downloads a complete stack down to the compiler. We know this is a bit annoying and weird, and you can absolutely try and manage the dependency installation yourself, but we've found that with such a wide variety of codes collected together getting a consistent set of requirements can be a huge pain. It's very easy to end up with, e.g. one lapack in scipy and another from source that are incompatible.

So our default install downloads a few hundred MB of compilers and other infrastructure to avoid this.

6.2 When I run under MPI my output file comes out wrong with some lines too short and some repeated

To run cosmosis under MPI you also need to pass it the –mpi flag, not just use mpirun, e.g.:

```
mpirun -n 4 cosmosis --mpi params.ini
```

6.3 How do I create a new module?

https://bitbucket.org/joezuntz/cosmosis/wiki/modules

6.4 Where can I find a list of modules?

https://bitbucket.org/joezuntz/cosmosis/wiki/default_modules

6.5 How does my code call CosmoSIS?

It doesn't. CosmoSIS calls your code. Any physics or likelihood calculation becomes a module, an pipeline element that can plug into cosmosis, and from there into other modules.

6.6 CAMB says it's failing to find the parameter w but it's definitely there

You may need to change:

```
w = -1
```

to:

```
w = -1.0
```

6.7 When trying to run "xcode-select –install" on a mac I get the error "Can't download the software because of a network problem"

Please see the instructions on this page to install manually:

http://discuss.binaryage.com/t/aesepsis-1-4-1-issue-with-update-installing-xcode-command-line-tools/2013

6.8 I get the message "Abort trap 6" when I try to post-process on a Mac.

Try this, e.g. for demo one: DYLD_LIBRARY_PATH="" postprocess demos/demo1.ini

---

6.9 *matplotlib* still is not working for me. Can I make plots some other way?

If you encounter any trouble concerning *matplotlib*, please note that *matplotlib* is only used for producing plots, and is not required to run *CosmoSIS* itself. If you have, or can get, an installation of the *R* statistical environment, you can use it for generating plots as well. *R* can be obtained by following the instructions at the R project home page. Some additional *R* packages are needed. Please *source* the script *cosmosis/plotting/install-r-packages* to install them:

```
#!bash
$> source cosmosis/plotting/install-r-packages
```

6.10 I used git to update cosmosis and now things don't work

There are two repositories, cosmosis and the subdirectory cosmosis-standard-library. Make sure you update both and then do make clean and make afterwards.

6.11 I'm worried that CosmoSIS will change and then my code won't work any more

In general we do aim to develop in a backward compatible way. Consider writing your own regression tests for CosmoSIS i.e. tests that you or the cosmosis core developers can run that checks if everything is still working the way you want. We can't promise that we won't break your regression test, but at least we will be able to try to keep things working for you and if not then we can let you know.

6.12 I get an error with _thread in the "six" module on OSX

If you get an error like this:

```
/Library/Python/2.7/site-packages/dateutil/rrule.py in <module>()
    14
    15 from six import advance_iterator, integer_types
---> 16 from six.moves import _thread
    17
    18 __all__ = ["rrule", "rruleset", "rrulestr",

ImportError: cannot import name _thread
```

Then this is a general problem with the python-dateutil module. Fix it like this:

```
sudo pip uninstall python-dateutil
sudo pip install python-dateutil==2.2
```

6.13 I'm getting a GSL interpolation error when running cosmic shear analyses

Your n(z) needs to go all the way down to z=0, and no higher than zmax that you gave CAMB.

6.14 The axis labels look wrong - they have weird subscripts in the middle of words

You're using a parameter that cosmosis doesn't know the latex name for.

For example, say you used a new parameter "m_max" in a section called "galaxies".

Make a new file with the parameters names in called a new file something like `my-latex.ini`:

```
#!ini
[galaxies]
m_max = M_\mathrm{max}
```

Then run the postprocess command with the flag `--more-latex=my-latex.ini`

6.15 Can I use the CosmoSIS bootstrap with my forked repo?

Yes, you can do this by installing the normal cosmosis using the bootstrap code, but then change it to point instead to the new repository. E.g. use the bootstrap to install everything into a new directory called "my_cosmosis"

```
./cosmosis-bootstrap my_cosmosis
cd my_cosmosis
source config/setup-cosmosis
git remote set-url origin https://bitbucket.org/accountname/myforked_cosmosis
git pull
cd cosmosis-standard-library
git remote set-url origin https://bitbucket.org/accountname/myforked_cosmosis-
↪standard-library
git pull
cd ..
make
```

6.16 How can I customize my contour plot colors and line styles?

Use a "tweak", a set of commands which are run after the plotting is complete to customize one or more plots. Demo 8 has an explanation of tweaks in general. Here's a specific example for customizing a plot with two contours on.

Put this text in a file `contour_tweaks.py` and then run your postprocess command with the flag `--extra contour_tweaks.py`:

```
from cosmosis.postprocessing.plots import Tweaks
import pylab

class ModifyContours(Tweaks):
    #This could also be a list of files.  Just put the base part in here,
    #not the directory, prefix, or suffix.
    filename="2D_cosmological_parameters--omega_m_cosmological_parameters--h0"

    def run(self):
        ax = pylab.gca()
        #if you want to try this interactively you can add this:
        # from IPython import embed; embed() #press ctrl-D when finished playing
        #you don't unfortunately see the results until it finishes, I think, though␣
↪you could
        # try adding a pylab.show?

        #each set has two contours in it, inner and outer
        contour_set_1 = ax.collections[:2]
        contour_set_2 = ax.collections[2:4]

        #set the properties of the contour face and line
        for f in contour_set_1:
            f.set(linestyle=':', linewidth=3, facecolor='none', edgecolor='k',␣
↪alpha=1.0)

        #you could do the same for contour set 2, etc.,  here.
        #just remember that 2 will always be drawn on top of 1; you may
        #need to choose the order of chain files on the command line accordingly
```

6.17 How can I save a parameter that I marginalize over analytically, or generate in some other way

If you have an extra parameter that is derived from your chain, for example one marginalized analytically or derived from other parameters, you can save it in the output chains along with the sampled parameters

In the pipeline section of your parameter ini file, set:

```
[pipeline]
extra_output = section_name/param_name    section_name2/param_name2
```

This would save a parameter `param_name` that you write to the data block in the `section_name` section.

6.18 How can I check convergence of the emcee sampler

One quick check for convergence of emcee is to plot each parameter the chain as points. If it has converged then the various chains should all gradually diffuse out from the starting position and then all come to a similar deviation from the mean. If the chains all still have a gradual drift across the chain, for example if they are all still moving outwards by the end of the chain, then that indicates non-convergence.

If you'd like you can also use the acor module to test convergence as in emcee. Install acor using `pip install acor` and then you can use `acor.acor(data)` from python - you will need to reshape the chain to make it `nwalker * nsample` (or possibly the other way around!).

6.19 How can I improve emcee convergence

There is an alpha parameter for emcee, but we do not currently expose it because it does not usually help convergence. Instead the best way is usually to improve burn-in. If you can guess a good distribution of starting points for the chain (one per walker; for example, from an earlier chain, or guessing) then you can set `start_points` to the name of a file with columns being the parameters and rows being the different starting points.

6.20 What parameters does the cosmosis data block include

The data block does not include a fixed set of parameters. Instead it can contain anything you want to put into it. At the start of a pipeline (i.e. at the start of a single likelihood evaluation) it will contain just the parameters put into it from the values file; after each module is run more things will be added.

6.21 During the bootstrap, installation of matplotlib has failed

If you are on a Linux machine, the main cause of this failure is lack of some system software that is required to build matplotlib. The package most commonly missing is the freetype development headers and libraries. The solution is to have someone with system management privileges install the appropriate package. On a RHEL-type system, the installation command is:

```
yum install freetype-devel.x86_64
```

CHAPTER 7

Indices and tables

- genindex
- modindex
- search