CosmoSIS: a system for MC parameter estimation

S Bridle¹, S Dodelson²,³,⁶, E Jennings³,⁴, J Kowalkowski², A Manzotti³,⁶, M Paterno², D Rudd³,⁵, S Sehrish² and J Zuntz¹

¹ Jodrell Bank Centre for Astrophysics, University of Manchester, Manchester M13 9PL, U.K.
² Fermi National Accelerator Laboratory, Batavia, IL 60510-0500, U.S.A.
³ Kavli Institute for Cosmological Physics, University of Chicago, Chicago, IL 60637, U.S.A
⁴ Enrico Fermi Institute, University of Chicago, Chicago, IL 60637, U.S.A
⁵ Research Computing Center, University of Chicago, Chicago, IL 60637, U.S.A
⁶ Department of Astronomy & Astrophysics, University of Chicago, Chicago, IL 60637, U.S.A
E-mail: paterno@fnal.gov

Abstract.
CosmoSIS is a modular system for cosmological parameter estimation, based on Markov Chain Monte Carlo and related techniques. It provides a series of samplers, which drive the exploration of the parameter space, and a series of modules, which calculate the likelihood of the observed data for a given physical model, determined by the location of a sample in the parameter space. While CosmoSIS ships with a set of modules that calculate quantities of interest to cosmologists, there is nothing about the framework itself, nor in the Markov Chain Monte Carlo technique, that is specific to cosmology. Thus CosmoSIS could be used for parameter estimation problems in other fields, including HEP.

This paper describes the features of CosmoSIS and show an example of its use outside of cosmology. It also discusses how collaborative development strategies differ between two different communities: that of HEP physicists, accustomed to working in large collaborations, and that of cosmologists, who have traditionally not worked in large groups.

1. Introduction
CosmoSIS [1] is a modular system for cosmological parameter estimation, based on Markov Chain Monte Carlo (MCMC) [2] and related techniques. It provides several samplers, each of which drives the exploration of the parameter space in a different fashion. It also provides a set of modules to perform the calculation of widely-used physical parameters (e.g., the linear matter power spectrum $P(k, z)$) from fundamental cosmological parameters, using several software packages popular in the cosmology community. CosmoSIS also includes likelihood modules, which calculate data likelihoods for a given physical model, determined by the location of a sample in the parameter space.

While the physics modules and data likelihood modules included with CosmoSIS are specifically of interest to the cosmology community, neither the samplers nor framework that drives the sampling process are in any way specific to cosmology. Thus they can be used for parameter estimation problems in other fields, including HEP.

CosmoSIS was developed in and for a community organized differently than the HEP community. Much HEP software is written by and for large collaborations, and the C++ programming language is overwhelmingly the most popular. Because most cosmologists do
not work in a large collaboration that enforces software guidelines, such as the choice of a programming language, the framework must support programming in multiple languages. Additionally, since scientists in the cosmology community are used to working independently, a system was needed for helping ensure that proper attribution is given to authors of contributed algorithms.

In section 2 of this paper, we describe the CosmoSIS framework; in section 3 we describe its use in a simplified HEP “bump hunt” problem. In section 4 we describe some of the differences between software development methods typical to the cosmology and HEP communities, and in section 5 we conclude with some observations about how each community might benefit by learning some techniques used successfully by the other.

2. The CosmoSIS framework

2.1. The structure of CosmoSIS

In CosmoSIS, a parameter estimation problem is represented as a pipeline consisting of one sampler and one or more modules. The sampler and modules are invoked by the runtime, which is also responsible for writing the relevant output files. The sampler and modules do not communicate directly; instead, they each read values from, and write values to, an object called the datablock.

A sampler drives the exploration of the parameter space according to some specified algorithm (e.g. Metropolis-Hastings). The configuration of the sampler defines the parameter space to be explored: the number and names of parameters, as well as the ranges of variation of the parameters, are determined by the user-specified configuration of the sampler.

Pipeline modules are used to perform the calculation of physical quantities required by other modules later in the pipeline. Modules can also calculate data likelihoods: the likelihood of observing some specified data, given the physical model and the values of the parameters for a given sample.

Modules do not communicate directly with each other, nor do they directly communicate with the sampler. They communicate indirectly through the datablock, which is an object that carries named parameters of a variety of numeric types: integral, real and complex, in both scalar and multi-dimensional array forms. The sampler inserts values of the sampled parameters into the datablock; modules can read these parameters and those inserted into the datablock by other modules, and write new values into the datablock. Modules can also calculate data likelihoods, which are combined by the runtime, and provided to the sampler, to guide the generation of the next sample. Figure 1 shows an example of how a full pipeline is organized.

![Diagram of CosmoSIS pipeline](image)

**Figure 1.** An example cosmological pipeline. The arrows indicate which modules write to, or read from, each element in the datablock. The order of execution of the modules in the diagram is left-to-right.
The similarity to most modular HEP event processing frameworks is clear. An interesting difference is that, unlike the typical Event class, the datablock does not carry the data corresponding to experimental observations; rather, it carries values corresponding to a realization of the model for which the parameter estimation is being done. Observational data is typically confined to individual likelihood modules, which calculate the likelihood of the observation of those data given the model parameters specified by the sample represented in the datablock. These data are typically loaded at module initialization time, and retained throughout the execution of the program.

Modules can be written in Python, C, C++, and Fortran. In order to help assure reproducibility and portability, CosmoSIS supports specific versions of various language standards. Release 1 of CosmoSIS supports Python 2, C99, C++11, and Fortran 2003. Several of the samplers can work in parallel; for these, CosmoSIS supports process-level parallelism using the Message Passing Interface (MPI). Thread-level parallelism in modules is also supported, for example using OpenMP.

CosmoSIS modules written in Python are also modules in Python’s sense of the word; they must provide the required interface: for C, C++ and Fortran, modules are dynamic libraries which provide specific function names and signatures. In each case, the required functions are an initialization function, a cleanup function called at program end, and a function that is called for each sample generated by the sampler.

2.2. Installation and building
CosmoSIS can be installed by downloading and executing a single script, which installs the entire system. The components of the system fall into four categories: 1) tools installed, but not modified by the user; 2) the core framework; 3) the CosmoSIS standard library; and 4) collections of third-party modules.

Tools like the C, C++ and Fortran compilers, the Python runtime, and the MPI implementation are delivered in binary format, using a combination of technologies: UPS [3], Python’s pip, and conda from Continuum Analytics. Binary delivery of a set of tools that has been verified to be compatible is critical for the use of a diverse body of code, written in several different languages. These tools are all installed in a fashion to not conflict with any other versions of the tools installed on the same machine.

The core framework, and the library of standard modules, is each delivered as a git repository. This allows users to modify the modules of the standard library, or even the core framework, if they wish to do so. Users can add their own modules alongside the modules of the standard library. Any number of third-party repositories may also be installed; this is especially convenient for collaborations, who can establish their own repository for modules to be shared among collaborators. Building the core framework and all compiled modules is done with make, and the system provides makefile fragments and examples to help authors of compiled modules assure that their modules are compiled and linked to be compatible with the rest of CosmoSIS.

2.3. Runtime environment
The runtime environment upon which CosmoSIS relies is established by a setup script that makes available, at the command line, the correct version of the C, C++ and Fortran compilers, of the Python interpreter, and of widely-used packages like LAPACK, NumPy, matplotlib.

Execution of CosmoSIS is controlled primarily by configuration files. The sampler and selection of modules to be executed, and the order of the modules, is determined by the configuration file. Samplers and modules are all user-configurable, using a simple configuration language (ini files, as defined by the Python standard library ConfigParser class).

The output produced by CosmoSIS includes the complete information of the configuration of the sampler and the set of modules used to create that output. In addition, each module can
have an associated text file with attribution information; if such a file is found, the information in it is also written to the output. This helps make it easier for authors who contribute modules for the use of others to obtain appropriate acknowledgment for their scientific contributions.

2.4. A complete toolkit
Realistic MCMC parameter estimation problems often require significant computational effort. In order to allow work on such problems, CosmoSIS supports the use of MPI. Users of modules do not need to deal with MPI directly; instead, the samplers amenable to parallelism use it to parallelize the calculation. Because MPI is a distributed parallel system, this allows the use of both multi-core nodes and distributed nodes. The system also provides the ability to continue sampling from a previously saved chain. The ability to run using MPI, combined with the delivery of an MPI system as part of the installed base of software, means that one can develop an analysis on a laptop, and move it to a cluster when greater computational resources are required.

In addition to the sampling application, CosmoSIS also includes a set of tools for analysis of the generated samples. These tools include extensible plotting facilities and tools for statistical analysis. The plotting tools provide generation of both one- and two-dimensional plots of posterior densities and likelihoods. Figures 2 and 3 show samples of these plots. Convergence tools include calculation of the Gelman-Rubin statistic and an autocorrelation length test.

![Figure 2. Plot of the likelihood for the cosmological parameter $\sigma_8$, from example 7 of the CosmoSIS instruction manual. The dotted lines indicate the 68% and 95% boundaries.](image1)

![Figure 3. Contour plot of the joint likelihood for the cosmological parameters $\sigma_8$ and $\Omega_m$, from example 7 of the CosmoSIS manual. The color scale shows the magnitude of the likelihood; the contour lines contain the 68% and 95% regions.](image2)

3. An HEP use case
To demonstrate the use of CosmoSIS outside of the domain of cosmology, and to show the simplicity of coding a likelihood module, we have implemented a toy version of a common HEP analysis: the determination of the cross section, mass, and width of a mass resonance, in the presence of a much larger background.

We proceed by generating a simulated sample of observed “events” from both signal and background. We choose our resonance to have mass $\mu$ and width $\delta$, with a Gaussian line shape,
and total production cross section $\sigma_s$. We choose the background to have a falling exponential distribution, with e-folding length $\beta$, and total production cross section $\sigma_b$. We simulate an experiment with integrated luminosity $L$ by selecting a number of signal events $N_s$ from a Poisson distribution with mean $L\sigma_s$, and a number of background events $N_b$ from a Poisson distribution with mean $L\sigma_b$. We then generate $N_s$ samples from a Gaussian distribution with mean $\mu$ and standard deviation $\delta$, and $N_b$ samples from an exponential distribution with parameter $\beta$. Each of these samples represents an event with observed “mass” $m$. For our simulated experiment, we choose (all in arbitrary units) $L=100$, $\sigma_b=2000$, $\beta=40$, $\sigma_s=2.5$, $\mu=232.2$, and $\delta=7.4$. The generated distribution of $m$ is shown in figure 4.

To perform the parameter estimation, we bin the observations, choosing a bin width of 2, in our arbitrary units; this is the same binning as shown in figure 4. The likelihood for observing these data, given the model parameters, is a product of the likelihood of observing the number of counts in each bin. For each bin $i$, the likelihood of observing a count $n_i$ is given by the Poisson distribution with mean $\hat{n}_i$, where $\hat{n}_i$ is determined by the product of the luminosity $L$ and the sum of the integrated signal and background cross sections over bin $i$:

$$\hat{n}_i = L\int_{\ell_i}^{u_i} dx \left( \frac{\sigma_s}{\sqrt{2\pi}\delta} e^{-\frac{1}{2}\left(\frac{x-\mu}{\delta}\right)^2} + \frac{\sigma_b}{\beta} e^{-x/\beta} \right),$$

where $\ell_i$ is the lower edge of bin $i$ and $u_i$ is the upper edge of the same bin.

Our likelihood function is the product of the Poisson likelihoods for each bin. For bin $i$, the probability $p_i$ is:

$$p_i = e^{-\hat{n}_i} \hat{n}_i^{k_i} / k_i!$$

where $k_i$ is the number of events observed in bin $i$.

Noting that the integral of the Gaussian distribution over a finite interval is given by the difference of two error functions, we find that we can encode the likelihood function using NumPy and SciPy in a succinct and efficient fashion, taking advantage of those libraries’ vectorized math functions. The code that does the likelihood calculation is shown in figure 6.
def execute(block, cfg):
    # Read this sample's parameters from the block
    lum, sigma_b, beta, sigma_s, mu, delta = ... # elided
    # Calculate the expected counts in each bin for this sample
    lows = cfg.lowedges
    highs = cfg.lowedges + cfg.binwidth
    f1 = numpy.exp(-1.0 * lows / beta)
    f2 = numpy.exp(-1.0 * highs / beta)
    expected_bkg = lum * sigma_b * (f1 - f2)
    sqrt2sigma = numpy.sqrt(2.0) * delta
    g1 = special.erf((mu - lows) / sqrt2sigma)
    g2 = special.erf((mu - highs) / sqrt2sigma)
    expected_signal = lum * sigma_s * (g1 - g2) / 2.0
    expected_counts = expected_signal + expected_bkg
    # Calculate log-likelihood for our data, for this sample
    loglike = numpy.sum(-expected_counts + cfg.counts *
                         numpy.log(expected_counts) - cfg.lnfactcounts)
    block[likes, 'BUMP_HUNT_LIKE'] = loglike
    return 0

Figure 6. The execute function from the bump_hunt.py likelihood module.

The fitted values of the mass and width are $231.8 \pm 1.0$ and $6.8 \pm 1.3$, respectively. The joint posterior density for these two parameters is shown in figure 5.

4. Development strategies and choices
One of the key differences between the HEP and cosmology communities is that cosmologists work on their own much more often. The HEP software community has thus had to solve some problems of scale which the cosmology community are only now solving for themselves. In developing CosmoSIS, we have found that some of the techniques used in HEP are of direct benefit. Here we concentrate on a few of these techniques.

4.1. Packaging and delivery of software
The ability to deliver a set of software tools (e.g., the GCC compiler suite or an MPI implementation) reliably, so that all supported platforms can be assured not merely of having a version of the tools but the same version of the tools, has been a critical part of making CosmoSIS reliable. Part of this reliability is reproducibility, which is enhanced by providing a consistent set of tools with specified versions. When comparing different outputs from different users’ execution of a program, one does not need to worry that the differences are because different versions of some underlying library are being used.

The lowest-level packages are one which, while necessary for work, are not one which our users have any interest in modifying; none of our users are going to alter the Fortran compiler itself, nor to re-implement part of MPI. Thus the delivery of pre-built libraries and executables allows users to gain access to the tools without wasting time building them locally, for supported platforms. Users who want to build products on their own are, of course free to do so.

It is critical that users with no elevated privileges be able to install CosmoSIS. That means we
could not require installation at any pre-specified filesystem path (e.g. /usr/local). The UPS system used by CosmoSIS permits this by providing relocatable products: products that can be built at one location in the filesystem, and then installed in another location. The UPS system is shared with the neutrino and muon experiments at Fermilab, and is supported by the Fermilab Scientific Computing Division. Thus we gained the ability to make use of a tested and supported system, without having to invent one ourselves. We also contributed to that system, for example by creating the first fully relocatable Python package. Using UPS we are able to control the specific version of software in current use, while allowing multiple versions to be present on the system. The tight control over the active versions provides a strong assurance of binary compatibility of the products, which is especially important for Fortran and C++ libraries, both of which can give rise to difficult-to-diagnose failures from subtle incompatibilities.

4.2. Hierarchy of code and contributions
While CosmoSIS can be installed by running a single script, it is not a monolithic body of code. Rather, it consists of several layers, with the different layers corresponding to different types of interaction with the code. The underpinning is the binary distribution of the lowest-level products, managed by UPS, as described in the previous subsection. Above this is the core framework, delivered via a git repository. This code is not touched by most users, but is available to them in case they want to experiment, or just want to study the code. Above the core framework are the modules and libraries supply as the CosmoSIS standard library, delivered by another git repository. This code is available for use, and modification and extension, by users. Finally, there can be any number of experiment- or user-supplied repositories of code. These allow the finest level over sharing, tuned to each experiment or user’s need. This layered approach allows tradeoffs between convenience and flexibility.

4.3. Attribution protocol
It is sometimes difficult for scientists who have developed significant and useful bodies of code to obtain appropriate attribution for their work. In large HEP collaborations, the internal social mechanisms of the collaboration largely alleviates this problem. But in a community that is not dominated by large collaborations, these mechanisms are lacking.

The adoption of a simple protocol for attribution (the presence of a text file named X.yaml, expected to be found alongside the code for the module named X) allows CosmoSIS to automatically include the appropriate attribution information in the output it generates. This simple mechanism has been sufficient to help convince authors to share their work through this framework.

4.4. Multilingual programming
Because cosmologists often work in small groups, each individual or group makes choices of programming language, tools, etc. independently. In particular, C, Fortran and Python all have a significant number of practitioners in the field. With no large collaboration to enforce a language choice, cosmologists will use CosmoSIS only if they find it convenient to their style of work. Thus supporting multiple programming languages was a necessity. Support of the major languages in use in the community helped lower the bar to accessing the abilities of the system, and allows cosmologists to use tools developed in languages that they themselves do not prefer for programming.

5. Conclusion
In developing CosmoSIS, we have been able to take advantage of some of the lessons learned by the HEP community, for the benefit of the cosmology community. In addition, we were able to
share effort between otherwise unrelated efforts, to the benefit of both, by sharing some of the underlying tools and technologies between HEP programs and CosmoSIS.

As the collaborations in which cosmologists work grow in size, they are finding the same needs—and often finding the same solutions—as the HEP community. The use of shared repositories and more clearly controlled software stacks are both becoming widespread. In many issues that have, and will continue to be raised, the lessons learned by the HEP community should be of value to the cosmology community.

The cosmology community also has experience from which the HEP community might learn. In particular, the more open-source model for sharing of software, with an appropriate attribution system, might help encourage more widespread sharing of software products not only within large HEP collaborations, but between collaborations. We are, perhaps, seeing the first signs of progress in this direction in the form of the HEP Software Foundation. In addition, experience in the cosmology community shows the multi-lingual programming system help encourage scientists to contribute work by “lowering the bar” for such contribution. Greater support for such multi-language programming in the common event-processing frameworks might provide the same benefit for the HEP community.

References