CosmoSIS: Modular cosmological parameter estimation

CosmoSIS: modular cosmological parameter estimation

Joe Zuntz[a], Marc Paterno[b], Elise Jennings[c,d], Douglas Rudd[c,e], Alessandro Manzotti[c,f], Scott Dodelson[1,c,f], Sarah Bridle[a], Saba Sehrish[b], James Kowalkowski[b]

[a] *Jodrell Bank Centre for Astrophysics, University of Manchester, Manchester M13 9PL, U.K.*
[b] *Fermi National Accelerator Laboratory, Batavia, IL 60510-0500, U.S.A.*
[c] *Kavli Institute for Cosmological Physics, University of Chicago, Chicago, IL 60637, U.S.A*
[d] *Enrico Fermi Institute, University of Chicago, Chicago, IL 60637, U.S.A*
[e] *Research Computing Center, University of Chicago, Chicago, IL 60637, U.S.A.*
[f] *Department of Astronomy & Astrophysics, University of Chicago, Chicago, IL 60637, U.S.A.*

Abstract

Cosmological parameter estimation is entering a new era. Large collaborations need to coordinate high-stakes analyses using multiple methods; furthermore such analyses have grown in complexity due to sophisticated models of cosmology and systematic uncertainties. In this paper we argue that *modularity* is the key to addressing these challenges: calculations should be broken up into interchangeable modular units with inputs and outputs clearly defined. We present a new framework for cosmological parameter estimation, CosmoSIS, designed to connect together, share, and advance development of inference tools across the community. We describe the modules already available in CosmoSIS, including CAMB, Planck, cosmic shear calculations, and a suite of samplers. We illustrate it using demonstration code that you can run out-of-the-box with the installer available at `http://bitbucket.org/joezuntz/cosmosis`.

1. Introduction

Cosmological parameter estimation (CPE) is the last step in the analysis of most cosmological data sets. After completing all the acquisition, verification, and reduction of the data from an experiment, we transform compressed data sets, such as power spectra or brightnesses of supernovae, into constraints on cosmological model parameters by comparing them to theoretical predictions.

The standard practice in cosmology is to take a Bayesian approach to CPE. A likelihood function is used to assess the probability of the data that were actually observed given a proposed theory and values of that theory's parameters. Those parameters are varied within a chosen prior in a sampling process such as Markov Chain Monte-Carlo [1].

The result is a distribution that describes the posterior probability of the theory's parameters, often summarised by the best fit values and uncertainties on the parameters given the model and data.

A golden age of CPE has just ended; over the past decade the most powerful cosmological probes measured either the background expansion of the universe (like supernovae and baryon acoustic oscillations) or linear perturbations to it (like the microwave background or large-scale structure). These observations established a powerful concordance model of cosmology, ΛCDM, in which pressureless dark matter forms the seed and backbone of physical structures and a cosmological constant dominates expansion.

In the background and linear regimes inter-probe correlations were negligible, statistical errors dominated, and predictions were easy to make. The challenge for the next decade, for stage III cosmological surveys [2] and beyond, is to test the ΛCDM model with data from new regimes. This will be extremely difficult: on the non-linear scales we must probe, systematic effects require complex modelling, adding new physics becomes much harder, and different probes have subtle statistical correlations. In this paper we argue that this new era requires a new generation of parameter estimation tools.

A range of CPE tools exists today. The first and most widely used has been CosmoMC [3], a sophisticated Metropolis-Hastings sampler coupled closely to the CAMB [4] Boltzmann integrator. Other Boltzmann codes have had their own samplers attached, such as AnalyzeThis [5] for the CMBEASY [6] code and MontePython [7] for CLASS [8]. Additions to CosmoMC to perform new kinds of sampling have been made by codes like CosmoNest [9], and methods to interpolate or approximate likelihood spaces have included CMBFIT [10], CMBWarp [11], PICO [12], DASH [13], BAMBI [14] and SCOPE [15]. Other methods, like Fisher4Cast [16] and ICOSMO [17] have focused on the Fisher matrix approximation, particularly for forecasting results. More recently, CosmoHammer [18] has introduced cloud computing and a more pipeline-like approach to the genre, while CosmoPMC [19] has focused on late-time data with a new sampling algorithm, and CosmoLike

Email address: `joseph.zuntz@manchester.ac.uk` (Joe Zuntz)

[20] has made great strides in the high-accuracy calculation of the covariances and other statistics of late-time quantities. Codes like COSMOPP [21] and COSMOSLIK have moved towards an object-oriented or plug-in approach to building pipelines.

In this paper we present CosmoSIS, a new parameter estimation code with *modularity* at its heart, and discuss how this focus can help overcome the most pressing challenges facing the current and future generations of precision cosmology. The goal of CosmoSIS is *not* to produce new calculations of physical quantities, but to provide a better way to bring together and connect existing code. It does this with a plug-in architecture to connect multi-language *modules* performing distinct calculations, and provides a simple mechanism for building and extending physics, likelihood, and sampler libraries. CosmoSIS differs from previous parameter estimation codes in simultaneously emphasizing this modular approach, and allowing cosmologists to develop in their language of choice and thus leverage the large amount of powerful existing code in the community.

In Section 2 we discuss the challenges brought on by the new generation of data. We describe how modularity addresses many of these challenges in 3. We outline the structure of CosmoSIS in 4 and illustrate the CosmoSIS features with a number of examples in 5. We propose a model for future collaborative development of CosmoSIS in Section 6 and wrap up with a discussion of future prospects in 7. Guides for developers and users, and a worked example are included among the appendices.

2. Challenges

Several problems have conspired to end the pleasant period of CPE. Cosmological data sets now probe a non-linear, multi-probe regime where complex physical and analysis systematics are dominant. These systematics (such as photometric redshift errors or baryonic effects on the power spectrum) are correlated between probes: we must take care to consistently model their impact on different measurements and inferred statistics.

A richer model accounting for more physics will also require a large increase in the number of parameters. The Planck mission, for example, required about 20 nuisance parameters to account for physical and instrumental systematics [22]. This expanded parameter space carries with it computational costs, and the number of parameters and the associated costs will increase with future experiments.

Since systematics are both dominant and poorly understood, each analysis must be run with different models for each systematic to ensure that conclusions are robust and insensitive to model choices. Galaxy bias, for example, which describes the relative clustering of a sample of galaxies compared to the underlying matter distribution, can be described by a range of different models and parameterizations that are accurate to varying degrees over a given range of scales or galaxy types. A computational framework that does not allow these models to be simply replaced with alternatives can quickly become overwhelmingly complicated.

It is not only models of systematics that are getting more complicated. With the rich data expected from current and next generation experiments, we will be able to test a wide range of alternatives to vanilla ΛCDM, such as theories of modified gravity or dynamical dark energy (see [23] for a recent review). Many analyses and calibration methods assume ΛCDM throughout and can make switching to another cosmological model very difficult. Clarifying and making assumptions explicit is vital to correct work in these areas. Moreover, alternative models or parameterizations vary from analyst to analyst, and the most generic of them contains dozens of new cosmological parameters (for example, the effective Newton's constant as a function of scale and time that relates density to potential), all of which can be constrained.

Since all these complexities can make for a rather slow likelihood evaluation, more advanced sampling methods than the basic Metropolis-Hastings sampling are often considered. Making it as easy as possible to change and explore sampling methods is therefore a key goal. Of particular interest are those samplers designed to perform their calculations in parallel which can be used on modern multi-core and multi-node computing platforms.

Many of these problems have been tackled (in code) in a heterogeneous way by multiple authors, with multiple programming languages and in different ways. A useful CPE framework must make use of the large amount of existing code that was created to tackle different parts of the problems already discussed.

A final problem is social. Most cosmology collaborations are large and widely geographically spread, making cleanly sharing and comparing code and methods a significant challenge. There may be multiple approaches to treatment of systematics, multiple ideas for theoretical models to be tested, and multiple preferred computer languages. An easy way to communicate about code development maximizes collaboration between experts at different institutions.

We therefore have a slate of problems: correlated systematics, dimensionality, systematic models, variant cosmologies, advanced sampling, legacy code, bug finding, and diverse approaches; these inform our requirements for CosmoSIS.

3. Modularity

Modularity is the key to solving most of the problems listed above.

A modular (or *loosely coupled*) approach breaks up a larger complicated code into smaller parts. The philosophy is then that each module has a specific task to complete - it does one thing, and does it well, and its functionality does not depend directly on what other modules are used in the

pipeline - provided that the required inputs are present a module does not care which other code they came from.

The modules are only connected in a specific and limited way - the inputs they take and the outputs they make are passed on only through a specific set of functions designed for this purpose, rather than, for example, creating new global variables or structures to pass around. They do not have direct read and write access to the data each other hold.[1].

All data is then passed around via a single mechanism - the loading and saving of information in a single place (in this case the datablock; see Section 4).

A likelihood function then becomes a sequence of modular processes, run one-by-one to form a pipeline. The last module(s) generates the final likelihood numbers. Any module in the sequence can be replaced at runtime by another calculation of the same step without affecting the others. This independence and easy replacement of modules creates a flexible CPE framework where systematic models and alternative cosmologies can be fully explored.

As an example, consider a likelihood calculation for a spectroscopic survey's galaxy power spectrum $P(k, z)$ (see, for example, [24] [25]). We can split the physical calculation into:

- Compute the linear matter power spectrum $P(k, z)$ from cosmological parameters using a Boltzmann code.
- From this, compute the non-linear $P_{\rm NL}(k, z)$ with Halofit [26] or another model.
- Use bias parameters to calculate a bias model $b(k, z)$ for the galaxy sample.
- Compute the galaxy power spectrum as $P_g(k, z) = b^2(k, z)P_{\rm NL}(k, z)$.
- Integrate over survey window functions to compute predictions to compare with measurements.
- Compare the predicted to observed measurements to give a likelihood.

This process is illustrated in Figure 1.

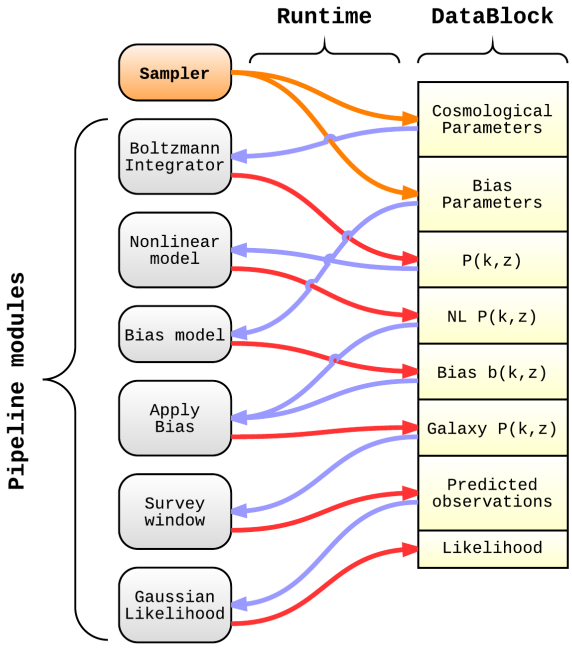

Figure 1: A schematic of the modular calculations for a galaxy power spectrum likelihood.

3.1. Benefits

There are many benefits to splitting a likelihood calculation into separate modules as shown in Figure 1:

Replacement. For many problems, including the first three modules shown in Figure 1, analysts have a choice of models with different parameterizations, each of which can be used to describe the particular physical process at each step. Making it easy to run different models without re-writing and recompiling code each time is easy in a modular architecture. In CosmoSIS a simple change to a configuration file (or even a command line option) suffices to switch between models.

Verifiability. It is far easier to test individual parts of a long pipeline than to regression-test the whole calculation. In CosmoSIS modules have their limitations and assumptions clearly specified, allowing analysts to create consistent pipelines that can be easily regenerated at a later date.

Debugging. Incorrect inputs and outputs or lack of clarity about the way pieces of code are supposed to connect accounts for a large fraction of software bugs. With the CosmoSIS architecture the inputs to a module are absolutely explicit and the connection between modules is clear.

Consistency. A treatment of shared physics and systematics that is consistent across probes is essential in order to obtain accurate constraints on cosmological parameters. Writing modules that read in the values they need from the shared CosmoSIS datablock rather than assuming them makes this problem explicit.

Languages. The CosmoSIS plug-in approach to adding modules makes it easy to switch between languages for different parts of the code. Complicated but fast portions can be written in python so they are easier to understand, and computationally intensive portions can be in a compiled language.

Legacy. A wide body of disorganized but powerful code already exists in cosmology. Wrapping parts of it as CosmoSIS modules allows the user to include it without hav-

[1] A modular design is the norm in many areas of software engineering; your web browser and operating system almost certainly take this approach.

ing to structure her own code around it.

Publishing. Modifications to a monolithic parameter estimation code such as COSMOMC, for example, are very unlikely to be compatible with each other or combine easily. For example, if one group creates a new data set with nuisance parameters while another makes a change to implement a theory of modified gravity, then combining those two alterations consistently is straightforward in the CosmoSIS structure.

Samplers. Splitting likelihood calculations into modules means we can create our entire CosmoSIS pipeline as a more easily reusable object. With a pipeline decoupled somewhat from the sampler, switching between samplers – and therefore studying which is optimal for a particular problem – becomes a far easier proposition. This is an important consideration as some samplers are ineffective at fully exploring multimodal distributions or parameter degeneracies.

3.2. Costs

A modular structure is not free; it imposes certain costs during both the design and execution of calculations.

Overheads. There is an overhead of code that must be written and run to connect each module to the system. For simpler modules this can be short, but for more complex modules with multiple options it can become more difficult. Being another layer of separation between parts of the pipeline it is also another place bugs can enter.

Interpolation. In a monolithic CPE architecture, functions in other parts of the code can be called freely; in this modular structure data must be explicitly saved by one module to be useable by another. This can mean that data is not sampled at the points that a module is expecting, and therefore require interpolation. This can be a source of inaccuracy. One mitigation is to define sample spacing in initialization files and check explicitly that the required accuracy is achieved, but this does place some burden on the user to perform validation tests.

Speed. The connections between modules can be (and are, in CosmoSIS) fast enough that they do not slow down cosmological likelihoods significantly. But short-cuts and efficiencies available in a tightly-coupled code may not be available in a modular context.

Consistency. Although a modular approach can help with consistency compared to a gradually accumulated codebase it is more vunerable to misuse compared to a rigid monolithic code that is designed from the start with consistency in mind. This can be particularly true in complex cases such as those where errors are cosmology-dependent. A key feature of CosmoSIS is that any pipeline output contains the runtime options and assumptions for each module used. This makes all the parameter values and cosmological model choices explicit and allows pipelines to be regenerated easily at a later date for verifiability or comparsion with collaborators. This feature limits the losses in moving away from a monolithic code and removes any ambiguity in the settings and assumptions used for a particular pipeline.

Temptation. As it becomes easier to specify and design a pipeline the temptation to over-complicate and build large and complex pipeline grows. The more parameters and steps a process has the more prone to error it is, and the more difficult the associated sampling problem becomes - larger spaces require more samples, longer burn-in, and make it harder to diagnose convergence. Having powerful pipeline tools must not become an excuse to avoid thinking about how to simplify a likelihood as much as possible.

Legacy. Most existing code is not written with modularity in mind. Much of it needs to be modified to fit into the CosmoSIS framework[2].

4. CosmoSIS structure

In this section we provide an overview of the structure of CosmoSIS and modules that link to it, and discuss the various samplers available in it in Section 4.4. More architectural details are available in Appendiex Appendix D.

4.1. Overview

In CosmoSIS a parameter estimation problem is represented by various components:

pipeline a sequence of calculations that computes a joint likelihood from a series of parameters.

modules the individual "pipes" in the pipeline, each of which performs a separate step in the calculation. Some do physics calculations, others interpolation, and at the end some generate likelihoods. Many modules are shipped with CosmoSIS as part of a standard library and users can easily write and include more.

datablock the object passed down the pipeline. For a given set of parameters all module inputs are read from the datablock and all module outputs are written to it.

sampler (generically) anything that generates sets of cosmological and other parameters to analyze. It puts the initial values for each parameter into the datablock.

runtime the code layer that connects the above components together, coordinates execution, and provides an output system that saves relevant results and configuration.

[2]See the CosmoSIS wiki for notes on importing legacy code. `https://bitbucket.org/joezuntz/cosmosis/wiki/modules`

The core CosmoSIS datablock is written in C++ and the runtime, samplers, and user interface are written in python. The latter was a clear choice: parsing user input and handling complex pipeline configuration and diverse other features is a field in which python excels. The choice of writing the core in C++ was driven first by speed requirements - we never want the runtime to be dominated by read/write - and second by the flexibility that modern C++ offers - it is easy to extend the datablock to include new data types. We also wanted a core that could be easily called all off Fortran, C, and Python, and this configuration offered an easy way to do that.

Modules can be written in C, C++, Fortran, or Python[3].

There are a number of technologies designed to connect C/C++ to python that we could have used to load and run modules, such as cython, boost-python, and swig, but we opted for a much simpler solution, the built-in *ctypes* modules, a very low-level interface into shared library functions. This was done for simplicity, clarity, and speed: ctypes is a very thin layer of abstraction, and so has minimal overhead and functions are called directly as defined. It places a little extra burden on the cosmosis developers when writing wrapper code, since undefined behavior can occur if mistakes are made, but the reduction in compile and thinking time is large.

In ctypes shared libraries are opened by name, and functions in them are extracted by name and manually assigned argument and return types. Once these assignments are made type checking is performed by python. This means the end user does not need to know anything about functions exposed by ctypes in order to use them.

4.2. Modules & Pipelines

The modularity that we advocate above is embodied in the splitting of the CosmoSIS likelihood function into a sequence of separate modules, each responsible for only a part of the calculation.

A module has two parts. The *main part* of the module code performs one or more calculations that go from physical input to output. The *interface* connects this to CosmoSIS by loading inputs from the datablock (see below) and saving outputs back to it. The interface is implemented either as a shared library or a python module.

Some modules exist to generate quantities for later modules to use - we refer to these as *physics modules*. Others use these values to produce data likelihoods - these are *likelihood modules*. Some can do both, and there is no structural difference between them. A sequence of modules connected together is referred to as a *pipeline*, and objects in the CosmoSIS runtime manage the creation and running of pipelines and modules.

[3]Generally the interface to modules could be easily extended to any language that can call and be called from C.

4.2.1. Examples

Camb [4] has been packaged as a CosmoSIS physics module. It loads cosmological parameters (and optionally a w(z) vector) from the datablock, and saves cosmic distance measurements and various linear power spectra. The Planck likelihood code [27] has been packaged as a likelihood module - it reads CMB spectra from the datablock, and saves likelihoods. A very simple CosmoSIS pipeline could just combine these two modules. We could substitute another Boltzmann code for camb, such as class [8], with no changes at all to the Planck module, and compare the two just by changing a single line in a configuration file, with no recompiling.

4.3. DataBlocks

We enforce modularity in CosmoSIS by requiring that all information to be used by later modules is stored in a single place, which we call a *DataBlock*. Storing all the cosmology information in one places makes it easier to serialize blocks. It also makes debugging easier because all the inputs that a given module receive are explicitly clear.

The datablock is a CosmoSIS object that stores scalar, vector, and n-dimensional integer, double, or complex data, as well as strings. It is the object that is passed through the pipeline and contains all the physical cosmological information that is needed by the pipeline, such as cosmic distances, power spectra, correlation functions, or finally likelihoods.

DataBlocks explicitly *cannot* store arbitrary structured information; wanting to do so would suggest that modularity is broken, since genuinely physical information is typically fairly unstructured. If complicated data structures are passed among code it would imply that code should be a single module. A good guideline is that data stored in blocks should have physical meaning independently of the particular code that generates or uses it.

More details may be found in Appendix Appendix D.

4.4. Samplers

We think of a "sampler" in very abstract terms, and do not limit ourselves to a Markov chain Monte Carlo (MCMC). A sampler is anything that produces one or more sets of input parameters, in any way, runs the pipeline on them, and then does something with the results. Some samplers then iterate this process. MCMC and maximum-likelihood samplers, for example, use previous parameter values to choose more samples, unlike a grid sampler, which decides them all in advance.

The most trivial possible sampler is implemented as the `test` sampler in CosmoSIS: it generates a single set of input parameters and runs the pipeline on that one set, saving the results.

For samplers that can work in parallel, like grid sampling or emcee, we provide a parallel "Pool" sampler architecture implemented with process-level parallelism using Message-Passing Interface (MPI). Thread parallelism

at the sampler level is not possible becase many key cosmology codes (like CAMB) are not thread-safe. Thread parallelism within modules is supported; for example using OpenMP.

The following samplers are available in CosmoSIS. The details of how to call each sampler in a pipeline are given in section Appendix A.5.

1. The `test` sampler evaluates the CosmoSIS pipeline at a single point in parameter space and is useful for ensuring that the pipeline has been properly configured. The `test` sampler is particularly useful for generating predictions for theoretical models, outside the context of parameter estimation.
2. The `grid` sampler is used to sample the CosmoSIS parameters in a regularly spaced set of points, or grid. This is an efficient way to explore the likelihood functions and gather basic statistics, particularly when only a few parameters are varied. When the number of parameters is large, the number of sampled points in each dimension must necessarily be kept small. This can be mitigated somewhat if the grid is restricted to parameter ranges of interest.
3. The `maxlike` sampler is a wrapper around the `SciPy minimize` optimization routine, which is by default an implementation of the Neader-Mead downhill simplex algorithm.
4. The `metropolis` sampler implements a straightforward Metropolis-Hastings algorithm with a proposal similar to the one in COSMOMC, using a multivariate Gaussian. Multiple chains can be run with MPI.
5. The `emcee` [28] sampler[4] (Daniel Foreman-Mackey, David W. Hogg, Dustin Lang, Jonathan Goodman) is a python implementation of an affine invariant MCMC ensemble sampler [29]. The `emcee` sampler simultaneously evolves an ensemble of "walkers" where the proposal distribution of one walker is updated based on the position of all other walkers in a complementary ensemble. The number of walkers specified in the CosmoSIS ini file must be even to allow a parallel stretch move where the ensemble is split into two sets (see [28]). The output will be (walkers $\times$ samples) number of steps for each parameter.
6. The `multinest` [30] sampler[5], a multi-modal nested sampler that integrates the likelihood throughout the prior range of the space using a collection of live points and a sophisticated proposal to sample in an ellipsoid containing them. It produces the Bayesian evidence in addition to samples from the posterior.

A discussion of the comparative advantanges of nested, emcee, and metropolis sampling can be found in [31].

[4] `http://dan.iel.fm/emcee/current/`
[5] `http://ccpforge.cse.rl.ac.uk/gf/project/multinest/`

4.5. User Interface

The primary user interface in CosmoSIS is configuration files in the "ini" format, extended slightly beyond the standard to allow the inclusion of other files. The ini file is converted into a DataBlock object to initialize modules. For convenience all ini file parameters can be overridden at the command line.

5. Examples

CosmoSIS ships with a selection of demos that illustrate its features. In this section we briefly overview them.

5.1. Example One: basic cosmology functions

The first CosmoSIS demo is a simple illustration of a very basic pipeline, which produces no likelihoods and just saves some cosmological results. The code can be run with the command `cosmosis demos/demo1.txt` and analyzed with `postprocess demos/demo1.txt` to produce plots like Figure 2.

The pipeline run has just two modules - CAMB and HALOFIT, and the results, which are saved into a newly created directory, illustrate the outputs that we extract from the two of them.

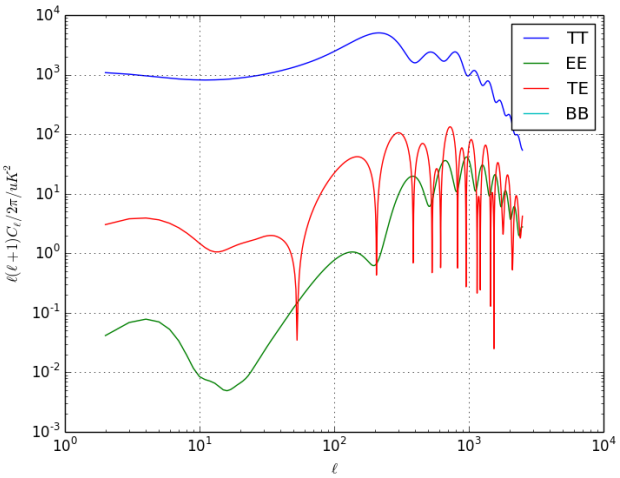

Figure 2: CosmoSIS demo one output plot, showing CMB spectra output from CAMB. CMB spectra and a host of other cosmology theory values are saved from the CAMB CosmoSIS module for later pipeline modules to use.

5.2. Example Two: Planck & BICEP2 likelihoods

In demo two we modify demo one by adding real likelihoods to the end - for Planck and BICEP2 [32]. Our pipeline is now: CAMB-PLANCK-BICEP2, and the code can be run with `cosmosis demos/demo2.txt`. The Planck data files are required for this demo to work.

This time our single-sample test sampler reports some output likelihood values for the pipeline.

5.3. Example Three: BICEP2 likelihood slice

In demo three we use our first non-trivial sampler: we take a line sample through the BICEP2 likelihood in the primordial tensor to scalar ratio r. All we must do to switch to the grid sampler is change the `sampler` setting in the configuration file to `grid` and tell it how many sample points to use.

Run this example with `cosmosis demos/demo3.txt` and the results in Figure 3 are produced with `postprocess demos/demo3.txt`, along with constraints on the r parameter.

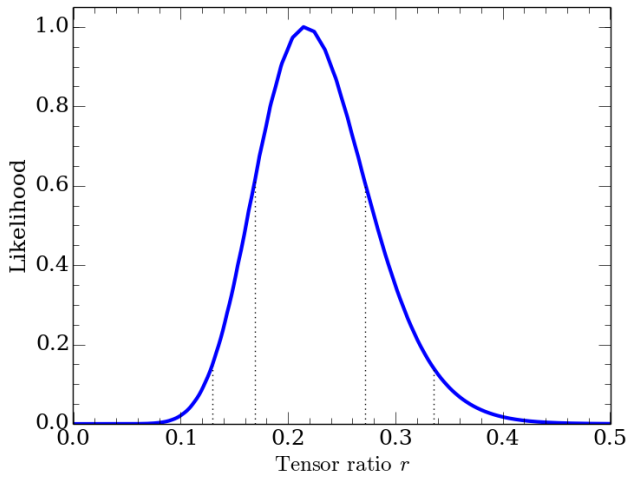

Figure 3: CosmoSIS demo 3 output plot, showing the constraints on the primordial tensor fraction r from BICEP2 B-mode data. The y-axis is the normalized likelihood and the vertical lines show 68% and 95% contours.

5.4. Example Four: Maximum-likelihood Planck

The fourth CosmoSIS demo uses a numerical optimizer to find the maximum likelihood parameter set for a set of Planck likelihoods. Run it with `cosmosis demos/demo4.txt`.

The sampler uses the Nelder-Mead method [33] to find the peak (though various other methods can be chosen in the ini file).

The best-fitting parameters are reported at the end, and since we often use max-like samplers to find a starting point for Monte-Carlo samplers, the CosmoSIS max-like sampler also outputs an ini file you can use for this purpose. In fact demo five below starts using an ini file generated like this.

5.5. Example Five: mcmc'ing JLA supernovae

Demo number five brings us to geniune MCMC sampling, using the EMCEE sampler. In this pipeline we configure CAMB to run only background quantities like $D_A(z)$, and then use a JLA likelihood module [34] to sample with. We include supernova light-curve nuisance parameters.

The post-process plotting code with CosmoSIS generates plots from MCMCs like the one in Figure 4 using kernel density estimation to smooth the samples, with a correction to ensure the right number of samples are under the 68% and 95% contours (see appendix Appendix F).

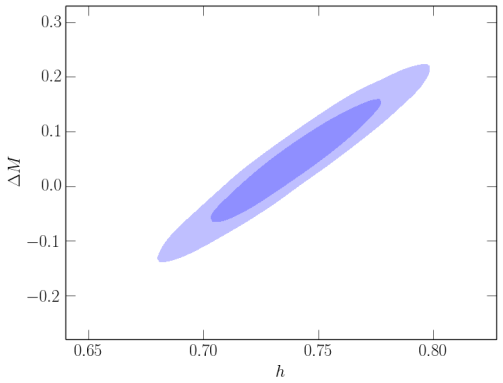

Figure 4: An example CosmoSIS constraint on the JLA supernova data set - all the 1D and 2D constraint plots are generated by CosmoSIS; this example (CosmoSIS demo 5) shows constraints from the JLA SDSS supernova sample on the Hubble parameter h and the supernova magnitude parameter ΔM made using the `emcee` sampler.

5.6. Example Six: CFHTLenS; a longer pipeline

CFHTLenS is an example of a more complex likelihood pipeline of the type that will be the norm in the coming decade. This pipeline is discussed in depth in Appendix Appendix C; briefly, it has six different modules: CAMB - HALOFIT - NUMBER-DENSITY - SHEAR-SPECTRA - SHEAR-CORRELATIONS - LIKELIHOOD.

Since this is quite a slow process we just run the test sampler for this demo, and produce (among other results) the plot in Figure 5. For sampling, the parallel capabilities of the EMCEE and MULTINEST samplers are invaluable for reasonable run times.

5.7. Example Seven: 2D grid sampling BOSS DR9

The grid sampler from example three can grid in arbitrary dimensions[6], and in parallel if required. In that example we used just a single dimension for a slice sample; in this example we run a 2D grid over BOSS [35] constraints on the growth rate and σ_8, wrapping a CosmoSIS standard library module that can calculate the linear growth rate as a function of redshift and for dynamical dark energy, $w(z)$, cosmologies.

The CosmoSIS post-process program reads the same ini file used to run the sampling, and thus knows automatically that our output files are grids and that grid plots rather than MCMC constraint plots should be made. The 2D plot for this example is shown in Figure 6. If we had sampled over other parameters they would be automatically marginalized over.

[6] though above about 4 dimensions this becomes unfeasible, since the number of samples $= (n_{\rm grid})^{n_{\rm dim}}$

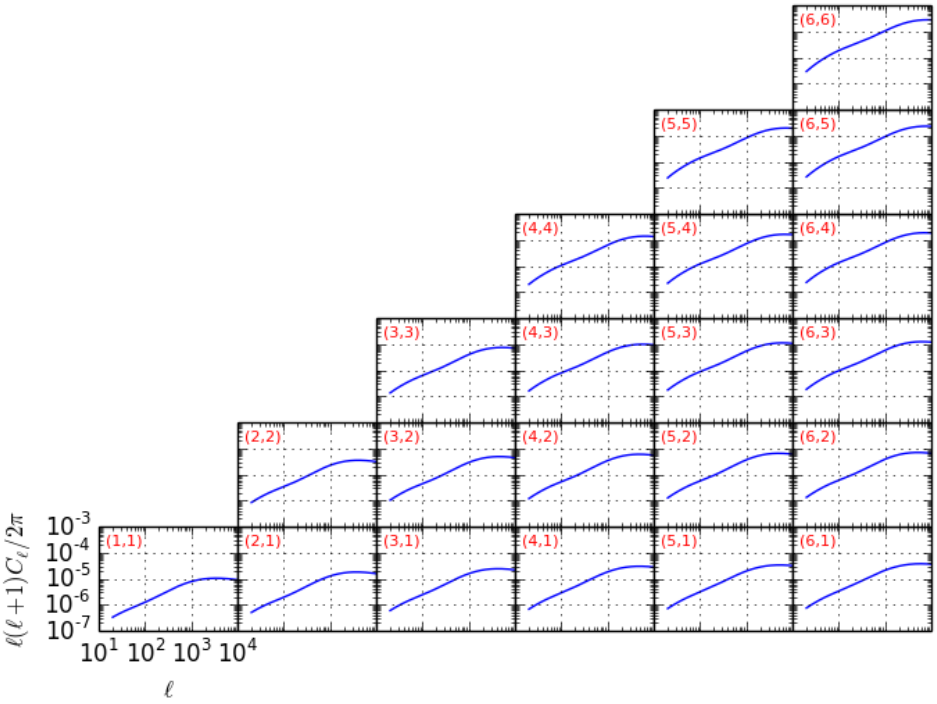

Figure 5: The CosmoSIS test sampler produces and saves all the cosmological outputs for a set of parameters, and they can immediately be plotted with the postprocess program. This example from CosmoSIS demo six shows the cosmic shear spectra generated for the CFHTLenS redshift bins.

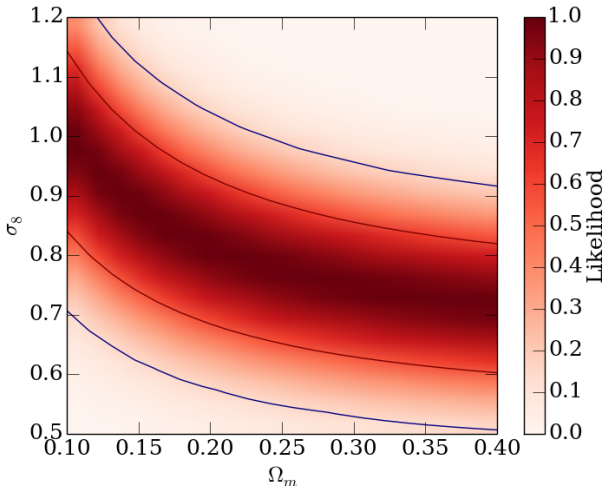

Figure 6: For smaller parameter spaces a grid sampler like the one shown here may be more suitable than an MCMC. Grid constraints can also be immediately plotted by the CosmoSIS postprocessor. This example (from demo seven) shows contraints on Ω_m and σ_8 from BOSS measurements of $f\sigma_8$.

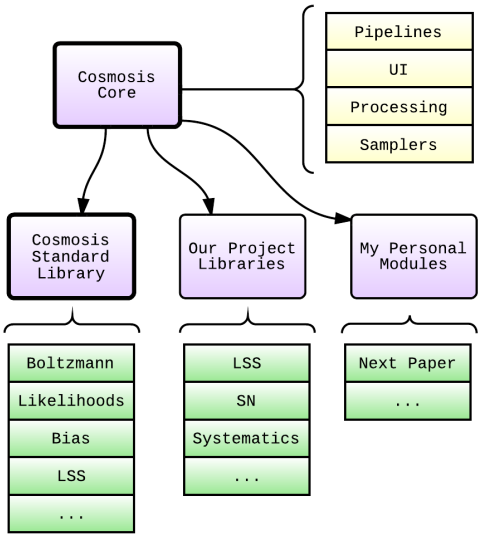

Figure 7: The structure of CosmoSIS. Purple boxes are repositories, in bold if they are part of the CosmoSIS package; the others are per-user. Green boxes contain (collections of) modules.

6. Sharing CosmoSIS modules

CosmoSIS comes with a collection of generally useful modules for common cosmological inference problems. We refer to this as the CosmoSIS standard library (CSL), and it includes the Boltzmann code camb; likelihoods like Planck and CFHTLenS; some common mass functions from the Komatsu Cosmology Routine Library[7] adapted into modules; bias parameterizations; source count number densities; and various other calculations.

Collaborations, projects, and individuals can easily create their own libraries of CosmoSIS modules to be used alongside or in place of the CSL. These might be used to perform calculations specific to a particular theory or experiment, or for a particular paper. They might augment the behaviour of CSL pipelines, for example by implementing a new systematic error effect, or replace standard behaviour, such as using a new improved mass function instead of a standard one. CSL is a sub-directory of the CosmoSIS main directory, as are any other libraries used by collaborations and individuals.

There is exactly one sensible way to organize collections of modules: in version-controlled repositories. The repositories used in CosmoSIS are described in this section and depicted in Fig. 7.

6.1. Module repositories

The CSL is stored in a publically accessible version control repository[8]. Such repositories store the code and a record of all the changes and additions made to it; a directory tree on disc corresponds to a remotely stored repository, and can be kept in sync with it.

Repositories are a convenient way of storing, managing, and sharing code, and have features for reviewing and accepting third-party contributions. A typical work pattern is to keep a repository private initially until a paper is released, and then make the module public alongside it.

Repositories can be written by and made accessible to individual collaborators, wider teams, or the community at large.

6.1.1. Creating a new module repository

CosmoSIS comes with a script to automate the process of creating new repositories for (groups of) modules that you want to manage or share. Run the script using `cosmosis/tools/create-repository.py --help` for information on using it.

6.2. Contributing to the standard library

The CSL is not immutable and we very strongly welcome contributions of modules of any sort so that the cosmology community can make use of code from different groups and experiments more easily.

We will gladly distribute any module with the CSL as long as:

- the code authors will give permission for us to distribute and if necessary modify it.
- any data included with it can be released publically.
- it will be of general use to the community.

[7] `http://www.mpa-garching.mpg.de/~komatsu/crl/`

[8] `https://bitbucket.org/joezuntz/cosmosis-standard-library`

- it meets accuracy, quality, and documentation standards.

Since most cosmologists are not trained programmers we do not enforce any specific coding standard or technology such as unit testing, but we strongly encourage contributors to write tests along with their code to ensure its functionality, and provide a mechanism within CosmoSIS to run all those tests on all modules.

Modules can be documented using the human- and machine-readable YAML format; a short YAML file included with each module describes its name, authorship, purpose, assumptions, inputs, and outputs.

7. Discussion

The core claim we make in this paper is that a modular approach is useful and perhaps vital if cosmological parameter estimation is to remain accessible across the cosmology community. We have presented CosmoSIS, a code that embodies a modular architecture. It is a freely available, flexible and extendable tool for the use of observers, analysts, and theorists alike.

While the CosmoSIS standard library contains a range of pre-existing and new modules for parameter estimation problems, its true value is its extensibility. We strongly encourage and welcome contributions of existing or new code wrapped as a module[9] and are happy to assist in any way[10].

To take your next steps with CosmoSIS, you can download and install the code and try the examples. After this there are further instructions in Appendices Appendix A and Appendix B on going further as a CosmoSIS user and developer.

Acknowledgements

We are grateful for help with beta testing the code to Youngsoo Park, Michael Schneider, Niall Maccrann, Matt Becker, Mustapha Ishak, Tim Eifler, Katarina Markovic, Marcelle Soares-Santos, Eric Huff, Tomasz Kacprzak. We also thank members of the Dark Energy Survey Theory and Combined Probes Science Working Group including Ana Salvador, Marisa March, Elisabeth Krause and Flavia Sobreira. We are grateful to the attendees of the Chicago CosmoSIS May 2014 workshop and the LSST DESC CosmoSIS Philadelphia June 2014 workshop for useful feedback.

JZ and SB acknowledge support from the European Research Council in the form of a Starting Grant with number 240672.

[9] See the wiki for guidance: `https://bitbucket.org/joezuntz/cosmosis/wiki/modules`

[10] You can either email us or open an issue on the CosmoSIS repository: `https://bitbucket.org/joezuntz/cosmosis/issues/new`

SD is supported by the U.S. Department of Energy, including grant DE-FG02-95ER40896. DOE HEP Computing supported development of CosmoSIS. EJ acknowledges the support of a grant from the Simons Foundation, award number 184549. This work was supported in part by the Kavli Institute for Cosmological Physics at the University of Chicago through grants NSF PHY-0114422 and NSF PHY-0551142 and an endowment from the Kavli Foundation and its founder Fred Kavli. We are grateful for the support of the University of Chicago Research Computing Center for assistance with the calculations carried out in this work.

This work was supported in part by National Science Foundation Grant No. PHYS-1066293 and the hospitality of the Aspen Center for Physics.

[1] N. Christensen, R. Meyer, L. Knox, B. Luey, Bayesian methods for cosmological parameter estimation from cosmic microwave background measurements, Classical and Quantum Gravity 18 (2001) 2677–2688. arXiv:astro-ph/0103134, doi:10.1088/0264-9381/18/14/306.

[2] A. Albrecht, et al., Report of the Dark Energy Task Force, ArXiv Astrophysics e-printsarXiv:astro-ph/0609591.

[3] A. Lewis, S. Bridle, Cosmological parameters from CMB and other data: A Monte Carlo approach, Physical Review D 66 (1) (2002) 103511.

[4] A. Lewis, A. Challinor, A. Lasenby, Efficient Computation of Cosmic Microwave Background Anisotropies in Closed Friedmann-Robertson-Walker Models, The Astrophysical Journal 538 (2) (2000) 473–476.

[5] M. Doran, C. M. Müller, Analyse this! A cosmological constraint package for CMBEASY, Journal of Cosmology and Astroparticle Physics 09 (0) (2004) 003–003.

[6] M. Doran, CMBEASY: an object oriented code for the cosmic microwave background, Journal of Cosmology and Astroparticle Physics 10 (1) (2005) 011–011.

[7] B. Audren, J. Lesgourgues, K. Benabed, S. Prunet, Monte Python: Monte Carlo code for CLASS in Python, Astrophysics Source Code Library (2013) 07002.

[8] J. Lesgourgues, The Cosmic Linear Anisotropy Solving System (CLASS) I: Overview, arXiv.org (2011) 2932arXiv:1104.2932.

[9] D. Parkinson, P. Mukherjee, A. Liddle, CosmoNest: Cosmological Nested Sampling, Astrophysics Source Code Library (2011) 10019.

[10] H. B. Sandvik, M. Tegmark, X. Wang, M. Zaldarriaga, CMBFIT: Rapid WMAP likelihood calculations with normal parameters, Physical Review D 69 (6) (2004) 63005.

[11] R. Jimenez, L. Verde, H. Peiris, A. Kosowsky, Fast cosmological parameter estimation from microwave background temperature and polarization power spectra, Physical Review D 70 (2) (2004) 23005.

[12] W. A. Fendt, B. D. Wandelt, Pico: Parameters for the Impatient Cosmologist, The Astrophysical Journal 654 (1) (2007) 2–11.

[13] M. Kaplinghat, L. Knox, C. Skordis, Rapid Calculation of Theoretical Cosmic Microwave Background Angular Power Spectra, The Astrophysical Journal 578 (2) (2002) 665–674.

[14] P. Graff, F. Feroz, M. P. Hobson, A. Lasenby, BAMBI: blind accelerated multimodal Bayesian inference, Monthly Notices of the Royal Astronomical Society 421 (1) (2012) 169–180.

[15] S. Das, T. Souradeep, SCoPE: An efficient method of Cosmological Parameter Estimation, arXiv.org (2014) 1271arXiv:1403.1271.

[16] B. A. Bassett, Y. Fantaye, R. Hlozek, J. Kotze, Fisher Matrix Preloaded — FISHER4CAST, International Journal of Modern Physics D 20 (1) (2011) 2559–2598.

[17] A. Refregier, A. Amara, T. D. Kitching, A. Rassat, iCosmo: an interactive cosmology package, Astronomy & Astrophysics 528

(2011) 33.
[18] J. Akeret, S. Seehars, A. Amara, A. Refregier, A. Csillaghy, CosmoHammer: Cosmological parameter estimation with the MCMC Hammer, Astronomy and Computing 2 (2013) 27–39.
[19] M. Kilbinger, et al., CosmoPMC: Cosmology Population Monte Carlo, arXiv.org (2011) 950arXiv:1101.0950.
[20] T. Eifler, E. Krause, P. Schneider, K. Honscheid, Combining probes of large-scale structure with COSMOLIKE, Monthly Notices of the Royal Astronomical Society 440 (2) (2014) 1379–1390.
[21] G. Aslanyan, Cosmo++: An Object-Oriented C++ Library for Cosmology, ArXiv e-printsarXiv:1312.4961.
[22] Planck Collaboration, Planck 2013 results. XVI. Cosmological parameters, arXiv.org (2013) 5076arXiv:1303.5076.
[23] A. Joyce, B. Jain, J. Khoury, M. Trodden, Beyond the Cosmological Standard Model, arXiv.org (2014) 59arXiv:1407.0059.
[24] D. Parkinson, et al., The WiggleZ Dark Energy Survey: Final data release and cosmological results, Physical Review D 86 (1) (2012) 103518.
[25] B. A. Reid, et al., Cosmological constraints from the clustering of the Sloan Digital Sky Survey DR7 luminous red galaxies, Monthly Notices of the Royal Astronomical Society 404 (1) (2010) 60–85.
[26] R. Smith, et al., Stable clustering, the halo model and nonlinear cosmological power spectra, Mon.Not.Roy.Astron.Soc. 341 (2003) 1311. arXiv:astro-ph/0207664.
[27] Planck Collaboration, Planck 2013 results. XV. CMB power spectra and likelihood, arXiv.org (2013) 5075arXiv:1303.5075.
[28] D. Foreman-Mackey, D. W. Hogg, D. Lang, J. Goodman, emcee: The MCMC Hammer, Publications of the Astronomical Society of the Pacific 125 (925) (2013) 306–312.
[29] J. Goodman, J. Weare, Ensemble samplers with affine invariance, ... in Applied Mathematics and Computational Science.
[30] F. Feroz, M. P. Hobson, M. Bridges, MULTINEST: an efficient and robust Bayesian inference tool for cosmology and particle physics, Monthly Notices of the Royal Astronomical Society 398 (4) (2009) 1601–1614.
[31] R. Allison, J. Dunkley, Comparison of sampling techniques for Bayesian parameter estimation, Monthly Notices of the Royal Astronomical Society 437 (4) (2014) 3918–3928.
[32] P. Ade, et al., Detection of B-Mode Polarization at Degree Angular Scales by BICEP2, Phys.Rev.Lett. 112 (2014) 241101. arXiv:1403.3985.
[33] J. A. Nelder, R. Mead, A Simplex Method for Function Minimization, The computer journal 7 (4) (1965) 308–313.
[34] M. Betoule, J. Guy, R. Kessler, J. Mosher, P. Astier, R. Biswas, P. El Hage, D. Hardin, J. Marriner, R. Pain, N. Regnault, Improved cosmological constraints from a joint analysis of the SNLS and SDSS surveys, American Astronomical Society 223.
[35] C.-H. Chuang, et al., The clustering of galaxies in the SDSS-III Baryon Oscillation Spectroscopic Survey: single-probe measurements and the strong power of f(z)σ8(z) on constraining dark energy, Monthly Notices of the Royal Astronomical Society 433 (4) (2013) 3559–3571.
[36] C. Heymans, et al., CFHTLenS tomographic weak lensing cosmological parameter constraints: Mitigating the impact of intrinsic galaxy alignments, Monthly Notices of the Royal Astronomical Society 432 (3) (2013) 2433–2453.

Appendix A. CosmoSIS user's guide

To run CosmoSIS you:

1. Choose a sequence of modules to form the pipeline.
2. Create a parameter file describing that pipeline.
3. Create a values file describing the numerical inputs for the parameters or their sampling ranges.
4. Check that your pipeline can run using CosmoSIS with the `test` sampler.
5. Choose and configure a sampler, such as `grid`, `maxlike`, `multinest`, or `emcee`.
6. Run CosmoSIS with that sampler
7. Run the `postprocess` command on the output to generate constraint plots and statistics.

In this section we describe each of these steps in more detail.

Appendix A.1. Choosing the pipeline

Choosing a pipeline means deciding what cosmological analysis you wish to perform, and then selecting (or writing) a sequence of modules that realize that analysis. This means asking the questions: what data do I want to use for my constraints? What theory predictions do I need to compare to that data? What calculations and parameters do I need to get that theory calculation? And what modules do these calculations and likelihood comparisons?

If you are using only one likelihood these questions are usually quite easy, but if there is more than one some more thought is required. For example, if you are using only weak lensing data then you can sample directly over the σ_8 parameter, whereas if you wish to combine with CMB data you need to start with A_s and derive σ_8 from it.

Typically, the pipeline will produce a likelihood, but one use of CosmoSIS is to generate the theory predictions for observables for a discrete set of parameters. Working within the framework of CosmoSIS enables the user to exploit tools such as CAMB to generate predictions.

Every module has a set of inputs and outputs, and for a pipeline to be valid every input that a module requires must be provided, either by an earlier module in the pipeline, or by the initial settings of the sampler. Two modules must also not try to provide the same output, unless one is explicity over-writing the other, since this would imply two different and probably inconsistent methods for calculating the same thing.

Often you can write a prototypical pipeline without including various systematic errors or other steps, and then add these as new modules in the middle of the pipeline as your analysis develops. For example, in weak lensing we must include the effect of error in our shape measurement by scaling the predicted spectra. For an initial analysis, though, this can be left out. A module performing the scaling can be inserted later.

Appendix A.2. Defining the pipeline

Once you have done the hard part and decided on a pipeline then you tell CosmoSIS what you have chosen. A CosmoSIS pipeline is described in the main parameter configuration file. Some example demos are included with CosmoSIS, and modifying one of those is a good place to start.

A section in the ini file tells CosmoSIS what modules make up your pipeline, where to find values to use as inputs to it, and what likelihoods to expect at the end. Here is

an example from demo six, which analyzes tomographic CFHTLenS data [36]:

```
[pipeline]
modules = camb halofit load_nz shear_shear 2pt
    cfhtlens
values = demos/values6.ini
likelihoods = cfhtlens
```

The `modules` parameter gives an ordered list of the modules to be run. The `values` parameter points to a file discussed in the next section. And the `likelihoods` tells CosmoSIS what likelihoods the pipeline should produce - because `cfhtlens` is listed in this case a module in the pipeline is expected to produce a scalar double value in the `likelihood` section called `cfhtlens-like` (actually the log-likelihood). Other values in the likelihood section created by the pipeline will not automatically be included in the likelihood value for the acceptance criterion - this can be useful for importance sampling, for example.

Each entry in the `modules` list refers to another section in the same ini file, which tells CosmoSIS where to find the module and how to configure it. For example, here is the section in demo six for the first module in the pipeline, CAMB:

```
[camb]
file = cosmosis-standard-library/boltzmann/camb/camb.
    so
mode=all
lmax=2500
feedback=0
```

The `file` parameter is mandatory for all modules, and describes the shared library or python module interface code. The other parameters are specific to CAMB, and are passed to it when the module is initialized. For example, the `lmax` parameter defines the maximum ℓ value to which the CMB should be calculated[11].

Parameter files can be "nested": the ini file that you run CosmoSIS on can use the syntax `%include other_params.ini` to mean that all the parameters defined in `other_params.ini` should also be used. This is particularly useful for running a number of similar chains with minor differences.

Appendix A.3. Defining parameters and ranges

In the last section we defined the pipeline and its expected outputs; in this section we define the inputs. An entry in the `[pipeline]` section of the main ini file described above was `values = demos/values6.ini`. This file specifies the parameter values and ranges that will be sampled. For example, in CosmoSIS demo four, which runs a maximum-likelihood sampler on Planck data [27], this file starts with:

[11]The parameters used by modules in the standard library of CosmoSIS are described at `https://bitbucket.org/joezuntz/cosmosis/wiki/default_modules`

```
[cosmological_parameters]
omega_m = 0.2 0.3 0.4
h0 = 0.6 0.7 0.8
omega_b = 0.02 0.04 0.06
A_s = 2.0e-9 2.1e-9 2.3e-9
n_s = 0.92 0.96 1.0
tau = 0.08
omega_k = 0.0
w = -1.0
wa = 0.0

[planck]
A_ps_100 = 152
A_ps_143 = 63.3
A_ps_217 = 117.0
A_cib_143 = 5.0
; ...
```

The values file is divided into sections, in this case two of them, `cosmological_parameters` and `planck`, reflecting different types of information that are stored in the datablock. Any module can access parameters from any section. There is no pre-defined list or number of inputs; if more are required by some modules they can be freely added.

Some of the values in the file are given a single value, such as the Planck parameters and Ω_k. That indicates that for this analysis the sampler should not vary these parameters, but leave them as fixed values. Others, like Ω_m, have a lower limit, starting value, and upper limit specified. These specifies the range of permitted values for the parameter and specifies an implicit flat prior.

Different samplers use the starting and limit values differently. The `test` sampler ignores the limits and just uses the starting value to generate a single sample. MCMC samplers reject proposed samples outside the range and initalize the chains at the starting value. And the `grid` sampler uses them to specify the range of points that should be gridded.

Appendix A.4. Test the pipeline

As noted in Section 4.4, the simplest sampler is one that provides just a single sample. After building a pipeline the next step is to test it with this trivial sampler. In the `[runtime]` section of the parameter file we set the sampler to `[test]`, and then in the `[test]` section we configure it:

```
[runtime]
sampler = test

[test]
save_dir=demo_output_1
fatal_errors=T
```

In this example we have asked the sampler to save all the data generated by all the modules in a directory structure. This is an excellent way to check whether a pipeline is working - all the important data in the pipeline can be compared to what is expected.

Most pipelines, like all codes, will not work the first time they are run! The test sampler also includes options to track down causes of errors, and to time code. For convenience we also supply a simple (and easily extensible) program to plot many of the standard cosmological observables that are saved by the pipeline, to aid debugging. An example of one such plot from CosmoSIS demo six, which generates CFHTLenS likelihoods [36], is shown in Figure 5.

Appendix A.5. Choosing a sampler

Different samplers produce results that are useful in different regimes.

The `grid` sampler has a number of advantages - it is straightforward to post-process, and there is no question of convergence. It is not however, feasible to use it in more than 4 dimensions for most problems, since the number of samples grows too large. For visualizing 1D or 2D slices in likelihood, however, we recommend it. This can also be useful at the start of an analysis - keeping all parameters but one or two fixed to gain an intuition for the problem.

For most standard problems we recommend starting with the `maxlike` sampler to find the peak of the probability distribution, and from there[12] running the `emcee` sampler.

For all samplers the command line usage is identical: `cosmosis [ini]` where `[ini]` is the user specified ini file. For samplers which can be run in parallel (`grid` and `emcee`) the command line usage is `mpirun cosmosis --mpi [ini]`. For technical reasons the `--mpi` flag should be ommitted when running `multinest`. When using each of the samplers the CosmoSIS ini file should contain the `[pipeline]`, `[output]` and `[module]` interface sections together with the following sampler specific options.

When using the `test` sampler the CosmoSIS ini file should contain the following

```
[runtime]
sampler = test

[test]
fatal-errors = [boolean T/F]
save_dir = [output directory]
```

After execution, `output directory` will contain any data products generated during pipeline execution. If `fatal-errors` is set, any exceptions will cause the sampler to exit immediately. The pipeline is evaluated at the start values for each parameter defined in values.ini.

When using the `grid` sampler the CosmoSIS ini file should contain the following

```
[runtime]
sampler = grid

[grid]
nsample_dimension = [integer]
```

where `nsample_dimension` is the number of points sampled in each parameter dimension.

When using the `maxlike` sampler the CosmoSIS ini file should contain the following

```
[runtime]
sampler = maxlike

[maxlike]
tolerance = 1e-3
maxiter = 1000
output_ini = [output ini file]
```

The `tolerance` sets the fractional convergence criterion for each parameter; `maxiter` is the maximum number of steps to take before giving up. If `output_ini` is set this provides an output ini file with the best fit as the central value. In particular the output_ini option is useful to provide to other samplers that benefit from starting positions near the global maximum.

When using the `metropolis` sampler the CosmoSIS ini file should contain the following

```
[runtime]
sampler = metropolis

[metropolis]
covmat = covmat.txt
samples = 100000
Rconverge = 0.01
nsteps = 100
```

`samples` is the maximum number of samples which will be generated. The run can stop earlier than this if multiple chains are run and the `Rconverge` is set - this is a limit on the Gelman Rubin statistic. The proposal is along eigenvectors of the covariance matrix, rotated to avoid backtracking. Of the covariance matrix is not specified in the `covmat` argument a default one will be generated based on the ranges specified in the values file.

When using the `emcee` sampler the CosmoSIS ini file should contain the following

```
[runtime]
sampler = emcee

[emcee]
walkers = 200
samples = 100
start_points = start.txt
nsteps = 50
```

The number of `walkers` must be at least $2 * \text{nparam} + 1$, but in general more than that usually works better (default = 2); `samples` is the number of steps which each walker takes (default = 1000) and sample points are output every `nsteps` (default = 100). The starting points for each walker in the chain may be specified in the ini file using `start_points = start.txt` where start_file.txt contains (number of walkers, number of params) values. If this start file is not given then all walkers are initialized with uniform random numbers from the range of values in

[12] The CosmoSIS max-like sampler has an option to output a values file starting from the best-fit point it finds.

values.ini. For practical Monte Carlo the accuracy of the estimator is given by the asymptotic behaviour of its variance in the limit of long chains. Forman-Mackney et al. advocate the following: Examining the acceptance fraction which should lie in the range 0.2-0.5. Increasing the number of walkers can improve poor acceptance fractions. Estimating the autocorrelation time which is a direct measure of the number of evaluations of the posterior probability distribution function required to produce independent samples of the target density.

Appendix A.6. Running CosmoSIS

Regardless of the sampler or other parameter choices, CosmoSIS is run through a single executable invoked on the configuration file. MPI parallelism is enabled at the command line flag (in combination with any mpirun command required), and any parameter in the configuration files can be over-written using another flag (this feature is mainly useful for debugging).

Appendix A.7. Processing outputs

A post-processing program for sampler output, simply called `postprocess` uses the output of chain and grid samplers to generate 1D and 2D marginalized constraint plots and numbers. You call it on the same ini file that was used to generate the chain in the first place, so that any type of chain (grid, mcmc, or any others that we add) are analyzed with the same executable.

An example output of the `postprocess` command on the `emcee` sampler is shown in Figure 4, and from the grid sampler in Figure 6.

Appendix B. Developers's Guide

Most users of parameter estimation codes go on to modify and write their own code. Making this easy is the whole point of CosmoSIS.

There are several ways you can develop within CosmoSIS:

- Modify modules in an existing pipeline, for example to study new physics.
- Add modules that do different physical calculations than what is currently available.
- Add a new likelihood to the end of a pipeline to combine new datasets.
- Insert a module into the middle of a pipeline, for example to model a new systematic error.
- Start from scratch creating a new pipeline using the CosmoSIS structure and samplers.
- Add a new sampler and test it on existing problems.

Appendix B.1. Creating new modules

If no existing module does the calculation you need, or if you wish to wrap an external piece of existing code, then you can create a new module for it. Each new module should live in its own directory.

Unless a module is exceedingly simple it is best for new modules to be in two parts - the part that does the calculation, and the part that interfaces with CosmoSIS. In the case of wrapping existing code the former usually exists already and only the latter needs to be created.

To write a module it suffices to write the three functions described in Section 4.2. This involves thinking carefully what the inputs and outputs will be to this module, and deciding which of the inputs will definitely be fixed throughout a run (for example, a path to a data file, a redshift at which an observation has been made, or a choice of which model to use), and which are those which may at some point be varied throughout a chain (such as cosmological parameters).

Appendix Appendix C shows an example of a simple module.

Appendix B.1.1. Module form

Modules implemented in python are connected to CosmoSIS using a single python file that implements the functions described below. Modules implemented in a compiled language (C, C++, or Fortran) must be compiled into a shared library, which can be loaded dynamically from python. This simply involves compiling all files and linking them together with the `-shared` flag. Examples can be found in the CosmoSIS standard library.

Appendix B.1.2. setup

The setup function is called once, at the start of a run when a pipeline is being created. It is a chance to read options from the CosmoSIS parameter files, allocate memory and other resources, and load any data from file. If a distributed parallel sampler is used this may mean being called by each processor separately.

The setup function is passed a datablock object which contains all the information from the parameter file. In particular, options set in the section of the ini file corresponding to this particular module have a specifically named section; modules need only look at this section. The API for accessing the data from the configuration file is described in Appendix Appendix E.

The setup function can return any arbitrary configuration information, which is then passed back to the exectue function below. This mechanism means that the same module can be run twice with different options (for example, the same likelihood module can be used with two different data sets).

Appendix B.1.3. execute

The execute function is the main workhorse of a module - it is called each time the sampler runs the pipeline

on a new set of parameters. The execute function takes two parameters, one containing the parameters from the sampler and any data from modules earlier in the pipeline, and one containing the configuration information from the setup function.

A typical module reads some scalar parameters or vector data from the block, and then performs some calculations with it depending on the choices made in the ini file. It then saves new scalar or vector data back to the block. Appendix Appendix E describes the API for loading and saving values.

Appendix B.1.4. cleanup

A cleanup function is run when the pipeline is finished, and should free any resources loaded by the setup function. In many cases this can be completely empty. This function is passed the configuration information from the setup function.

Appendix B.1.5. New likelihoods

Any module can, as well as doing any other calculations, save values into the `likelihoods` section. This section has a special meaning - the samplers will search in it for any likelihoods that they are told to find in the parameter file. If the parameter file says, for example:

```
[pipeline]
likelihoods = hst planck
```

then the sampler will look for `hst_like` and `planck_like` in the `likelihoods` section.

Appendix B.2. Modifying existing modules

When you want to test a new theory it is usually easiest to take an existing module and modify it to implement the new theory. For example, the CosmoSIS module to calculate the growth function could be changed to implement a modified gravity scenario.

Modifying an existing module to extend it by add new calculations, rather than modifying existing ones, is usually a sub-optimal choice, since new calculations can be better integrated with other modules if they are in a separate module. Consider writing a new module instead.

You would take these steps to modify an existing module:

- Copy the existing module to a new location.
- Version control your new module.
- Modify the main module science code.
- Modify the interface code if any new inputs or outputs are required.

Appendix B.3. Inserting modules

If you want to make a modification to a quantity from a physical effect that a pipeline does not currently consider, then you can insert a new module in the middle of the pipeline to implement it.

For example, it is known that baryons have a feedback effect on the matter power spectrum which is important for observations probing non-linear cosmic scales. One might insert a module after e.g., the Halofit module, to modify the non-linear matter power to account for this effect, so that subsequent modules would use the modified power.

Inserted modules can be created as described in Section Appendix B.1, but there is one additional consideration. The CosmoSIS DataBlock (see Section 4.3) makes a distinction between saving new values and replacing existing ones. Modules that modify existing data need to use the `replace` operations described in Appendix Appendix E instead of the `put` ones.

Appendix B.4. Adding samplers

New samplers can be easily added to CosmoSIS if they can be called from python; this includes any sampler usable as a library (e.g. with a few simple functions that can be called to run the sampler with an arbitrary likelihood function) in C, C++, or Fortran, as well as python samplers.

Interfaces to samplers are implemented by subclassing a Sampler base class, in the `cosmosis/samplers` subdirectory. The subclasses must implement three methods:

- `config`, which should read options from the ini file, and perform any required set up. The superclass instance has an instance of the ini file used to create the run and the instantiated pipeline itself.
- `execute`, which should perform a single chunk of sampling, and saving the result - superclass methods can be used for the latter. CosmoSIS will keep re-running the `execute` function until the sampler reports it is converged.
- `is_converged`, which should return True if the sampling should cease. A simple sampler might always return True, but a more complex sampler can examine the chain so far and use real convergence diagnostics.

Parallel samplers inherit instead from the ParallelSampler superclass, which as well as Sampler's features maintains a pool of processes, each with their own instance of the pipeline, and an `is_master` method to decide if the given process is the root one.

Appendix C. Worked example

In this appendix we show a worked example of a CosmoSIS pipeline, and go into detail about one of the modules in it.

Our example will be CosmoSIS demo number six, which calculates the likelihood of the CFHTLenS tomographic data set given some cosmological parameters.

Appendix C.1. Overview

The CFHTLenS observed data vector is a set of measurements of ξ_+ and ξ_-, correlation functions of the cosmic shear. We have measuremnts $\xi_\pm^{ij}(\theta_k)$ for each pair of redshift bins[13] (i, j) and for a range of angular scales θ_k.

The steps we need to take to calculate the likelihood from the cosmological parameters are therefore:

1. Calculate the linear matter power spectrum $P(k, z)$ across the desired redshifts
2. Calculate the non-linear power spectrum from the linear
3. Calculate the redshift distribution of the survey
4. Perform the Limber integral to get the shear angular power spectra.
5. Integrate the angular power spectra with Bessel functions to get the angular correlation function.
6. Get the likelihood of the CFHTLenS measurements given these theory correlation functions.

This pipeline is shown in Figure C.8.

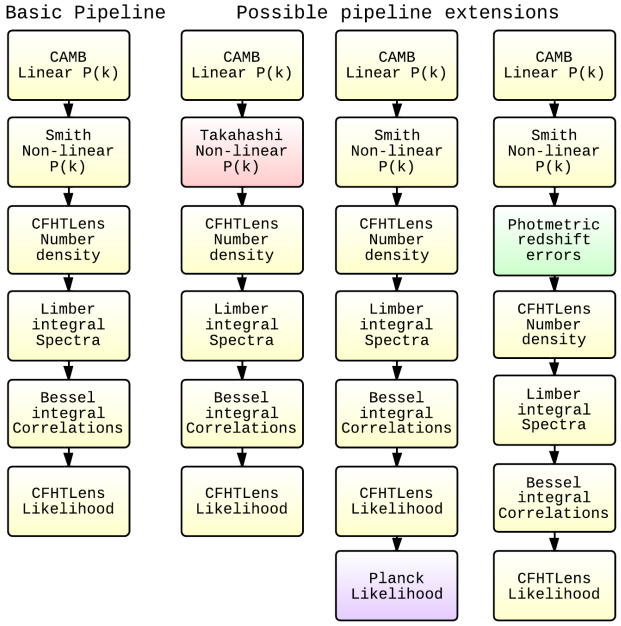

Figure C.8: A schematic of the CFHTLenS pipeline. The basic pipeline is on the left, and in the three alternatives on the right we illustrate replacing (red), appending (purple) and inserting (green) modules.

[13]Since CFHTLenS is a photometric experiment the redshifts are approximate, so the actual redshift distribution in each bin is different from the nominal one. We must account for this in the analysis.

Appendix C.2. Modifications

Figure C.8 also shows various changes we might wish to make to this pipeline; making it easy to implement these changes is a core goal of CosmoSIS.

The simplest example is replacing one module with another, in this case changing the method used to calculate non-linear power. Provided each module supplies the same outputs, subsequent modules can be left unchanged by this replacement.

We can also straightforwardly attach new likelihoods to the end of the pipeline, illustrated here by adding the Planck likelihood. This likelihood requires new nuisance parameters, which can be supplied by the sampler simply by adding them to the input values file.

Finally, we might insert a module into the middle of the pipeline, in this case to test for a systematic error. If we decided, for example, to model errors in the photometric redshifts determined by CFHTLenS then we could modify and replace the $n(z)$ used in the spectra.

Appendix C.3. Pipeline implementation

Each box in Figure C.8 is a single CosmoSIS module. Each does a single calculation and gets a relatively small collection of inputs from the DataBlock, and puts its outputs there once they are calculated.

The configuration file that runs this module needs to define the sampler to be used, just the test sampler in this case:

```
[runtime]
sampler = test

[test]
save_dir=demo6_output
fatal_errors=T
```

and also the modules to be run and the likelihoods extracted:

```
[pipeline]
modules = camb halofit load_nz shear_shear 2pt
    cfhtlens
values = demos/values6.ini
likelihoods = cfhtlens
```

The `value` parameter lists a file where the values and ranges of the parameters to be sampled are specified. Each module to be run is described elsewhere in thre configuration file, in some cases with extra options:

```
[camb]
file = cosmosis-standard-library/boltzmann/camb/camb.
    so
mode=all
lmax=2500
feedback=0

[halofit]
file = cosmosis-standard-library/boltzmann/halofit/
    halofit_module.so
...
```

Appendix C.4. Module implementation

The simplest module in this pipeline is the one that performs the integration with Bessel functions to convert angular power spectra to correlation functions. In this section we will describe the interface that connects this module to CosmoSIS in detail.

This particular module is implemented in python, but similar (if slightly more complex) considerations apply to the other supported languages.

Appendix C.4.1. Preamble

We will not delve here into the implementation of the main workhorse of this module; we simply import it into python. Note that we have separated the main functionality of the code from the part that connects it to CosmoSIS; this make it easy, for example, to use the same code in other programs easily

```
import cl_to_xi
import numpy as np
from cosmosis import option_section, names as
    section_names
```

Appendix C.4.2. Initial setup

The setup function is run once when the pipeline is created. Options from the configuration file are passed into it as the `options` object, which is a DataBlock. The `option_section` is a shorthand for the section that applies to this particular module (modules can in principle find out what other modules are to be run also).

The various required options (which concern the angular ranges at which to calculate correlations) are read here and the range constructed.

Anything returned by this function will be passed to the `execute` function laster. In this case that means a dictionary, `config`, that contains the vector `theta`, over which to compute the correlation functions.

```
def setup(options):
        config = {}
        n_theta = options[option_section, "n_theta"]
        theta_min = options[option_section, "theta_min
            "]
        theta_max = options[option_section, "theta_max
            "]
        theta_min = cl_to_xi.arcmin_to_radians(
            theta_min)
        theta_max = cl_to_xi.arcmin_to_radians(
            theta_max)
        theta = np.logspace(np.log10(theta_min), np.
            log10(theta_max), n_theta)
        config["theta"] = theta

        return config
```

Appendix C.4.3. Execution

This function is called each time the pipeline is run with new cosmological parameters. The `block` input contains the parameter-specific information: the values provided by the sampler itself, and the calculation results done by previous modules. The `config` input contains fixed data passed from the `setup` function (though this could in principle be modified, for example to cache results).

It reads the inputs from the pipeline section `shear_cl`, which are the ell range `ell` provided by the modules that came before it, and the `bin_1_1`, etc., giving the angular power spectra. The results are saved into `shear_xi`.

```
def execute(block, config):
        thetas = config["theta"]
        n_theta = len(thetas)

        section = section_names.shear_cl
        output_section = section_names.shear_xi
        ell = block[section, "ell"]
        nbin = block[section, "nbin"]
        block[output_section, "theta"] = thetas
        block.put_metadata(output_section, "theta", "
            unit", "radians")

        for i in xrange(1,nbin+1):
                for j in xrange(1,i+1):
                        name = "bin_%d_%d"%(i,j)
                        c_ell = block[section, name]
                        xi_plus, xi_minus = cl_to_xi.
                            calculate_xi(ell, c_ell,
                            thetas)
                        block[output_section, "xiplus_%
                            d_%d"%(i,j)] = xi_plus
                        block[output_section, "ximinus_
                            %d_%d"%(i,j)] = xi_minus
        return 0
```

Appendix C.4.4. Clean up

In python there is rarely any clean up to be done, since memory is managed automatically. In C or Fortran you might deallocate memory here.

```
def cleanup(config):
    return 0
```

Appendix D. Architectural Details

Appendix D.1. DataBlocks

DataBlocks are organized into sections, named categories of information. For example, `cosmological_parameters`, `cmb_cl` and `intrinsic_alignment_parameters` can all be sections. A number of common sections are pre-defined in CosmoSIS, but they are simple strings and new ones can be arbitrarily created. A datablock may be thought of as a dictionary mapping from a pair of strings (a section name

and a specific name for data in that section) to a generic value.

For example, the `cosmological_parameters` section would typically contain `omega_m`, `h0`, and other scalar doubles, and the `cmb_cl` section contains an integer array `ell` and double arrays `TT`, `TE`, `EE`, and so on.

Native APIs that act on datablocks exist for C, Fortran, C++, and Python to read or write the data stored in the block. The interfaces to modules (see below) call this API, as do the samplers when they create the block in the first place.

There are also introspection functions that work on datablocks so that modules can peform context-dependent calculations (for example, we might check for a galaxy bias grid $b(k, z)$ and if one is not found revert to a single scalar b value).

Appendix D.2. Samplers

Samplers are connected to CosmoSIS by sub-classing from a base class which provides access to the pipeline and to configuration file input, and to output files. Subclasses implement methods to `config` (read options from the ini file and perform setup) `execute` (run a chunk of samples) and test `is_converged` to see if the process should stop. Adding a new sampler is straightforward and we would welcome contributions.

Appendix E. API

The CosmoSIS application programming interface (API) defines a way for a module to save and load data from a block designed to collect together all the theoretical predictions about a cosmology. The API is fully documented on the CosmoSIS wiki[14].

The API can handle the following types of data:

- 4-byte integers
- 8-byte floating-point (real) values
- 4-byte boolean (logical) values
- ASCII strings
- $8 + 8$ byte complex numbers
- vectors of 4-byte integers
- vectors of 8-byte floats
- vectors of $8 + 8$-byte complexes
- n-dimensional arrays of 4-byte integers
- n-dimensional arrays of 8-byte floats
- n-dimensional arrays of $8 + 8$-byte complexes

[14] `https://bitbucket.org/joezuntz/cosmosis/wiki`

Any value stored in a block is referenced by two string parameters, a section defining the group in which it is stored, and a name of the value. For each type in each supported programming language there are `get`, `put` and `replace` functions. There are also a number of additional utility functions to check whether values exist in the block, and similar tasks.

In this section we show a handful of available API calls to demonstrate their general structure.

Appendix E.1. C

Most C functions return type `DATABLOCK_STATUS`, an enum. For each type listed above there are function to `get`, `put`, and `replace` values. For the scalar types there are also alternative `get` functions where a default value can be supplied if the value is not found. In the case of 1-d array there are two `get` functions, one to use preallocated memory and one to allocate new space, which the module is reponsible for disposing of. For the n-d arrays there are also functions to query the number of dimensions and shape of the array.

The function to get an integer, for example, has this prototype:

```
DATABLOCK_STATUS
c_datablock_get_int(c_datablock* s, const char*
    section, const char* name, int* val);
```

Appendix E.2. Fortran

The Fortran functions closely follow the C ones, and use the `iso_c_binding` intrinsic module to define types. For example:

```
function datablock_get_double(block, section, name,
    value) result(status)
    integer(cosmosis_status) :: status
    integer(cosmosis_block) :: block
    character(*) :: section
    character(*) :: name
    real(c_double) :: value
```

Appendix E.3. Python

While `get_int` and similar values are present in the python API, the most straightforward mechanism to load and save values is the idiomatic python get and set syntax:

```
block["section_name", "value_name"] = value
```

All the python functions are methods on a DataBlock object.

Appendix E.4. C++

In C++ the `get`, `put`, and `replace` functions are all templated methods on a `DataBlock` object, so the same method is used for all data types, for example:

```
template <class T>
DATABLOCK_STATUS put_val(std::string section, std::
    string name, T const& val);
```

Appendix F. Improving 2D KDE

Kernel density estimation (KDE) is a method for smoothing a collection of MCMC samples to produce a better constraint plot. It can be applied in any number of dimensions, and can be thought of as placing a smooth Gaussian (or other) kernel atop each MCMC sample and using the sum of all these Gaussians as the likelihood surface. The main choice to be made is the covariance matrix (or just width in one dimension) of the kernel, which is typically taken as some scaling factor times the covariance matrix of the samples.

An occasional objection to KDE is that the recovered contours drawn on the smoothed distribution do not typically contain the correct fraction of samples (68%, 95%, etc.) that they should do if the samples and the contours accurately represented the same posterior surface.

The CosmoSIS post-processing code implements uses KDE with a minor improvement when used in 2D. The 2D likelihood surface is generated as in normal KDE. The contours drawn on them, though, are not chosen with reference to the probability volume beneath the smoothed contours, but rather by interpolating so that the correct number of samples from the MCMC is beneath them. That is, the KDE provides the *shape* of the contours, but the sample count provides their *size*. We find that this procedure improves the fidelity of the recovered contours.

Appendix G. Parameter consistency & alternate specifications

In cosmology, as in most parameter estimation problems, there are a number of different parameterizations one can use to specify the space. The choice affects how easy it is to specify parameter priors, and how efficent sampling in the space can be (for most algorithms parameters with roughly Gaussian posteriors make for better sampling).

In some cases deducing the "derived" parameters from the specified ones requires complex calculations (for example, getting σ_8 from A_s) but in other cases the relations are relatively simple arithmetic.

CosmoSIS includes a module with an algorithm for the latter case which allows one to specify any sufficient combination of parameters, and deduce the rest, no matter which combination they are in. The steps of this algorithm are:

1. Specify a comprehensive collection of relations between parameters as strings, for example `omega_m = omega_c+omega_b`, `omega_c = omega_m-omega_b`, etc. Call the number of relations n.
2. Parse the left-hand side of the relations to get a set of all the parameters to be calculated.
3. Initialize a dictionary of all these parameters with the special value NaN (not-a-number) for each of them.
4. For any parameters which are provided by the user, initialize with the specified value.
5. Iterate at most n times:
   (a) Evaluate each relation in the collection, with the current parameters using the python `eval` function with the parameter dictionary as the namespace. There are three cases:
   - If the result is NaN, do nothing - this means at least one input to the relation was unspecified (NaN).
   - If the result is not NaN and the current parameter value is NaN, then we have a newly calculated parameter. Update the dictionary with this value.
   - If the result is not NaN and the current parameter value is not NaN, then we have recalculated the parameter. Check that this new calculation is the same as the old one. If not, raise an error: the model is over-specified.

   (b) If there are no NaN parameter left then we have finished; save all the parameters.
6. If we evaluate all the relations n times without calculating all parameters then there are some we cannot calculate - the model is under-specified and we raise an error.