CosmoSlik Documentation

Release 1.0.0

Marius Millea

May 30, 2018

Contents

CosmoSlik stands for **Cosmo**logy **S**ampler of **Lik**elihoods.

CosmoSlik is a simple but powerful tool to quickly put together, run, and analyze an MCMC chain for analysis of cosmological data. You can use plugins for things like CAMB, CLASS, the Planck likelihood, other cosmological likelihoods, your own customizations, emcee, etc… It has advantages in ease-of-use, flexibility, and modularity as compared to similar codes like CosmoMC, MontePython, or CosmoSIS.

- **Documentation:** cosmoslik.readthedocs.io
- **Source code:** github.com/marius311/cosmoslik
- **Citing:** http://ascl.net/code/v/1601

To get started,

- *Install* CosmoSlik
- Read through the *Quickstart* guide
- Look at the different available plugins (*Likelihoods*, *Models*, or *Samplers*)
- Run your own analysis!

Note: Many of the pages in this documentation are IPython notebooks. If you download the source code, you can run and experiment with any of the examples that you see.

CHAPTER 1

Table of Contents:

1.1 Install

CosmoSlik has the following requirements:

- Python >= 3.3
- Cython >= 0.16
- numpy >= 1.11.0
- scipy >= 0.17.0
- setuptools >= 3.2

If you don't have these, they will be installed along with CosmoSlik, although its probably best to try and install them beforehand with whatever package manager you use on your system.

To install CosmoSlik, first get the source code*1*,

```
git clone https://github.com/marius311/cosmoslik.git
cd cosmoslik
```

then run,

```
python setup.py develop --user
```

This installs CosmoSlik in-place so you can update it later by simlpy running `git pull`.

[1]: If your system doesn't have `git` installed, you can also manually download the source code from https://github.com/marius311/cosmoslik

1.2 Quickstart

1.2.1 Basic Examples

To create a CosmoSlik *script* which runs a chain, you start with the following boilerplate code, which you should put in some file, e.g. `myscipt.py`,

```
from cosmoslik import *

class myscript(SlikPlugin):
    def __init__(self):
        super().__init__()
        # your initialization code will go here

    def __call__(self):
        # you'll return negative log-likelihood here
```

This script is like the “ini” file other similar codes use to define a particular run, but is much more flexible because its just Python code which can do arbitrarily complex things, as we’ll see below.

Lets code up a simple example that samples a 2D unit Gaussian likelihood. In the initialization section we define the parameters to be sampled by the MCMC chain, and in the likelihood section we use these parameters to return a likelihood (by convention, CosmoSlik wants the negative log-likelihood). Finally, we set which MCMC sampler to use. The new script looks like this:

```
from cosmoslik import *

class myscript(SlikPlugin):
    def __init__(self):
        super().__init__()

        # define sampled parameters
        self.a = param(start=0, scale=1)
        self.b = param(start=0, scale=1)

        # set the sampler
        self.sampler = samplers.metropolis_hastings(
            self,
            num_samples=1e5,
            output_file="myscript.chain",
        )

    def __call__(self):
        # compute the likelihood
        return (self.a**2 + self.b**2)/2
```

You can now immediately run this chain from the command line by doing,

```
$ cosmoslik -n 8 myscript.py
```

The `-n 8` option runs 8 chains in parallel with MPI (you will need `mpi4py` installed) and automatically updates the proposal covariance. The resulting 8 chains are all written to the single file specified in the script, in our case `myscript.chain`. You can read the chain file (including while the chain is running to see its progress) via

```
>>> chain = load_chain("myscript.chain")
```

Alternatively, you could run this script from an interactive session and get a chain directly via

```
>>> chain = run_chain("myscript.py", nchains=8)
```

Or you could have skipped the script file entirely and just run

```
>>> chain = run_chain(myscript, nchains=8)
```

assuming you defined `myscript` in your interactive session.

`load_chain` and `run_chain` return `Chain` or `Chains` objects which have a number of useful methods for manipulating, post-processing, and plotting results. For example, since we ran 8 chains, we might want to concatenate them into a single chain, first burning-off some samples at the beginning of each chain, and then create a "triangle" plot from the result. This would be done with,

```
>>> chain.burnin(500).join().likegrid()
```

1.2.2 Script options and flexibility

The power in using Python to define scripts is that we can do lots of advanced things that can't be done in "ini" files or other mini-languages. This means a single script can in fact be a powerful multi-purpose tool that runs several different chains. For example, we can decide on the fly how many parameters to sample. Consider the following script which samples an `ndim`-dimensional Gaussian where `ndim` is a parameter we will pass in,

```
from cosmoslik import *

class myscript(SlikPlugin):
    def __init__(self, ndim, num_samples=1000):
        super().__init__()
        # save the ndim parameter so we can use it below in __call__ too
        ndim = self.ndim = int(ndim)

        # dynamically create the sampled parameters we need
        for i in range(ndim):
            self["param%i"%i] = param(start=0, scale=1)

        self.sampler = samplers.metropolis_hastings(
            self,
            num_samples=num_samples,
            output_file="myscript_ndim_%i.chain"%ndim
        )

    def __call__(self):
        return sum([self["param%i"%i]**2 for i in range(self.ndim)])/2
```

Let's break down some new things we did here:

- We gave the `__init__` function some arguments, `ndim` and `num_samples`
- We defined the sampled parameters for this chain dynamically with a `for` loop.
- We used `self` as a dictionary, i.e. `self["param"]` in place of `self.param`, which allows us to create the numbered parameters

Additionally, simply by having written this script, CosmoSlik creates a nice wrapper you can use to call this script from the command line and specify parameters. You can see the "help" for it via:

```
$ cosmoslik myscript.py -h
usage: cosmoslik myscript.py [-h] [--num_samples NUM_SAMPLES] ndim

positional arguments:
  ndim

optional arguments:
  -h, --help            show this help message and exit
  --num_samples NUM_SAMPLES
                        default: 1000
```

You can then run e.g. a a 10-dimensional chain with 10000 steps via:

```
$ cosmoslik myscript.py 10 --num_samples 10000
```

1.2.3 Cosmological Examples

Having seen the basic machinery of how to write CosmoSlik scripts and run them, lets see how to run a real cosmology chain. As an example, we'll run a Planck chain, using CAMB to compute the CMB Cl's. The script file looks like this,

```
class planck(SlikPlugin):

    def __init__(self):
        super().__init__()

        # define sampled cosmological parameters
        param = param_shortcut('start','scale')
        self.cosmo = SlikDict(
            logA  = param(3.108,   0.03),
            ns    = param(0.962,   0.006),
            ombh2 = param(0.02221, 0.0002),
            omch2 = param(0.1203,  0.002),
            theta = param(0.0104,  0.00003),
            tau   = param(0.055,   0.01),
        )

        # sample the Planck calibration as well
        self.calPlanck = param(1,0.0025,gaussian_prior=(1,0.0025))

        # load CAMB to compute Cls
        self.camb = models.camb()

        # load the Planck likelihood files
        self.highlTT = likelihoods.clik(clik_file='plik_lite_v18_TT.clik')
        self.lowlTT = likelihoods.clik(clik_file='commander_rc2_v1.1_l2_29_B.clik')

        self.sampler = samplers.metropolis_hastings(
            self,
            num_samples = 1e7,
            output_file = 'planck.chain',
        )


    def __call__(self):
        # we sample logA but CAMB needs As
        self.cosmo.As = exp(self.cosmo.logA)*1e-10
```

(continues on next page)

(continued from previous page)

```
        # compute Cls
        self.cls = self.camb(**self.cosmo)

        # the two Planck likelihoods read the calibration from `A_Planck` and
        # `A_planck`, so set those based on our sampled parameter
        self.highlTT.A_Planck = self.lowlTT.A_planck = self.calPlanck

        # compute likelihood
        return lsum(
            lambda: self.highlTT(self.cls),
            lambda: self.lowlTT(self.cls)
        )
```

Some new things here are:

- "Attaching" sampled parameters not directly to `self` but to a sub-attribute, in this case, `self.cosmo`. CosmoSlik will find all sampled parameters if they are attached to any `SlikDicts` attached to `self` (recursively, any number of `SlikDicts` deep). You can use this to organize parameters into convenient subgroups.
- Using the `lsum` function. This is a convenience function which simply sums up its arguments in order, but if one of them returns `inf`, it doesn't waste time evaluating the rest.

1.3 Likelihoods

1.3.1 Planck (via `clik`)

This plugin is an interface between the Planck likelihood code `clik` and CosmoSlik. You need `clik` already installed on your machine, which you can get from here.

You also need to download the "`clik` files" for whichever likelihoods you would like to use. You can find these here under "Likelihoods" / "Notes".

1.3.1.1 Quickstart

CosmoSlik provides several plugins which wrap `clik` and have all the necessary nuisance parameters set up for particular data files. You can use them in your script by adding something like the following to your `__init__`,

```
# set up cosmological params and solver
self.cosmo = models.cosmology("lcdm")
self.cmb = models.classy()

# load Planck clik file and set up nuisance parameters
self.clik = likelihoods.planck.planck_2015_highl_TT(
    clik_file="plik_dx11dr2_HM_v18_TT.clik/",
)
```

then compute the likelihood in `__call__` by calling `clik` with a parameter `cmb` of the kind returned by `CAMB` or `CLASS`,

```
# compute likelihood
self.clik(self.cmb(**self.cosmo))
```

1.3.1.2 The generic `clik` wrapper

Using the `SlikPlugin` named `clik`, we can load up any generic `clik` file. Supposing we've downloaded the file `plik_lite_v18_TT.clik`, we can load it in via,

```
In [1]: %pylab inline
        sys.path = sys.path[1:]
        from cosmoslik import *
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
In [2]: clik = likelihoods.planck.clik(
            clik_file="plik_lite_v18_TT.clik/",
            A_Planck=1
        )
        clik
```

```
Out[2]: {'A_Planck': 1,
         'auto_reject_errors': False,
         'clik': <clik.lkl.clik at 0x7f03a8eda510>}
```

Note that we gave it a parameter `A_Planck`. Most `clik` files have extra nuisance parameters, which you can list (for a given file) with,

```
In [3]: clik.clik.get_extra_parameter_names()
```

```
Out[3]: ('A_Planck',)
```

You should attach parametes with these names to the `clik` object as we have done above (usually in a script these will be sampled parameters).

With the `clik` object created, we can call it to compute the likelihood. The function expects a parameter `cmb` of the kind returned by `CAMB` or `CLASS`.

```
In [4]: cmb = models.classy(lmax=3000)()
        cmb
```

```
Out[4]: {'BB': array([  0.00000000e+00,   0.00000000e+00,   1.77707779e-06, ...,
                  1.41341956e-02,   1.41160893e-02,   1.40979982e-02]),
         'EE': array([ 0.        ,  0.        ,  0.05413355, ...,  0.92829409,
                 0.92452331,  0.92077857]),
         'PP': array([  0.00000000e+00,   0.00000000e+00,   6.55525164e+04, ...,
                  4.37564298e-04,   4.36858020e-04,   4.36153030e-04]),
         'TE': array([ 0.        ,  0.        ,  3.57593976, ..., -1.57751022,
                -1.57159761, -1.56562643]),
         'TP': array([  0.00000000e+00,   0.00000000e+00,   3.60597308e+03, ...,
                  4.68131564e-05,   4.67227124e-05,   4.66374675e-05]),
         'TT': array([    0.        ,     0.        ,  1110.41116627, ...,    26.09287144,
                   26.04596539,    25.99892254])}
```

Here's the negative log likelihood:

```
In [5]: clik(cmb)
```

```
Out[5]: 201.12250756838722
```

Putting it all together, a simple script which runs this likelihood would look like:

```
In [6]: class planck(SlikPlugin):

            def __init__(self, **kwargs):
                super().__init__()

                # load Planck clik file and set up nuisance parameters
                self.clik = likelihoods.planck.clik(
                    clik_file="plik_lite_v18_TT.clik/",

                    # sample over nuisance parameter
                    A_Planck=param(start=1, scale=0.0025, gaussian_prior=(1,0.0025))
                )

                # set up cosmological params and solver
                self.cosmo = models.cosmology("lcdm")
                self.cmb = models.classy(lmax=3000)

                self.sampler = samplers.metropolis_hastings(self)

            def __call__(self):
                # compute likelihood
                return self.clik(self.cmb(**self.cosmo))
```

```
In [7]: s = Slik(planck())
        lnl, e = s.evaluate(**s.get_start())
        lnl
```

```
Out[7]: 956.87241321269414
```

1.3.1.3 Ready-to-go wrappers for specific `clik` files

The previous example was easy because there was one single nuisance parameter, `A_Planck`. Other `clik` files have many more nuisance parameters, which must all be sampled over and in some cases have the right priors applied (which you can read about here), otherwise you will not get the right answer.

This is, of course, a huge pain.

For this reason, CosmoSlik comes with several SlikPlugins already containing the correct sampled nuisance parameters for many of these `clik` files, making writing a script extremely easy. For example, here is the source code for one such plugin, `planck_2015_highl_TT`:

```
param = param_shortcut('start','scale')

class planck_2015_highl_TT(clik):

    def __init__(
        self,
        clik_file,
        A_cib_217         = param(60,   10,     range=(0,200)),
        A_planck          = param(1,    0.0025, range=(0.9,1.1), gaussian_prior=(1,0.
↪0025)),
        A_sz              = param(5,    3,      range=(0,10)),
        calib_100T        = param(1,    0.001,  range=(0,3),     gaussian_prior=(0.999,
↪0.001)),
        calib_217T        = param(1,    0.002,  range=(0,3),     gaussian_prior=(0.995,
↪0.002)),
        cib_index         = -1.3,
        gal545_A_100      = param(7,    2,      range=(0,50),    gaussian_prior=(7,2)),
        gal545_A_143      = param(9,    2,      range=(0,50),    gaussian_prior=(9,2)),
        gal545_A_143_217  = param(21,   8.5,    range=(0,100),   gaussian_prior=(21,8.
↪5)),
        gal545_A_217      = param(80,   20,     range=(0,400),   gaussian_prior=(80,
↪20)),
        ksz_norm          = param(2,    3,      range=(0,10)),
        ps_A_100_100      = param(250,  30,     range=(0,4000)),
```

(continues on next page)

(continued from previous page)

```
        ps_A_143_143     = param(45,  10,     range=(0,4000)),
        ps_A_143_217     = param(40,  10,     range=(0,4000)),
        ps_A_217_217     = param(90,  15,     range=(0,4000)),
        xi_sz_cib        = param(0.5, 0.3,    range=(0,1)),
    ):
        super().__init__(**arguments())
```

As you can see, all the sampled parameters as automatically set, including ranges and priors. The script to use this likelihood is then extremely simple:

```
In [8]: class planck(SlikPlugin):

            def __init__(self):
                super().__init__()

                # load Planck clik file and set up nuisance parameters
                self.clik = likelihoods.planck.planck_2015_highl_TT(
                    clik_file="plik_dx11dr2_HM_v18_TT.clik/",
                )

                # set up cosmological params and solver
                self.cosmo = models.cosmology("lcdm")
                self.cmb = models.classy(lmax=3000)

                self.sampler = samplers.metropolis_hastings(self)

            def __call__(self):
                # compute likelihood
                return self.clik(self.cmb(**self.cosmo))
```

```
In [9]: s = Slik(planck())
        lnl, e = s.evaluate(**s.get_start())
        lnl
```

```
Out[9]: 2672.4139027829988
```

1.3.1.4 Common calibration parameters

Despite that the Planck likelihood is broken up into different pieces, they sometimes share the same calibration parameters. To apply this correctly in your script, just define one single sampled calibration parameter, then in your `__call__`, set it across all the different likelihoods.

```
In [10]: class planck(SlikPlugin):

             def __init__(self):
                 super().__init__()

                 # set up low and high L likelihood
                 self.highl = likelihoods.planck.planck_2015_highl_TT(
                     clik_file="plik_dx11dr2_HM_v18_TT.clik/",
                 )
                 self.lowl = likelihoods.planck.planck_2015_lowl_TT(
                     clik_file="commander_rc2_v1.1_l2_29_B.clik/",
                     A_planck=None, #turn off this cal parameter, use the one from self.highl
                 )

                 # set up cosmological params and solver
                 self.cosmo = models.cosmology("lcdm")
```

```
            self.cmb = models.classy(lmax=3000)

            self.sampler = samplers.metropolis_hastings(self)

        def __call__(self):
            # set the calibration parameters the same
            self.lowl.A_planck = self.highl.A_planck

            # compute likelihood
            cmb = self.cmb(**self.cosmo)
            return self.lowl(cmb) + self.highl(cmb)
```

1.3.2 South Pole Telescope low-ℓ

This plugin implements the South Pole Telescope likelihood from Story et al. (2012) and Keisler et al. (2011). The data comes included with this plugin and was downloaded from here and here, respectively.

You can choose which likelihood to use by specifying `which='s12'` or `which='k11'` when you initialize the plugin.

The plugin also supports specifying an $\ell_{\min}$, $\ell_{\max}$, or dropping certain data bins.

1.3.2.1 cosmoslik_plugins.likelihoods.spt_lowl package

1.3.2.1.1 Submodules

class `cosmoslik_plugins.likelihoods.spt_lowl.spt_lowl.`**`spt_lowl`**(*which='s12'*, *lmin=None*, *lmax=None*, *drop=None*, *egfs='default'*, *cal=None*, ***kwargs*)

Bases: *`cosmoslik.cosmoslik.SlikPlugin`*

The SPT "low-L" likelihood from Keisler et al. 2011 or Story et al. 2012.

See *spt_lowl.ipynb* for some examples using this plugin.

`__call__`(*cmb*, *egfs=None*, *cal=None*)
Compute the likelihood.

Parameters

- **cmb** (*dict*) – A dict which has a key "TT" which holds the CMB Dl's in muK^2
- **egfs/cal** – Override whatever (if anything) was set in *`__init__()`*. See *`__init__()`* for documentation on these arguments.

`__init__`(*which='s12'*, *lmin=None*, *lmax=None*, *drop=None*, *egfs='default'*, *cal=None*, ***kwargs*)

Parameters

- **which** – 'k11' or 's12'
- **lmin/lmax** (*int*) – restrict the data to this ℓ-range
- **drop** (*slice, int, or array of ints*) – remove the bins at these indices

- **cal** (*float*) – calibration parameter, defined to multiply the data power spectrum. (only available if which='s12')
- **egfs** (*callable, None, or 'default'*) – A callable object which will be called to compute the extra-galactic foreground model. It should accept keyword arguments *lmax* which specifies the length of the array to reuturn, and *egfs_specs* which contains information like frequency and fluxcut needed for more sophisticated foreground models. If *egfs* is None, its assumed it will be set later (you must do so before computing the likelihood). If 'default' is specified, the default foreground model will be used (see `spt_lowl_egfs`)

plot (*ax=None*, *residuals=False*, *show_comps=False*)
Plot the data (multplied by calibration) and the model.

Calibration and model taken from previous call to `__call__()`.

Parameters

- **ax** (*axes*) – matplotlib axes object
- **residual** (*bool*) – plot residuals to the model
- **show_comps** (*bool*) – plot CMB and egfs models separately

class cosmoslik_plugins.likelihoods.spt_lowl.spt_lowl.**spt_lowl_egfs** (*Asz=<cosmoslik.cosmoslik.param object>*, *Acl=<cosmoslik.cosmoslik.param object>*, *Aps=<cosmoslik.cosmoslik.param object>*, ***kwargs*)

Bases: `cosmoslik.cosmoslik.SlikPlugin`

The default foreground model for the SPT low-L likelihood from Story/Keisler et al. which features (tSZ+kSZ), CIB clustering, and CIB Poisson templates, with free amplitudes and priors on these amplitudes.

__call__ (*lmax*, ***_*)
Calculation code here.

__init__ (*Asz=<cosmoslik.cosmoslik.param object>*, *Acl=<cosmoslik.cosmoslik.param object>*, *Aps=<cosmoslik.cosmoslik.param object>*, ***kwargs*)
Initialization code here.

1.3.2.1.2 Module contents

```
In [1]: %pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
In [2]: from cosmoslik import *
```

1.3.2.2 Basic Script

Here's a script which runs a basic SPT-only chain:

```
In [3]: class spt(SlikPlugin):

            def __init__(self, **kwargs):
                super().__init__()
                self.cosmo = models.cosmology("lcdm")
                self.spt_lowl = likelihoods.spt_lowl(**kwargs)
                self.cmb = models.camb(lmax=4000)
                self.sampler = samplers.metropolis_hastings(self)

            def __call__(self):
                return self.spt_lowl(self.cmb(**self.cosmo))
```

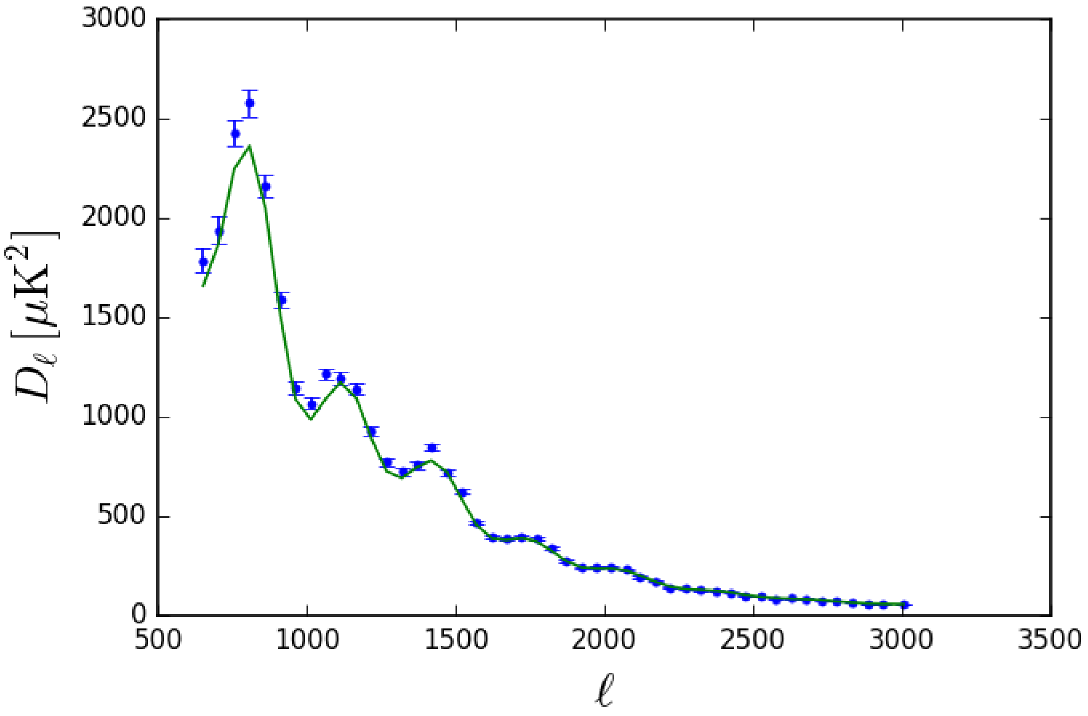
Four sampled parameters, one for calibration and three for foregrounds, come by default with the SPT plugin:

```
In [4]: spt().spt_lowl.find_sampled().keys()
```

```
Out[4]: odict_keys(['cal', 'egfs.Acl', 'egfs.Aps', 'egfs.Asz'])
```

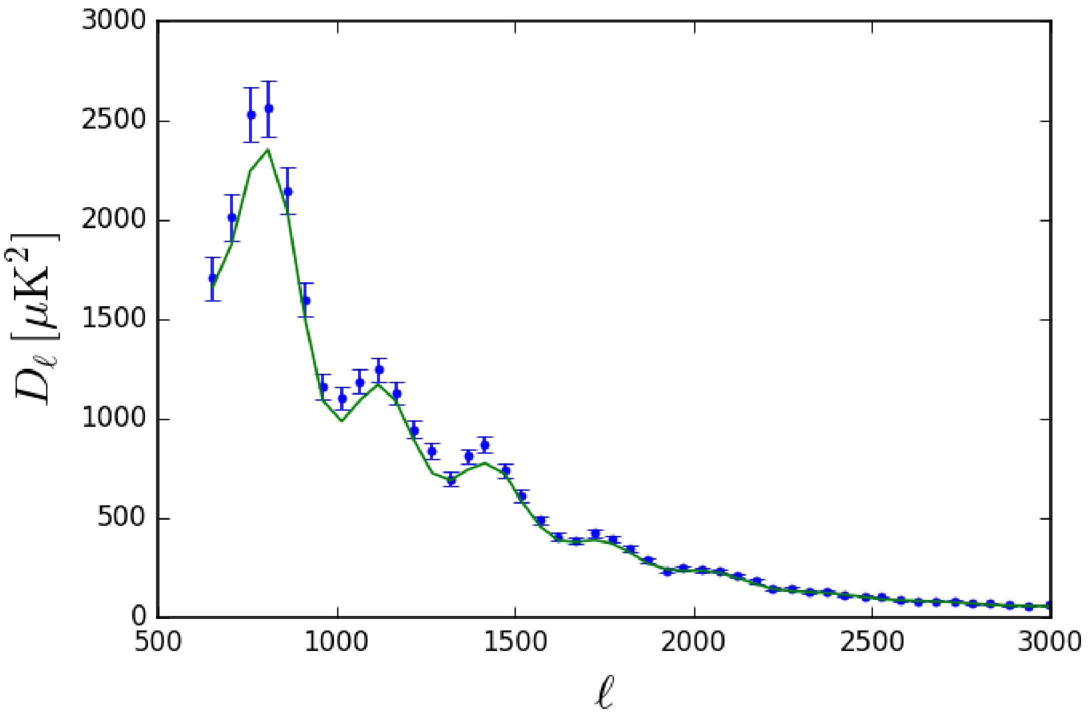
The plugin has a convenience method for plotting the data and current model:

```
In [5]: s = Slik(spt(which='s12'))
        lnl, e = s.evaluate(**s.get_start())
        e.spt_lowl.plot()
```



Or you can plot the "k11" bandpowers. You can see they're less constraining.

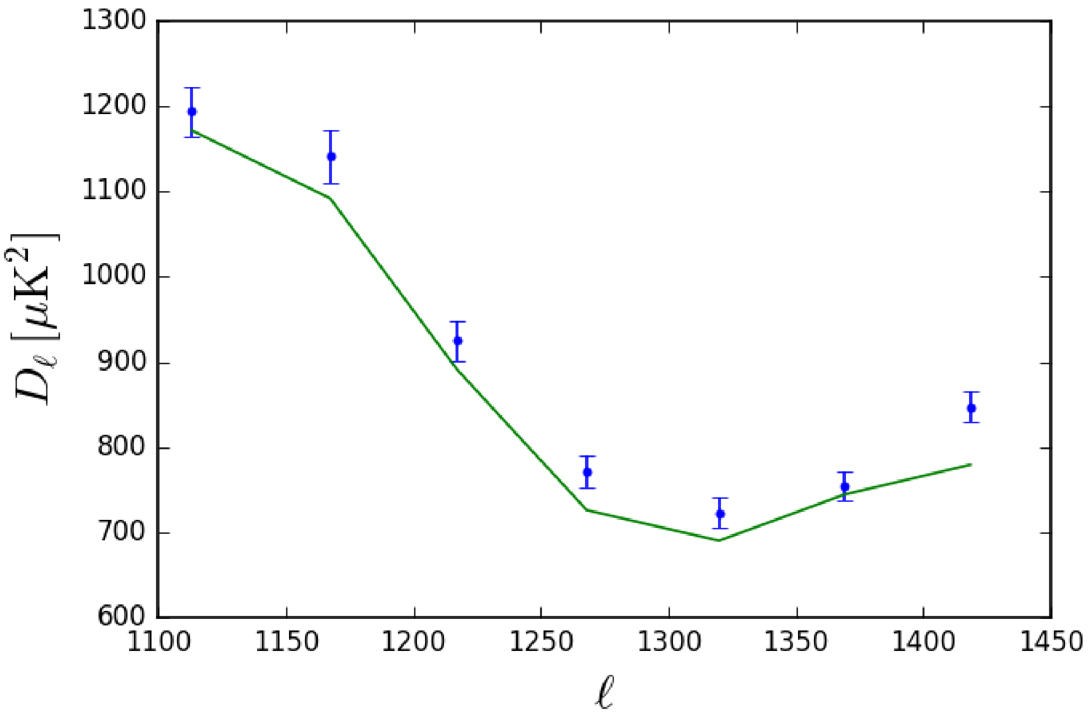
```
In [6]: s = Slik(spt(which='k11'))
        lnl, e = s.evaluate(**s.get_start())
        e.spt_lowl.plot()
```



1.3.2.3 Choosing subsets of data

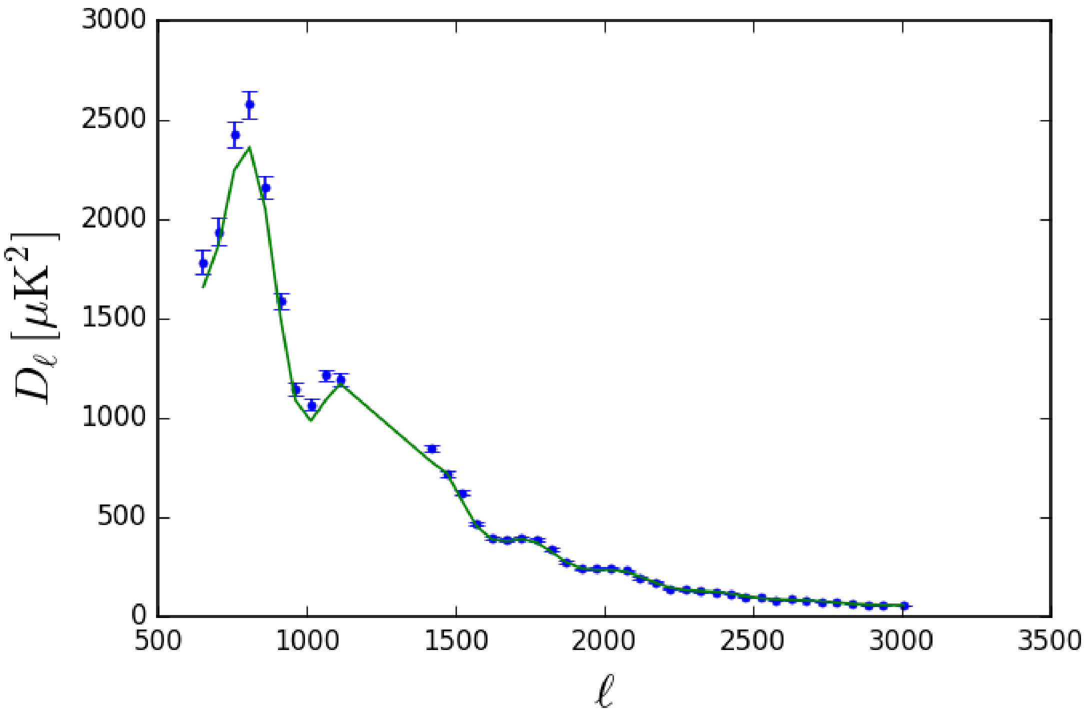
You can also set some ℓ-limits,

```
In [7]: s = Slik(spt(which='s12',lmin=1000,lmax=1500))
        lnl, e = s.evaluate(**s.get_start())
        e.spt_lowl.plot()
```



Or drop some individual data points (by bin index),

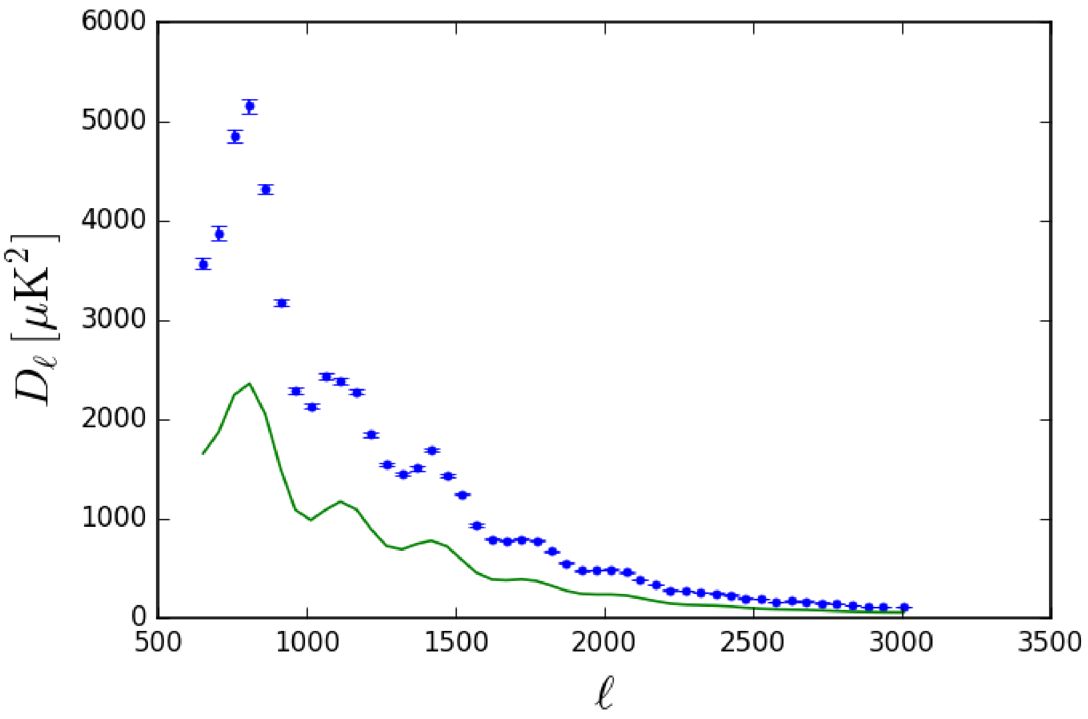
```
In [8]: s = Slik(spt(which='s12',drop=range(10,15)))
        lnl, e = s.evaluate(**s.get_start())
        e.spt_lowl.plot()
```



1.3.2.4 Calibration parameter

The calibration parameter is called `cal` and defined so it multiplies the data bandpowers. For "s12" it comes by default with a prior 1 ± 0.026. You can't use it for "k11" because this likelihood has the calibration pre-folded into the covariance.

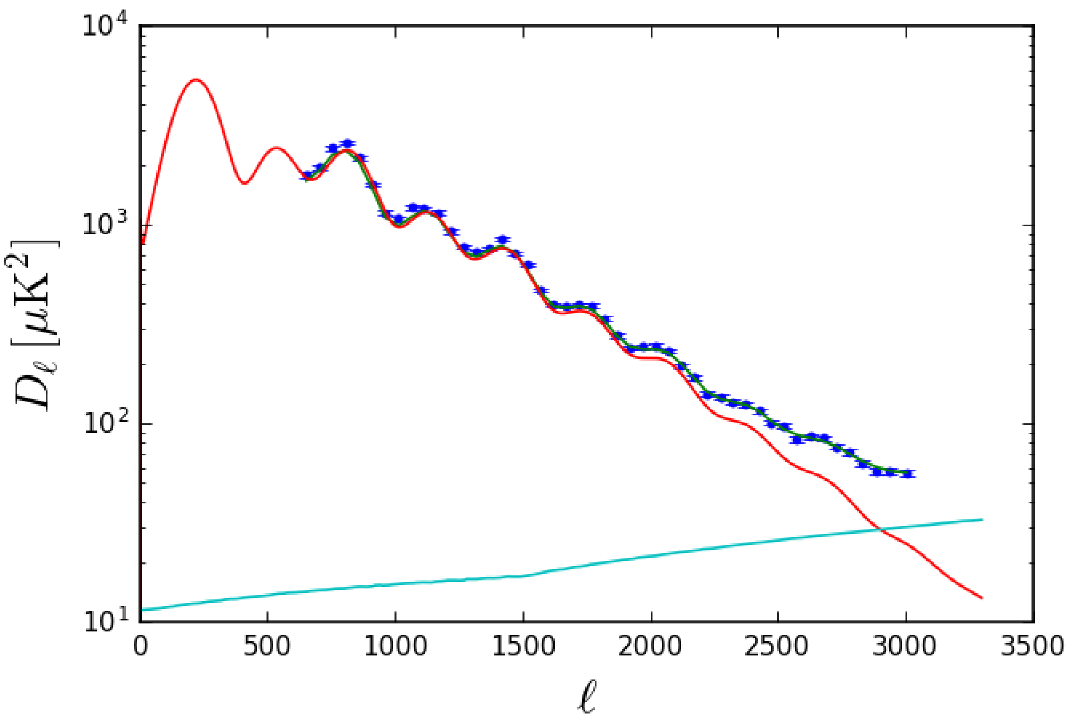
```
In [9]: s = Slik(spt(which='s12',cal=2))
        lnl, e = s.evaluate(**s.get_start())
        e.spt_lowl.plot()
```



1.3.2.5 Foreground model

By default the foreground model is the one used in the Story/Keisler et al. papers (same for both). There's an option to `plot` which shows you the CMB and foreground components separately so you can see it.

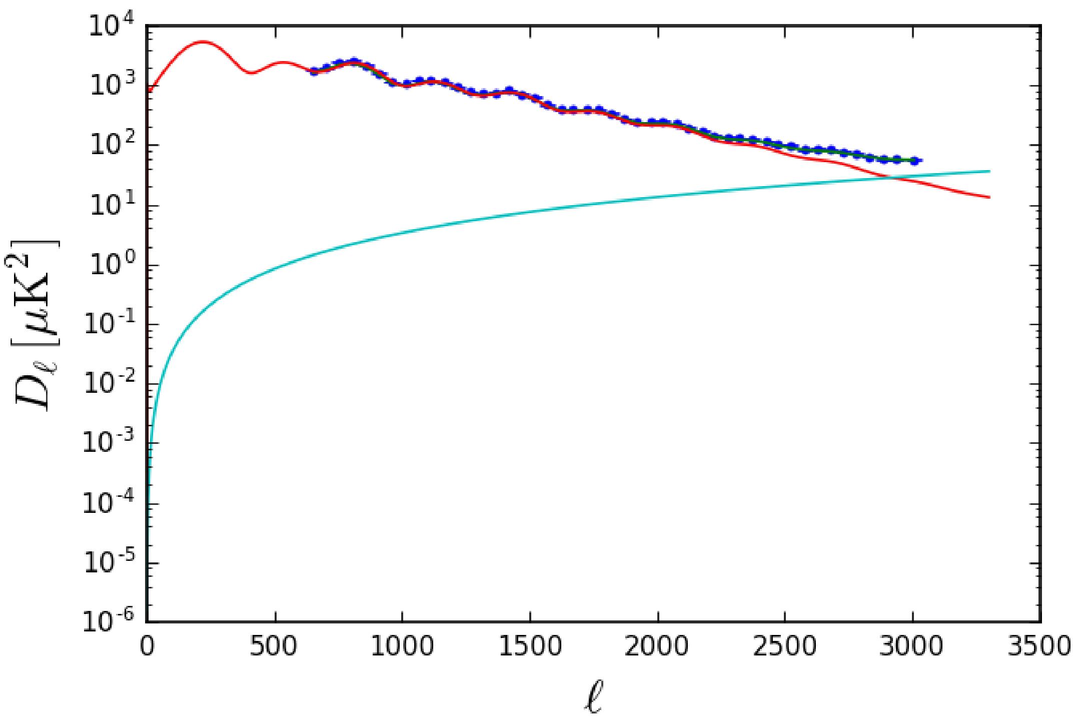
```
In [10]: s = Slik(spt(which='s12'))
         lnl, e = s.evaluate(**s.get_start())
         e.spt_lowl.plot(show_comps=True)
         yscale('log')
```



The foreground model is taken from the `spt_lowl.egfs` attribute which is expected to be a function which can be called with parameters `lmax` to specify the length of the array returned, and `egfs_specs` which provides some info about frequencies/fluxcut of the SPT bandpowers for more advanced foreground models. You can customize this by attaching your own callable function to `spt_lowl.egfs` when you call `__init__`, or passing something in during `__call__`.

For example, say we wanted a Poisson-only foreground model, we could write the script like so:

```
In [11]: class spt_myfgs(SlikPlugin):

             def __init__(self):
                 super().__init__()
                 self.cosmo = models.cosmology("lcdm")
                 self.spt_lowl = likelihoods.spt_lowl(egfs=None) #turn off default model & params
                 self.Aps = param(start=30, scale=10, min=0) #add our own sampled parameter here
                 self.cmb = models.camb(lmax=4000)
                 self.sampler = samplers.metropolis_hastings(self)

             def __call__(self):
                 return self.spt_lowl(self.cmb(**self.cosmo),
                                      egfs=lambda lmax,**_: self.Aps * (arange(lmax)/3000.)**2) #compu

In [12]: s = Slik(spt_myfgs())
         lnl, e = s.evaluate(**s.get_start())
         e.spt_lowl.plot(show_comps=True)
         yscale("log")
```



1.4 Models

1.4.1 cosmo_derived

This plugin calculates the following "derived" cosmological quantities: * $H(z)$ * $D_A(z)$ * $r_s(z)$ * $\theta_s(z)$ * z_{drag} (Hu & Sugiyama fitting formula)

It can also be used as a θ to H_0 converter.

Credit: Lloyd Knox, code adapted by Marius Millea

```
In [1]: from cosmoslik import *
```

```
In [2]: cosmo_derived = get_plugin('models.cosmo_derived')()
```

To use cosmo_derived, first call the set_params function to set all the cosmological parameters, then call the other functions which will subsequently use those values. Here's the signature for set_params:

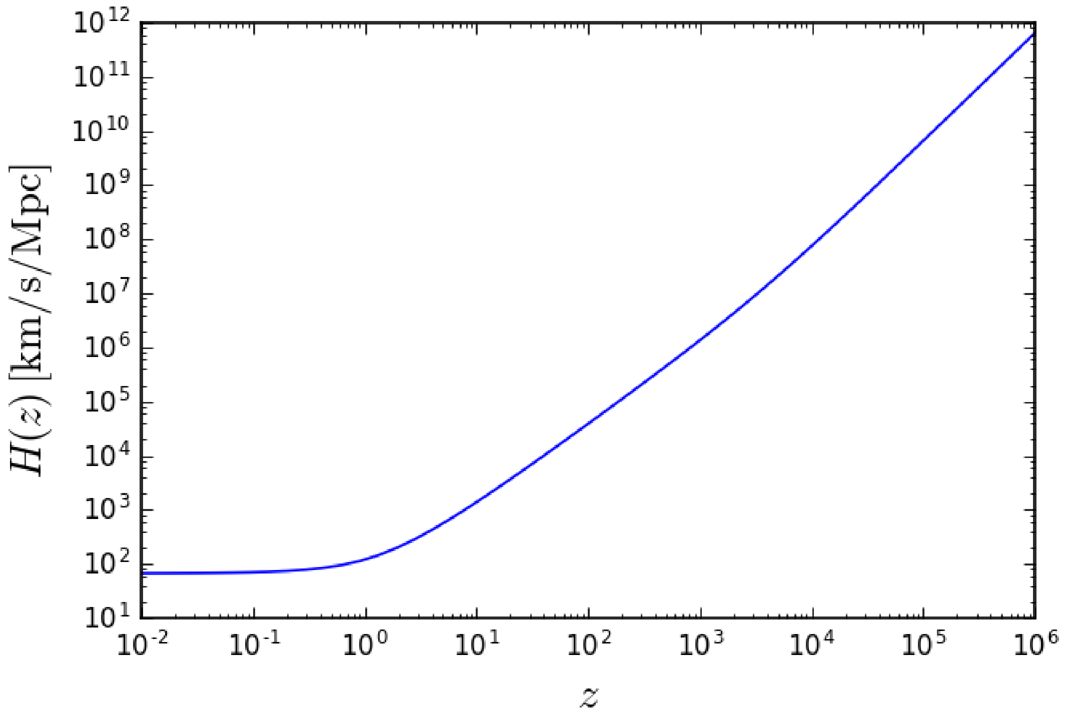
```
In [3]: help(cosmo_derived.set_params)
```

```
Help on built-in function set_params:

set_params(...) method of cosmoslik_plugins.models.cosmo_derived.cosmo_derived._cosmo_derived instanc
    _cosmo_derived.set_params(self, H0=None, theta_mc=None, ombh2=None, omch2=None, omk=None, mnu=Nor

    Args:
        H0 : hubble constant today in km/s/Mpc
        ombh2, omch2, omk : densities today
        mnu : neutrino mass sum in eV
        massive_neutrinos : number of massive species (mnu divided equally among them)
        massless_neutrinos : number of massless species
        Yp : helium mass fraction
        Tcmb : CMB temperature in Kelvin
        theta_mc : if given, will convert to H0 and set H0 to that
```

```
In [4]: cosmo_derived.set_params(H0=67.04, ombh2=0.022032, omch2=0.12038, omk=0, mnu=0.06, massive_ne
```

Plotting $H(z)$,

```
In [8]: z=logspace(-2,6)
        loglog(z,list(map(cosmo_derived.Hubble,z)))
        xlabel(r'$z$',size=16)
        ylabel(r'$H(z) \, [\rm km/s/Mpc]$',size=16);
```



Here's the angular size of sound horizon at Planck's best-fit z_* (Table 2, Planck XVI). The number for θ_s in that same table is 0.0104136, or a difference of 0.09σ. This is likely due to differences in numerical values for physical constants that were used, or numerical integration error.

```
In [9]: z_star = 1090.48
        cosmo_derived.theta_s(z_star)
```

```
Out[9]: 0.010414141562032136
```

We can also use this plugin to convert θ to H_0. Here θ refers to $\theta_{\rm MC}$ which uses the Hu & Sugiyama fitting formula for $z_{\rm drag}$.

```
In [10]: cosmo_derived.theta2hubble(0.0104)
```

```
Out[10]: 66.95260969910498
```

This plugin is written in Cython and is highly optimizied, so its pretty fast.

```
In [11]: %%timeit
         cosmo_derived.theta_s(z_star)
```

```
100 loops, best of 3: 9.16 ms per loop
```

```
In [12]: %%timeit
         cosmo_derived.theta2hubble(0.0104)
```

```
10 loops, best of 3: 56 ms per loop
```

The θ_s calculation about 7 times slower than the equivalent Fortran code used in CosmoMC. The theta to hubble conversion is about 3 times **faster** than the CosmoMC version because the equation solver is more sophisticated and requires fewer θ_s evaluations. Not bad for being very readable and modifiable Cython code instead. As an example, here's what the code looks like:

```
def r_s(self, double z):
    """
    Returns : comoving sound horizon scale at redshift z in Mpc.
    """
    cdef double Rovera=3*self.ombh2*rhoxOverOmegaxh2/(4*self.rhogamma0)
    return quad(lambda double zp: 1/self.Hubble(zp) / sqrt(3*(1+Rovera/(1+zp))),z,inf,
↪epsabs=0,epsrel=self.epsrel)[0] / KmPerSOverC
```

1.5 Samplers

1.6 API Reference

1.6.1 cosmoslik package

1.6.1.1 Subpackages

1.6.1.1.1 cosmoslik.mpi package

Submodules

`cosmoslik.mpi.mpi.`**`flatten`**`(`*l*`)`
Returns a list of lists joined into one

`cosmoslik.mpi.mpi.`**`get_mpi`**`()`
Return (rank,size,comm)

`cosmoslik.mpi.mpi.`**`get_pool`**`()`
Gets a *pool* object which can be passed to things that expect it to have a *pool.map* function.

`cosmoslik.mpi.mpi.`**`mpi_consistent`**`(`*value*`)`
Returns the value that the root process provided.

`cosmoslik.mpi.mpi.`**`mpi_map`**`(`*function*, *sequence*, *distribute=True*`)`
A map function parallelized with MPI. If this program was called with mpiexec -n $NUM, then partitions the sequence into $NUM blocks and each MPI process does the rank-th one. Note: If this function is called recursively, only the first call will be parallelized

Keyword arguments: distribute – If true, every process receives the answer

otherwise only the root process does (default=True)

`cosmoslik.mpi.mpi.`**`partition`**`(`*l*, *n*`)`
Partition l into n nearly equal sublists

Module contents

1.6.1.2 Submodules

Python module for dealing with MCMC chains.

Contains utilities to:

- Load chains in a variety of formats
- Compute statistics (mean, std-dev, conf intervals, . . .)
- Plot 1- and 2-d distributions of one or multiple parameters.
- Post-process chains

class `cosmoslik.chains.`**`Chain`**(**args*, ***kwargs*)

Bases: `dict`

An MCMC chain. This is just a Python dictionary mapping parameter names to arrays of values, along with the special keys 'lnl' and 'weight'

`__init__`(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

`__repr__`()
Return repr(self).

`__str__`()
Print a summary of the chain.

`__weakref__`
list of weak references to the object (if defined)

`acceptance`()
Returns the acceptance ratio.

`add_gauss_prior`(*params*, *mean*, *covstd*, *nthreads=1*, *pool=None*)
Post-process a gaussian prior into the chain.

Parameters

- **`- a parameter name, or a list of parameters`** (*`params`*) –
- **`- the mean`** (*`mean`*) –
- **`- if params was a list, this should be a 2-d array holding the covariance`** (*`covstd`*) – if params was a single parameter, this should be the standard devation

Returns A new chain, without altering the original chain

`best_fit`()
Get the best fit sample.

`burnin`(*n*)
Remove the first n non-unique samples from the beginning of the chain.

`confbound`(*param*, *level=0.95*, *bound=None*)
Compute an upper or lower confidence bound.

Parameters

- **`param`** (*`string`*) – name of the parameter

- **level** (*float*) – confidence level
- **bound** (*'upper', 'lower', or None*) – whether to compute an upper or lower confidence bound. If set to None, will guess which based on the skewness of the distribution (will be lower bound for positively skewed distributions)

copy()
Deep copy the chain so post-processing, etc… works right

corr(*params=None*)
Returns the correlation matrix of the parameters (or some subset of them) in this chain.

cov(*params=None*)
Returns the covariance of the parameters (or some subset of them) in this chain.

iterrows()
Iterate over the samples in this chain.

join()
Does nothing since already one chain.

length(*unique=True*)
Returns the number of unique samples. Set unique=False to get total samples.

like1d(*p*, ***kwargs*)
Plots 1D likelihood contours for a parameter. See `like1d()`

like2d(*p1*, *p2*, ***kwargs*)
Plots 2D likelihood contours for a pair of parameters. See `like2d()`

likegrid(***kwargs*)
Make a grid (aka "triangle plot") of 1- and 2-d likelihood contours. See *`likegrid()`*

likegrid1d(***kwargs*)
Make a grid of 1-d likelihood contours. See *`likegrid1d()`*

likepoints(**args*, ***kwargs*)
Plots samples from the chain as colored points. See *`likepoints()`*

matrix(*params=None*)
Return this chain as an nsamp * nparams matrix.

mean(*params=None*)
Returns the mean of the parameters (or some subset of them) in this chain.

params(*non_numeric=False*, *fixed=False*)
Returns the parameters in this chain (i.e. the keys except 'lnl' and 'weight') :param numeric: whether to include non-numeric parameters (default: False) :param fixed: whether to include parameters which don't vary (default: False)

plot(*param=None*, *ncols=5*, *fig=None*, *size=4*)
Plot the value of a parameter as a function of sample number.

postprocd(*func*, *nthreads=1*, *pool=None*)
Post-process some values into this chain.

Parameters

- **func** – a function which accepts all the keys in the chain and returns a dictionary of new keys to add. *func* must accept *all* keys in the chain, if there are ones you don't need, capture them with **_ in its call signature, e.g. to add in a parameter 'b' which is 'a' squared, use postprocd(lambda a,**_: {'b':a**2})
- **nthreads** – the number of threads to use

- **pool** – any worker pool which has a pool.map function. default: multiprocessing.Pool(nthreads)

Returns A new chain with the new values post-processed in. Does not alter the original chain. If for some rows in the chain *func* did not return all the keys, these will be filled in with *nan*.

Note: This repeatedly calls *func* on rows in the chain, so its very inneficient if you already have a vectorized version of your post-processing function. *postprocd* is mostly useful for slow non-vectorized post-processing functions, allowing convenient use of the *nthreads* option to this function.

For the default implementation of *pool*, *func* must be picklable, meaning it must be a module-level function.

---

reweighted (*func*, *nthreads=1*, *pool=None*)
Reweight this chain.

Parameters

- **func** – a function which accepts all keys in the chain, and returns a new weight for the step. *func* must accept *all* keys, if there are ones you don't need, capture them with **_ in its call signature, e.g. to add unit gaussian prior on parameter 'a' use reweighted(lambda a,**_: exp(-a**2/2)
- **nthreads** – the number of threads to use
- **pool** – any worker pool which has a pool.map function. default: multiprocessing.Pool(nthreads)

Returns A new chain, without altering the original chain

Note: This repeatedly calls *func* on rows in the chain, so its very inneficient compared to a vectorized version of your post-processing function. *postprocd* is mostly useful for slow post-processing functions, allowing you to conveniently use the *nthreads* option to this function.

For the default implementation of *pool*, *func* must be picklable, meaning it must be a module-level function.

---

sample (*s*, *keys=None*)
Return a sample or a range of samples depending on if s is an integer or a slice object.

savechain (*file*, *params=None*)
Write the chain to a file where the first line is specifies the parameter names.

savecov (*file*, *params=None*)
Write the covariance to a file where the first line is specifies the parameter names.

skew (*params=None*)
Return skewness of one or more parameters.

std (*params=None*)
Returns the std of the parameters (or some subset of them) in this chain.

thin (*delta*)
Take every delta non-unique samples.

thinto (*num*)
Thin so we end up with *num* total samples

class `cosmoslik.chains.`**Chains**
Bases: `list`

A list of chains, e.g. from a run of several parallel chains

`__repr__`()
Return repr(self).

`__str__`()
Print a summary of the chain.

`__weakref__`
list of weak references to the object (if defined)

`burnin`(*n*)
Remove the first n samples from each chain.

`join`()
Combine the chains into one.

`plot`(*param=None*, *fig=None*, ***kwargs*)
Plot the value of a parameter as a function of sample number for each chain.

`cosmoslik.chains.`**`likegrid`**(*chains*, *params=None*, *lims=None*, *ticks=None*, *nticks=4*, *nsig=4*, *spacing=0.05*, *xtick_rotation=30*, *colors=None*, *filled=True*, *nbins1d=30*, *smooth1d=False*, *kde1d=True*, *nbins2d=20*, *labels=None*, *fig=None*, *size=2*, *legend_loc=None*, *param_name_mapping=None*, *param_label_size=None*)

Make a grid (aka "triangle plot") of 1- and 2-d likelihood contours.

Parameters

- **`chains`** – one or a list of *Chain* objects
- **`optional`** (*`nbins2d,`*) – the chain used to get default parameters names, axes limits, and ticks either an index into chains or a *Chain* object (default: chains[0])
- **`optional`** – list of parameter names which to show (default: all parameters from default_chain)
- **`optional`** – a dictionary mapping parameter names to (min,max) axes limits (default: +/- 4 sigma from default_chain)
- **`optional`** – a dictionary mapping parameter names to list of [ticks] (default: automatically picks *nticks*)
- **`optional`** – roughly how many ticks per axes (default: 5)
- **`optional`** – numbers of degrees to rotate the xticks by (default: 30)
- **`optional`** – space in between plots as a fraction of figure width (default: 0.05)
- **`optional`** – figure of figure number in which to plot (default: figure(0))
- **`optional`** – size in inches of one plot (default: 2)
- **`optional`** – colors to cycle through for plotting
- **`optional`** – whether to fill in the contours (default: True)
- **`optional`** – list of names for a legend
- **`optional`** – (x,y) location of the legend (coordinates scaled to [0,1])
- **`optional`** – number (or len(chains) length list) of bins for 1d plots (default: 30)
- **`optional`** – number (or len(chains) length list) of bins for 2d plots (default: 20)

`cosmoslik.chains.`**`likegrid1d`**(*chains*, *params='all'*, *lims=None*, *ticks=None*, *nticks=4*, *nsig=3*, *colors=None*, *nbins1d=30*, *smooth1d=False*, *kde1d=True*, *labels=None*, *fig=None*, *size=2*, *aspect=1*, *legend_loc=None*, *linewidth=1*, *param_name_mapping=None*, *param_label_size=None*, *tick_label_size=None*, *titley=1*, *ncol=4*, *axes=None*)

Make a grid of 1-d likelihood contours.

chains : one or a list of *Chain* objects

default_chain, optional : the chain used to get default parameters names, axes limits, and ticks either an index into chains or a *Chain* object (default: chains[0])

params, optional : list of parameter names which to show can also be 'all' or 'common' which does the union/intersection of the params in all the chains

lims, optional : a dictionary mapping parameter names to (min,max) axes limits (default: +/- 4 sigma from default_chain)

ticks, optional : a dictionary giving a list of ticks for each parameter

nticks, optional : roughly how many x ticks to show. can be dictionary to specify each parameter separately. (default: 4)

fig, optional : figure of figure number in which to plot (default: new figure)

ncol, optional : the number of colunms (default: 4)

axes, optional : an array of axes into which to plot. if this is provided, fig and ncol are ignored. must have len(axes) >= len(params).

size, optional : size in inches of one plot (default: 2)

aspect, optional : aspect ratio (default: 1)

colors, optional : colors to cycle through for plotting

filled, optional : whether to fill in the contours (default: True)

labels, optional : list of names for a legend

legend_loc, optional : (x,y) location of the legend (coordinates scaled to [0,1])

nbins1d, optional : number of bins for 1d plots (default: 30)

nbins2d, optional : number of bins for 2d plots (default: 20)

`cosmoslik.chains.`**`likepoints`**(*chain*, *p1*, *p2*, *pcolor*, *npoints=1000*, *cmap=None*, *nsig=3*, *clim=None*, *marker='.'*, *markersize=10*, *ax=None*, *zorder=-1*, *cbar=True*, *cax=None*, ***kwargs*)

Plot p1 vs. p2 as points colored by the value of pcolor.

Parameters

- **`p1,p2,pcolor`** – parameter names
- **`npoints`** – first thins the chain so this number of points are plotted
- **`cmap`** – a colormap (default: jet)
- **`nsig`** – map the range of the color map to +/- nsig
- **`ax`** – axes to use for plotting (default: current axes)
- **`cbar`** – whether to draw a colorbar
- **`cax`** – axes to use for colorbar (default: steal from ax)

- **markersize, zorder, **kwargs** (*marker,*) – passed to the plot() command

`cosmoslik.chains.`**`load_chain`**(*filename*, *repack=False*)

Load a chain produced by a compatible CosmoSlik sampler like metropolis_hastings or emcee.

repack : If the chain file is not currently open (i.e. the chain is not currently running), and if the chain is stored in chunks as output from an MPI run, then overwrite the file with a more efficiently stored version which will be faster to load the next time.

`cosmoslik.chains.`**`load_cosmomc_chain`**(*path*, *paramnames=None*)

Load a chain from a file or files in a variety of different formats.

If `path` is a CosmoSlik ini, the read the `output_file` key from the ini and load that chain.

If `path` is a file, return a *`Chain`* object. The names of the parameters are expected either as a whitespace-separated comment on the first line of the file (CosmoSlik style) or in a separate file called <path>.paramnames (CosmoMC style).

If `path` is a directory, assumes it contains one file for each parameter (WMAP style) and gets the parameter name from the file name.

If `path` is a prefix such that there exists <path>_1, <path>_2, etc… then returns a *`Chains`* object which is a list of chains.

`cosmoslik.cosmoslik.`**`load_script`**(*script*)

Read a CosmoSlik script from a file and return the *`SlikPlugin`* object.

class `cosmoslik.cosmoslik.`**`SlikDict`**(**args*, ***kwargs*)

Bases: `dict`

`__contains__`(*k*)

True if D has a key k, else False.

`__getitem__`(*k*)

x.__getitem__(y) <==> x[y]

`__init__`(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

`__setitem__`(*k*, *v*)

Set self[key] to value.

`__weakref__`

list of weak references to the object (if defined)

`get`(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

class `cosmoslik.cosmoslik.`**`SlikPlugin`**(**args*, ***kwargs*)

Bases: *`cosmoslik.cosmoslik.SlikDict`*

`__call__`(**args*, ***kwargs*)

Calculation code here.

`__init__`(**args*, ***kwargs*)

Initialization code here.

class `cosmoslik.cosmoslik.`**`SlikSampler`**(**args*, ***kwargs*)

Bases: *`cosmoslik.cosmoslik.SlikDict`*

class `cosmoslik.cosmoslik.`**`param`**(***kwargs*)

Bases: `object`

Marks a parameter to be sampled by the MCMC.

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__weakref__
list of weak references to the object (if defined)

`cosmoslik.cosmoslik.`**param_shortcut**(**args*)
Create a *param* constructor which splices in keyword arguments for you.

Example:

```
param = param_shortcut('start','scale')
param(1,2)
> {'start':1, 'scale':2}
```

`cosmoslik.cosmoslik.`**get_all_plugins**(*ignore_import_errors=True*)
Search recursively through the namespace package *cosmoslik_plugins* for any ‘SlikPlugin‘s.

Returns keys are the shortest qualified name for the object and values are a type object.

Return type dict

`cosmoslik.cosmoslik.`**lsum**(**args*)
If your likelihood function is the sum of a bunch of others, you can do:

```
def __call__(self):
    return lsum(lambda: myfunc1(), lambda: myfunc2())
```

This returns the sum `myfunc1()+myfunc2()`, but never evaluates `myfunc2()` if `myfunc1()` returns `inf`.

See also *lsumk()* to automatically store the results of `myfunc1/2`.

`cosmoslik.cosmoslik.`**lsumk**(*lnls, args*)
If your likelihood function is the sum of a bunch of others, you can do:

```
def __call__(self):
    self['lnls']={}
    return lsumk(self['lnls'], [('key1',lambda: myfunc1()),
                                ('key2',lambda: myfunc2())])
```

This returns the sum `myfunc1()+myfunc2()`, but never evaluates `myfunc2()` if `myfunc1()` returns *inf*. It also stores the result `myfunc1/2` to `lnls['key1/2']` (stores *nan* if a function wasn’t called)

`cosmoslik.cosmoslik.`**run_chain**(*main, nchains=1, pool=None, args=None, kwargs=None*)
Run a CosmoSlik chain, or if *nchains*!=1 run a set of chains in parallel.

Non-trivial parallelization, e.g. using MPI or with communication amongst chains, is handled by each cosmoslik.Sampler module. See individual documentation.

Parameters

- **main** – the class object for your *SlikPlugin* (i.e. `myplugin`, not `myplugin()`)
- **pool** – any worker pool which has a `pool.map` function. Defaults to `multiprocessing.Pool(nchains)`
- **nchains** (*int*) – the number of chains to run in parallel using `pool.map`
- ***args** – args to pass to *main*
- ****kwargs** – kwargs to pass to *main*

Returns A `chain.Chain` instance if nchains=1, otherwise a `chain.Chains` instance `cosmoslik.cosmoslik.`**`SlikMain`**(*cls*)

Class decorator which marks a plugin as being the main one when running a script from the command line.

Example:

```
# somefile.py

@SlikMain  #run this plugin when I run `$ cosmoslik somefile.py`
class plugin1(SlikPlugin):
    ...

class plugin2(SlikPlugin):
    ...
```

`cosmoslik.cosmoslik.`**`arguments`**(*exclude=None*, *exclude_self=True*, *include_kwargs=True*, *ifset=False*)

Return a dictionary of the current function's arguments (excluding self, and optionally other user specified ones or default ones)

Parameters

- **`exclude`** (*`list`*) – argument names to exclude
- **`exclude_self`** (*`bool`*) – exclude the "self" argument
- **`include_kwargs`** (*`bool`*) – include the args captured by kwargs
- **`ifset`** (*`bool`*) – exclude keyword arguments whose value is their default

Example:

```
def foo(x, y=2, **kwargs):
    return arguments()

f(1)        # returns {'x':1, 'y':2}
f(1,3)      # returns {'x':1, 'y':3}
f(1,z=5)    # returns {'x':1, 'y':3, 'z':5}
```

Adapted from: http://kbyanc.blogspot.fr/2007/07/python-aggregating-function-arguments.html

`cosmoslik.cosmoslik.`**`get_caller`**(*depth=1*)

Get the calling function. Works in many typical (but not all) cases.

Thanks to: http://stackoverflow.com/a/39079070/1078529

Example:

```
def foo():
    return get_caller()

foo() #returns the 'foo' function object
```

or

def foo(): return bar()

def bar(): return get_caller(depth=2)

foo() #returns the 'foo' function object

class `cosmoslik.cosmoslik.`**`sample`**(*x*, *lnl*, *weight=1*, *extra=None*)

Bases: `object`

A SlikSampler's sample method should yield objects of this type.

__init__(*x*, *lnl*, *weight=1*, *extra=None*)
Initialize self. See help(type(self)) for accurate signature.

__weakref__
list of weak references to the object (if defined)

1.6.1.3 Module contents

1.6.2 cosmoslik_plugins package

1.6.2.1 Subpackages

1.6.2.1.1 cosmoslik_plugins.likelihoods package

Subpackages

cosmoslik_plugins.likelihoods.SPTSZ_lowl_2017 package

Submodules

class cosmoslik_plugins.likelihoods.SPTSZ_lowl_2017.SPTSZ_lowl.**SPTSZ_lowl**(*lmin=None*, *lmax=None*, *ab_on=False*, *cal=1*, ***kwargs*)

Bases: *cosmoslik.cosmoslik.SlikPlugin*

__call__(*cmb*)
Calculation code here.

__init__(*lmin=None*, *lmax=None*, *ab_on=False*, *cal=1*, ***kwargs*)
Initialization code here.

class cosmoslik_plugins.likelihoods.SPTSZ_lowl_2017.SPTSZ_lowl.**SPTSZ_lowl_egfs**(*Asz=<cosmoslik. ob- ject>*, *Acl=<cosmoslik. ob- ject>*, *Aps=<cosmoslik ob- ject>*, ***kwargs*)

Bases: *cosmoslik.cosmoslik.SlikPlugin*

The default foreground model for the SPT low-L likelihood from Story/Keisler et al. which features (tSZ+kSZ), CIB clustering, and CIB Poisson templates, with free amplitudes and priors on these amplitudes.

__call__(*lmax*, ***_*)
Calculation code here.

__init__(*Asz=<cosmoslik.cosmoslik.param object>*, *Acl=<cosmoslik.cosmoslik.param object>*, *Aps=<cosmoslik.cosmoslik.param object>*, ***kwargs*)
Initialization code here.

Module contents

cosmoslik_plugins.likelihoods.planck package

Submodules

class `cosmoslik_plugins.likelihoods.planck.clik.`**`clik`**(*clik_file*, *auto_reject_errors=False*, *lensing=None*, ***nuisance*)

Bases: `cosmoslik.cosmoslik.SlikPlugin`

`__call__`(*cmb*)
Calculation code here.

`__init__`(*clik_file*, *auto_reject_errors=False*, *lensing=None*, ***nuisance*)
Initialization code here.

class cosmoslik_plugins.likelihoods.planck.planck.**planck_2015_highl_TT**(*clik_file, A_cib_217=<cosmoslik.cosm ob- ject>, A_planck=<cosmoslik.cosmo ob- ject>, A_sz=<cosmoslik.cosmoslik. ob- ject>, calib_100T=<cosmoslik.cosn ob- ject>, calib_217T=<cosmoslik.cosn ob- ject>, cib_index=- 1.3, gal545_A_100=<cosmoslik.c ob- ject>, gal545_A_143=<cosmoslik.c ob- ject>, gal545_A_143_217=<cosmo ob- ject>, gal545_A_217=<cosmoslik.c ob- ject>, ksz_norm=<cosmoslik.cosmo ob- ject>, ps_A_100_100=<cosmoslik.c ob- ject>, ps_A_143_143=<cosmoslik.c ob- ject>, ps_A_143_217=<cosmoslik.c ob- ject>, ps_A_217_217=<cosmoslik.c ob- ject>, xi_sz_cib=<cosmoslik.cosmo ob- ject>, **kwargs*)

Bases: `cosmoslik_plugins.likelihoods.planck.clik.clik`

`__init__`(*clik_file, A_cib_217=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>, A_planck=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>, A_sz=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>, calib_100T=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>, calib_217T=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>, cib_index=-1.3, gal545_A_100=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>, gal545_A_143=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>, gal545_A_143_217=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>, gal545_A_217=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>, ksz_norm=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>, ps_A_100_100=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>, ps_A_143_143=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>, ps_A_143_217=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>, ps_A_217_217=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>, xi_sz_cib=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>, **kwargs*)

Initialization code here.

class cosmoslik_plugins.likelihoods.planck.planck.**planck_2015_highl_TTTEEE**(*A_pol=1*, *calib_100P=1*, *calib_143P=1*, *calib_217P=1*, *galf_EE_A_100=<cosm ob- ject>*, *galf_EE_A_100_143=< ob- ject>*, *galf_EE_A_100_217=< ob- ject>*, *galf_EE_A_143=<cosm ob- ject>*, *galf_EE_A_143_217=< ob- ject>*, *galf_EE_A_217=<cosm ob- ject>*, *galf_EE_index=- 2.4*, *galf_TE_A_100=<cosm ob- ject>*, *galf_TE_A_100_143=< ob- ject>*, *galf_TE_A_100_217=< ob- ject>*, *galf_TE_A_143=<cosm ob- ject>*, *galf_TE_A_143_217=< ob- ject>*, *galf_TE_A_217=<cosm ob- ject>*, *galf_TE_index=- 2.4*, ***kwargs*)

Bases: *cosmoslik_plugins.likelihoods.planck.planck.planck_2015_highl_TT*

__init__(*A_pol=1*, *calib_100P=1*, *calib_143P=1*, *calib_217P=1*, *galf_EE_A_100=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>*, *galf_EE_A_100_143=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>*, *galf_EE_A_100_217=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>*, *galf_EE_A_143=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>*, *galf_EE_A_143_217=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>*, *galf_EE_A_217=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>*, *galf_EE_index=-2.4*, *galf_TE_A_100=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>*, *galf_TE_A_100_143=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>*, *galf_TE_A_100_217=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>*, *galf_TE_A_143=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>*, *galf_TE_A_143_217=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>*, *galf_TE_A_217=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>*, *galf_TE_index=-2.4*, ***kwargs*)

Initialization code here.

class cosmoslik_plugins.likelihoods.planck.planck.**planck_2015_lowl_TT**(*clik_file*, *A_planck=<cosmoslik.cosmos* *ob-* *ject>*)

Bases: *cosmoslik_plugins.likelihoods.planck.clik.clik*

__init__(*clik_file*, *A_planck=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>*)

Initialization code here.

class cosmoslik_plugins.likelihoods.planck.planck.**planck_2015_lowl_TEB**(*clik_file*, *A_planck=<cosmoslik.cosmo* *ob-* *ject>*)

Bases: *cosmoslik_plugins.likelihoods.planck.clik.clik*

__init__(*clik_file*, *A_planck=<cosmoslik.cosmoslik.param_shortcut.<locals>.param_shortcut object>*)

Initialization code here.

Module contents

Submodules

class cosmoslik_plugins.likelihoods.priors.**priors**(*params*)

Bases: *cosmoslik.cosmoslik.SlikPlugin*

__call__(*params*)

Calculation code here.

__init__(*params*)

Initialization code here.

Module contents

1.6.2.1.2 cosmoslik_plugins.misc package

Subpackages

cosmoslik_plugins.misc.cyquad package

Submodules

Module contents

Module contents

1.6.2.1.3 cosmoslik_plugins.models package

Subpackages

cosmoslik_plugins.models.cosmo_derived package

Submodules

Module contents

cosmoslik_plugins.models.hubble_theta package

Module contents

class `cosmoslik_plugins.models.hubble_theta.`**`hubble_theta`**

Bases: *`cosmoslik.cosmoslik.SlikPlugin`*

`__init__``()`

Initialization code here.

Submodules

class `cosmoslik_plugins.models.bbn_consistency.`**`bbn_consistency`**

Bases: *`cosmoslik.cosmoslik.SlikPlugin`*

`__call__``(`*ombh2*, *Neff=3.046*, ***kwargs*`)`

Calculation code here.

`__init__``()`

Initialization code here.

class `cosmoslik_plugins.models.camb.`**`camb`**`(`***defaults*`)`

Bases: *`cosmoslik.cosmoslik.SlikPlugin`*

Compute the CMB power spectrum with CAMB.

`__call__`(*ALens=None*, *As=None*, *DoLensing=None*, *H0=None*, *k_eta_max_scalar=None*, *lmax=None*, *massive_neutrinos=None*, *massless_neutrinos=None*, *mnu=None*, *Neff=None*, *NonLinear=None*, *ns=None*, *ombh2=None*, *omch2=None*, *omk=None*, *pivot_scalar=None*, *tau=None*, *theta=None*, *Yp=None*, *nowarn=False*, ***kwargs*)

Parameters

- **`nowarn`** (*`bool`*) – don't warn about unrecognized parameters which were passed in
- **`Returns`** (*`dict`*) – dictionary of {'TT':array(), 'TE':array(), … } giving the CMB Dl's in muK^2

`__init__`(***defaults*)

defaults [dict] any of the parameters accepted by __call__. these will be their defaults unless explicitly passed to __call__.

`convert_params`(***params*)
Convert from CosmoSlik params to pycamb

`class` `cosmoslik_plugins.models.classy.`**`classy`**(***defaults*)
Bases: *`cosmoslik.cosmoslik.SlikPlugin`*

Compute the CMB power spectrum with CLASS.

Based on work by: Brent Follin, Teresa Hamill

`__call__`(*As=None*, *DoLensing=True*, *H0=None*, *lmax=None*, *mnu=None*, *Neff=None*, *nrun=None*, *ns=None*, *ombh2=None*, *omch2=None*, *omk=None*, *output='tCl*, *lCl*, *pCl'*, *pivot_scalar=None*, *r=None*, *tau=None*, *Tcmb=2.7255*, *theta=None*, *w=None*, *Yp=None*, *nowarn=False*, ***kwargs*)
Calculation code here.

`__init__`(***defaults*)
Initialization code here.

`convert_params`(***params*)
Convert from CosmoSlik params to CLASS

`class` `cosmoslik_plugins.models.clust_poisson_egfs.`**`clust_poisson_egfs`**(**args*, ***kwargs*)
Bases: *`cosmoslik_plugins.models.egfs.egfs`*

`class` `cosmoslik_plugins.models.cosmology.`**`cosmology`**(*model="*, ***kwargs*)
Bases: *`cosmoslik.cosmoslik.SlikPlugin`*

`__init__`(*model="*, ***kwargs*)
Initialization code here.

`class` `cosmoslik_plugins.models.egfs.`**`egfs`**(**args*, ***kwargs*)
Bases: *`cosmoslik.cosmoslik.SlikPlugin`*

An

To create your own extra-galactic foreground model, create a subclass of `cosmoslik.plugins.models.egfs.egfs` and override the function `get_egfs` to return a dictionary of extra-galactic foreground components.

Also passed to the *get_egfs* function is information from the dataset, such as

- *spectra* : e.g. *cl_TT* or *cl_EE*
- *freq* : a dictionary for different effective frequencies, e.g. *{'dust': 153, 'radio': 151, 'tsz':150}*
- *fluxcut* : the fluxcut in mJy

- *lmax* : the necessary maximum l

Here's an example egfs model:

from cosmoslik.plugins.models.egfs import egfs

class MyEgfs(egfs):

def get_egfs(self, p, spectra, freq, fluxcut, lmax, **kwargs): return {'single_component': p['amp'] * ones(lmax)}

`__call__`(***kwargs*)
Calculation code here.

class `cosmoslik_plugins.models.egfs.`**`egfs_specs`**(**args*, ***kwargs*)
Bases: `cosmoslik.cosmoslik.SlikDict`

This class stores information needed to calculate the extra-galactic foreground contribution to some particular power spectrum. This information is,

kind : 'TT', 'TE', ... freqs : tuple of dicts

the pair of frequencies being correlated. each entry isnt a number, rathers its a dict with keys 'dust', 'radio', and 'tsz', specfying the band center for each type of component in GHz

fluxcut [the fluxcut] the fluxcut in mJy

class `cosmoslik_plugins.models.pico.`**`pico`**(*datafile*)
Bases: `cosmoslik.cosmoslik.SlikPlugin`

`__call__`(*outputs=[]*, *force=False*, *onfail=None*, ***kwargs*)
Calculation code here.

`__init__`(*datafile*)
Initialization code here.

Module contents

1.6.2.1.4 cosmoslik_plugins.samplers package

Submodules

class `cosmoslik_plugins.samplers.emcee.`**`emcee`**(*params*, *num_samples*, *nwalkers=100*, *cov_est=None*, *output_extra_params=None*, *output_file=None*, *output_freq=10*)
Bases: `cosmoslik.cosmoslik.SlikSampler`

`__init__`(*params*, *num_samples*, *nwalkers=100*, *cov_est=None*, *output_extra_params=None*, *output_file=None*, *output_freq=10*)
Initialize self. See help(type(self)) for accurate signature.

`cosmoslik_plugins.samplers.emcee.`**`multivariate_normal`**(*mean*, *cov*[, *size*, *check_valid*, *tol*])
Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

Parameters

- **mean** (*1-D array_like, of length N*) – Mean of the N-dimensional distribution.
- **cov** (*2-D array_like, of shape (N, N)*) – Covariance matrix of the distribution. It must be symmetric and positive-semidefinite for proper sampling.
- **size** (*int or tuple of ints, optional*) – Given a shape of, for example, `(m, n, k)`, `m*n*k` samples are generated, and packed in an *m*-by-*n*-by-*k* arrangement. Because each sample is *N*-dimensional, the output shape is `(m, n, k, N)`. If no shape is specified, a single (*N*-D) sample is returned.
- **check_valid** (*{ 'warn', 'raise', 'ignore' }, optional*) – Behavior when the covariance matrix is not positive semidefinite.
- **tol** (*float, optional*) – Tolerance when checking the singular values in covariance matrix.

Returns

out – The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

Return type ndarray

1.6.2.1.5 Notes

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ...x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j. The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- Spherical covariance (*cov* is a multiple of the identity matrix)
- Diagonal covariance (*cov* has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0, 0]
>>> cov = [[1, 0], [0, 100]]  # diagonal covariance
```

Diagonal covariance means that points are oriented along x or y-axis:

```
>>> import matplotlib.pyplot as plt
>>> x, y = np.random.multivariate_normal(mean, cov, 5000).T
>>> plt.plot(x, y, 'x')
>>> plt.axis('equal')
>>> plt.show()
```

Note that the covariance matrix must be positive semidefinite (a.k.a. nonnegative-definite). Otherwise, the behavior of this method is undefined and backwards compatibility is not guaranteed.

1.6.2.1.6 References

1.6.2.1.7 Examples

```
>>> mean = (1, 2)
>>> cov = [[1, 0], [0, 1]]
>>> x = np.random.multivariate_normal(mean, cov, (3, 3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> list((x[0,0,:] - mean) < 0.6)
[True, True]
```

class `cosmoslik_plugins.samplers.metropolis_hastings.`**`metropolis_hastings`**(*params*, *output_file=None*, *output_extra_params=None*, *num_samples=100*, *print_level=0*, *cov_est=None*, *proposal_scale=2.4*, *proposal_update=True*, *proposal_update_start=1000*, *mpi_comm_freq=100*, *max_weight=10*, *debug_output=False*, *temp=1*, *reseed=True*, *yield_rejected=False*)

Bases: *`cosmoslik.cosmoslik.SlikSampler`*

An adaptive metropolis hastings sampler.

To run with MPI, call your script with:

cosmoslik -n <nchains+1> script.py

(Note one process is a "master" so run one more process than you want chains)

`__init__`(*params*, *output_file=None*, *output_extra_params=None*, *num_samples=100*, *print_level=0*, *cov_est=None*, *proposal_scale=2.4*, *proposal_update=True*, *proposal_update_start=1000*, *mpi_comm_freq=100*, *max_weight=10*, *debug_output=False*, *temp=1*, *reseed=True*, *yield_rejected=False*)

Parameters

- **`params`** – The script to which this sampler is attached
- **`output_file`** – File where to save the chain (if running with MPI, everything still

gets dumped into one file). By default only sampled parameters get saved. Use *cosmoslik.utils.load_chain* to load chains.

- **`output_extra_params`** – Extra parameters besides the sampled ones which to save to file. Arbitrary objects can be outputted, in which case entires should be tuples of (<name>,'object'), or for more efficient and faster file write/reads (<name>,<dtype>) where <dtype> is a valid numpy dtype (e.g. '(10,10)d' for a 10x10 array of doubles, etc…)
- **`num_samples`** – The number of total desired samples (including rejected ones)
- **`print_level`** – 0/1/2 to print little/medium/alot
- **`cov_est`** – One or a list of covariances which will be combined with K.chains.combine_cov (see documentation there for understood formats) to produce the full proposal covariance. Covariance for any sampled parameter not provided here will be taken from the *scale* attribute of that parameters. This should be a best estimate of the posterior covariance. The actual proposal covariance is multiplied by *proposal_scale**2 / N* where *N* is the number of parameters. (default: diagonal covariance taken from the *scale* of each parameter)
- **`proposal_scale`** – Scale the proposal matrix. (default: 2.4)
- **`proposal_update`** – Whether to update the proposal matrix. Ignored if not running with MPI. The proposal is updated by taking the sample covariance of the last half of each chain. (default: True)
- **`proposal_update_start`** – If *proposal_update* is True, how many total samples (including rejected) per chain to wait before starting to do updates (default: 1000).
- **`mpi_comm_freq`** – Number of accepted samples to wait inbetween the chains communicating with the master process and having their progress written to file (default: 50)
- **`max_weight`** – If a the chain stays in the same location more than this number of samples, it is broken up as distinct steps
- **`reseed`** – Draw a random seed based on system time and process number before starting. (default: True)
- **`yield_rejected`** – Yield samples with 0 weight (default: False)
- **`debug_output`** – Print (code) debugging messages.

`sample`(*lnl*)

Returns a generator which yields samples from the likelihood using the Metropolis-Hastings algorithm.

The samples returned are tuples of (x,weight,lnl,extra) lnl - likelihood weight - the statistical weight (could be 0 for rejected steps) x - the vector of parameter values extra - the extra information returned by lnl

`cosmoslik_plugins.samplers.metropolis_hastings.`**`multivariate_normal`**(*mean*, *cov*[, *size*, *check_valid*, *tol*])

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

Parameters

- **mean** (*1-D array_like, of length N*) – Mean of the N-dimensional distribution.
- **cov** (*2-D array_like, of shape (N, N)*) – Covariance matrix of the distribution. It must be symmetric and positive-semidefinite for proper sampling.
- **size** (*int or tuple of ints, optional*) – Given a shape of, for example, `(m,n,k)`, `m*n*k` samples are generated, and packed in an *m*-by-*n*-by-*k* arrangement. Because each sample is *N*-dimensional, the output shape is `(m,n,k,N)`. If no shape is specified, a single (*N*-D) sample is returned.
- **check_valid** (*{ 'warn', 'raise', 'ignore' }, optional*) – Behavior when the covariance matrix is not positive semidefinite.
- **tol** (*float, optional*) – Tolerance when checking the singular values in covariance matrix.

Returns

out – The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

Return type ndarray

1.6.2.1.8 Notes

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ...x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j. The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- Spherical covariance (*cov* is a multiple of the identity matrix)
- Diagonal covariance (*cov* has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0, 0]
>>> cov = [[1, 0], [0, 100]]  # diagonal covariance
```

Diagonal covariance means that points are oriented along x or y-axis:

```
>>> import matplotlib.pyplot as plt
>>> x, y = np.random.multivariate_normal(mean, cov, 5000).T
>>> plt.plot(x, y, 'x')
>>> plt.axis('equal')
>>> plt.show()
```

Note that the covariance matrix must be positive semidefinite (a.k.a. nonnegative-definite). Otherwise, the behavior of this method is undefined and backwards compatibility is not guaranteed.

1.6.2.1.9 References

1.6.2.1.10 Examples

```
>>> mean = (1, 2)
>>> cov = [[1, 0], [0, 1]]
>>> x = np.random.multivariate_normal(mean, cov, (3, 3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> list((x[0,0,:] - mean) < 0.6)
[True, True]
```

`cosmoslik_plugins.samplers.metropolis_hastings.`**`seed`**`(`*seed=None*`)`

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

Parameters **`seed`** (*`int or 1-d array_like, optional`*) – Seed for *RandomState*. Must be convertible to 32 bit unsigned integers.

See also:

`RandomState()`

`cosmoslik_plugins.samplers.metropolis_hastings.`**`uniform`**`(`*low=0.0*, *high=1.0*, *size=None*`)`

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

Parameters

- **`low`** (*`float or array_like of floats, optional`*) – Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.
- **`high`** (*`float or array_like of floats`*) – Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.
- **`size`** (*`int or tuple of ints, optional`*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `low` and `high` are both scalars. Otherwise, `np.broadcast(low, high).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized uniform distribution.

Return type ndarray or scalar

See also:

`randint()` Discrete uniform distribution, yielding integers.

`random_integers()` Discrete uniform distribution over the closed interval `[low, high]`.

`random_sample()` Floats uniformly distributed over `[0, 1)`.

`random()` Alias for *random_sample*.

rand() Convenience function that accepts dimensions as input, e.g., `rand(2,2)` would generate a 2-by-2 array of floats, uniformly distributed over `[0, 1)`.

1.6.2.1.11 Notes

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b-a}$$

anywhere within the interval `[a, b)`, and zero elsewhere.

When `high` == `low`, values of `low` will be returned. If `high` < `low`, the results are officially undefined and may eventually raise an error, i.e. do not rely on this function to behave when passed arguments satisfying that inequality condition.

1.6.2.1.12 Examples

Draw samples from the distribution:

```
>>> s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

class `cosmoslik_plugins.samplers.priors.`**`priors`**(*params*, *output_extra_params=None*, ***kwargs*)

Bases: *`cosmoslik_plugins.samplers.metropolis_hastings.metropolis_hastings`*

Sample from the prior.

`__init__`(*params*, *output_extra_params=None*, ***kwargs*)

Args: params:

The script to which this sampler is attached

output_file: File where to save the chain (if running with MPI, everything still gets dumped into one file). By default only sampled parameters get saved. Use *cosmoslik.utils.load_chain* to load chains.

output_extra_params: Extra parameters besides the sampled ones which to save to file. Arbitrary objects can be outputted, in which case entires should be tuples of (<name>,'object'), or for more efficient and faster file write/reads (<name>,<dtype>) where <dtype> is a valid numpy dtype (e.g. '(10,10)d' for a 10x10 array of doubles, etc…)

num_samples: The number of total desired samples (including rejected ones)

print_level: 0/1/2 to print little/medium/alot

cov_est: One or a list of covariances which will be combined with K.chains.combine_cov (see documentation there for understood formats) to produce the full proposal covariance. Covariance for any sampled parameter not provided here will be taken from the *scale* attribute of that parameters. This should be a best estimate of the posterior covariance. The actual proposal covariance is multiplied by *proposal_scale**2 / N* where *N* is the number of parameters. (default: diagonal covariance taken from the *scale* of each parameter)

proposal_scale: Scale the proposal matrix. (default: 2.4)

proposal_update: Whether to update the proposal matrix. Ignored if not running with MPI. The proposal is updated by taking the sample covariance of the last half of each chain. (default: True)

proposal_update_start: If *proposal_update* is True, how many total samples (including rejected) per chain to wait before starting to do updates (default: 1000).

mpi_comm_freq: Number of accepted samples to wait inbetween the chains communicating with the master process and having their progress written to file (default: 50)

max_weight: If a the chain stays in the same location more than this number of samples, it is broken up as distinct steps

reseed: Draw a random seed based on system time and process number before starting. (default: True)

yield_rejected: Yield samples with 0 weight (default: False)

debug_output: Print (code) debugging messages.

`cosmoslik_plugins.samplers.priors.`**`uniform`**(*low=0.0*, *high=1.0*, *size=None*)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

Parameters

- **`low`** (*`float or array_like of floats, optional`*) – Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.
- **`high`** (*`float or array_like of floats`*) – Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.
- **`size`** (*`int or tuple of ints, optional`*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `low` and `high` are both scalars. Otherwise, `np.broadcast(low, high).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized uniform distribution.

Return type ndarray or scalar

See also:

`randint()` Discrete uniform distribution, yielding integers.

`random_integers()` Discrete uniform distribution over the closed interval `[low, high]`.

`random_sample()` Floats uniformly distributed over `[0, 1)`.

`random()` Alias for *random_sample*.

`rand()` Convenience function that accepts dimensions as input, e.g., `rand(2,2)` would generate a 2-by-2 array of floats, uniformly distributed over `[0, 1)`.

1.6.2.1.13 Notes

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b - a}$$

anywhere within the interval `[a, b)`, and zero elsewhere.

When `high` == `low`, values of `low` will be returned. If `high` < `low`, the results are officially undefined and may eventually raise an error, i.e. do not rely on this function to behave when passed arguments satisfying that inequality condition.

1.6.2.1.14 Examples

Draw samples from the distribution:

```
>>> s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

`cosmoslik_plugins.samplers.utils.`**`initialize_covariance`**`(`*sampled*, *covs=None*`)`

Get the covariance for the set of parameters in *sampled*

Prepare the proposal covariance based on anything passed to self.proposal_cov, defaulting to the *scale* of each sampled parameter otherwise.

Module contents

1.6.2.2 Module contents

Python Module Index

Index

Symbols