Monte Python Documentation

Release 2.2.0

Benjamin Audren

October 21, 2015

Contents

The main page lives here, from which you can download the code, see the changelog. The Github page is available there.

All useful information concerning the installation, some tips on how to organize the folder, and the complete description of the code source is found below.

For the list of command line arguments, please see the documentation of the *create parser function*. You can also ask this same information interactively by asking:

```
python montepython/MontePython.py -h / --help
python montepython/MontePython.py run -h
python montepython/MontePython.py info -h
```

The first one gives you all the possible modes for running (`run`, or `info`), while the other two give you the information specific for each modes. Note that asking for `-h` or `--help` will result in using a short or long format for the help.

Contents:

CHAPTER 1

Installation Guide

1.1 Prerequisites

1.1.1 Python

First of all, you need a clean installation of Python (version 2.7 is better, though it works also with 2.6 and 2.5 (see below). version 3.0 is not supported), with at least the numpy module (version $\geq 1.4.1$) and the cython module. This last one is to convert the C code CLASS into a Python class.

If you also want the output plot to have cubic interpolation for analyzing chains, you should also have the scipy module (at least version 0.9.0). In case this one is badly installed, you will have an error message when running the analyze module of *Monte Python*, and obtain only linear interpolation. Though not fatal, this problem produces ugly plots.

To test for the presence of the modules **numpy**, **scipy**, **cython** on your machine, you can type

```
$ python
$ >>> import numpy
$ >>> import scipy
$ >>> import cython
$ >>> exit()
```

If one of these steps fails, go to the corresponding websites, and follow the instructions (if you have the privilege to have the root password on your machine, an *apt-get install python-numpy*, *python-scipy* and *cython* will do the trick. Otherwise, all these packages can also be downloaded and installed locally, with the command `python setup.py install --user`).

Note that you can use the code with Python 2.6 also, even though you need to download two packages separately ordereddict and argparse. For this, it is just a matter of downloading the two files *ordereddict.py* and *argparse.py*), and placing them in your code directory without installation steps.

1.1.2 Class

Next in line, you must compile the python wrapper of CLASS. Download the latest version ($\geq 1.5.0$), and follow the basic instruction. Instead of `make class`, type `make -j`. If you are using a *Class* version $\geq 2.3.0$, the wrapper will be installed by doing this step, so skip ahead.

In case you are using an older version of *Class*, the compilation only created an archiv *.ar* of the code, useful in the next step. After this, do:

```
class]$ cd python/
python]$ python setup.py build
python]$ python setup.py install --user
```

If you have correctly installed cython, this should add Classy as a new python module. You can check the success of this operation by running the following command:

```
~]$ python
>>> from classy import Class
```

If the installation was successful, this should work within any directory. If you get no error message from this line, you know everything is fine.

Note: If the step `python setup.py install --user` does not succeed, but that the `build` is successful, then as far as *Monte Python* is concerned, there are no issues. The code will be found nonetheless.

If at some point you have several different coexisting versions of *Class* on the system, and you are worried that *Monte Python* is not using the good one, rest reassured. As long as you run *Monte Python* with the proper path to the proper *Class* in your configuration file (see *Installation*) then it will use this one.

1.2 Installation

1.2.1 Main code

Move the latest release of *Monte Python* to one of your folders, called e.g. `code/` (for instance, this could be the folder containing also `class/`), and untar its content:

```
code]$ bunzip montepython-v1.0.0.tar.bz2
code]$ tar -xvf montepython-v1.0.0.tar
code]$ cd montepython
```

You will have to create one file holding the path of the codes you want to use. There is a predefined template, `default.conf.template`, in the root directory of the code. You should copy it to a new file called `default.conf`, which will tell *Monte Python*, where your other programs (in particular *Class*) are installed, and where you are storing the data for the likelihoods. It will be interpreted as a python file, so be careful to reproduce the syntax exactly. At minimum, **default.conf** should contain one line, filled with the path of your `class/` directory:

```
path['cosmo']    = 'path/to/your/class/'
```

To check that *Monte Python* is ready to work, simply type `python montepython/MontePython.py --help` (or just `montepython/MontePython.py --help`). This will provide you with a short description of the available command line arguments, explained in Parser module.

1.2.2 Planck likelihood

With the release of Planck data comes the release of its likelihood. It is distributed from this ESA website, along with the data. Download all *tar.gz* files, extract them to the place of your convenience.

The Planck Likelihood Code (**plc**) is based on a library called *clik*. It will be extracted, alongside several *.clik* folders that contain the likelihoods. The installation of the code is described in the archive, and it uses an auto installer device, called *waf*.

Warning: Note that you **are strongly advised** to configure *clik* with the Intel mkl library, and not with lapack. There is a massive gain in execution time: without it, the code is dominated by the execution of the low-l polarisation data from WMAP.

In your *Monte Python* configuration file, to use this code, you should add the following line

```
path['clik'] = 'path/to/your/plc/folder/'
```

The four likelihoods defined in *Monte Python* for Planck are *Planck_highl*, *Planck_lowl*, *Planck_lensing*, *lowlike* (the polarization data from WMAP). In each of the respective data files for these likelihood, please make sure that the line, for instance,

```
Planck_highl.path_clik = data.path['clik']+'../something.clik'
```

points to the correct clik file. Now, before trying to run this likelihood, you will need to source the code to your system, by typing:

```
~]$ source /path/to/your/plc/folder/bin/clik_profile.sh
```

Once you made sure of this, you can then use the base.param file distributed with MontePython, that defines all the needed nuisance parameters, the covariance matrix as well as the bestfit file, in this command:

```
python montepython/MontePython.py -o planck/ -p base.param -c covmat/base.covmat \
-bf bestfit/base.bestfit --conf default.conf -f 1.5
```

Note: The use of the factor 1.5 is to increase the acceptance rate, due to the non gaussianity of the nuisance parameters posterior.

1.2.3 WMAP likelihood

Warning: As of version 1.2.5, with Planck data being available, installing this likelihood might not be so important anymore. You might prefer to skip this, at it is an **optional** part of the installation process.

Warning: So far, the use of the WMAP wrapper is separated from the Planck wrapper, but it might be merged in the future, as it is based on the same code *clik* developped internally for Planck by Karim Benabed.

To use the likelihood of WMAP, we propose a python wrapper, located in the `wrapper_wmap` directory. Just like with the *Class* wrapper, you need to install it, although the procedure differs. Go to the wrapper directory, and enter:

```
wrapper_wmap]$ ./waf configure install_all_deps
```

This should read the configuration of your distribution, and install the WMAP likelihood code and its dependencies (cfitsio) automatically on your machine. For our purpose, though, we prefer using the intel mkl libraries, which are much faster. To tell the code about your local installation of mkl libraries, please add to the line above some options:

```
--lapack_mkl=/path/to/intel/mkl/10.3.8 --lapack_mkl_version=10.3
```

Once the configuration is done properly, finalize the installation by typing:

```
wrapper_wmap]$ ./waf install
```

The code will generate a configuration file, that you will need to source before using the WMAP likelihood with *Monte Python*. The file is `clik_profile.sh`, and is located in `wrapper_wmap/bin/`. So if you want to use the likelihood `'wmap'`, before any call to *Monte Python* (or inside your scripts), you should execute

```
~]$ source /path/to/MontePython/wrapper_wmap/bin/clik_profile.sh
```

The wrapper will use the original version of the WMAP likelihood codes downloaded and placed in the folder `wrapper_wmap/src/likelihood_v4p1/` during the installation process. This likelihood will be compiled later, when you will call it for the first time from the *Monte Python* code. Before calling it for the first time, you could eventually download the WMAP patch from Wayne Hu's web site, for a faster likelihood.

You should finally download the WMAP data files by yourself, place them anywhere on your system, and specify the path to these data files in the file `likelihoods/wmap/wmap.data`.

CHAPTER 2

Getting Started

2.1 Foreword

Python has a very nice way of handling errors in the execution. Instead of a segmentation fault as in C, when the code breaks, you have access to the whole stack of actions that lead to the error. This helps you pin-point which function was called, which line was responsible for the error.

It can however be lengthy, and to help everyone reading it, a messaging system was implemented in Monte Python. After a blank line, a summary of the actual error will be displayed. When reporting for an error, please attach the entire output, as this is priceless for debugging.

2.2 Input parameter file

An example of input parameter file is provided with the download package, under the name `example.param`. Input files are organised as follows:

```
data.experiments = ['experiment1', 'experiment2', ...]

data.parameters['cosmo_name']       = [mean, min, max, sigma, scale, 'cosmo']
...

data.parameters['nuisance_name']    = [mean, min, max, sigma, scale, 'nuisance']
...

data.parameters['cosmo_name']       = [mean, min, max, sigma, scale, 'derived']
...

data.cosmo_arguments['cosmo_name']           = value

data.N = 10
data.write_step = 5
```

The first command is rather explicit. You will list there all the experiments you want to take into account. Their name should coincide with the name of one of the several sub-directories in the `montepython/likelihoods/` directory. Likelihoods will be explained in the Likelihood class module

In `data.parameters`, you can list all the cosmo and nuisance parameter that you want to vary in the Markov chains. For each of them you must give an array with six elements, in this order:

- **mean value** (your guess for the best fitting value, from which the first jump will start)
- **minimum value** (set to *-1* or *None* for unbounded prior edge),

- **maximum value** (set to *-1* or *None* for unbounded prior edge),
- **sigma** (your guess for the standard deviation of the posterior of this parameter, its square will be used as the variance of the proposal density when there is no covariance matrix including this parameter passed as an input),
- **scale** (most of the time, it will be 1, but occasionnaly you can use a rescaling factor for convenience, for instance {tt 1.e-9} if you are dealing with A_s or `0.01` if you are dealing with ω_b)
- **role** (`cosmo` for MCMC parameters used by the Boltzmann code, `nuisance` for MCMC parameters used only by the likelihoods, and `derived` for parameters not directly varied by the MCMC algorithm, but to be kept in the chains for memory).

In `data.cosmo_arguments`, you can pass to the Boltzmann code any parameter that you want to fix to a non-default value (cosmological parameter, precision parameter, flag, name of input file needed by the Bolztmann code, etc.). The names and values should be the same as in a *Class* input file, so the values can be numbers or a strings, e.g:

```
data.cosmo_arguments['Y_He']           = 0.25
```

or

```
data.cosmo_arguments['Y_He']       = 'BBN'
data.cosmo_arguments['sBBN file'] = data.path['cosmo']+'/bbn/sBBN.dat'
```

All elements you input with a `cosmo`, `derived` or `cosmo_arguments` role will be interpreted by the cosmological code (only *Class* so far). They are not coded anywhere inside *Monte Python*. *Monte Python* takes parameter names, assigns values, and passes all of these to *Class* as if they were written in a *Class* input file. The advantages of this scheme are obvious. If you need to fix or vary whatever parameter known by *Class*, you don't need to edit *Monte Python*, you only need to write these parameters in the input parameter file. Also, *Class* is able to interpret input parameters from a *Class* input file with a layer of simple logic, allowing to specify different parameter combinations. Parameters passed from the parameter file of *Monte Python* go through the same layer of logic.

If a `cosmo`, `derived` or `cosmo_arguments` parameter is not understood by the Boltzmann code, *Monte Python* will stop and return an explicit error message. A similar error will occur if one of the likelihoods requires a `nuisance` parameter that is not passed in the list.

You may wish occasionally to use in the MCMC runs a new parameter that is not a *Class* parameter, but can be mapped to one or several *Class* parameters (e.g. you may wish to use in your chains $\log(10^{10}A_s)$ instead of A_s). There is a function, in the module *data*, that you can edit to define such mappings: it is called `update_cosmo_arguments`. Before calling CLASS, this function will simply substitute in the list of arguments your customized parameters by some *Class* parameters. Several exemple of such mappings are already implemented, allowing you for instance to use `'Omega_Lambda'`,`'ln10^{10}A_s'` or `'exp_m_2_tau_As'` in your chains. Looking at these examples, the user can easily write new ones even without knowing python.

The last two lines of the input parameter file are the number of steps you want your chain to contain (`data.N`) and the number of accepted steps the system should wait before writing it down to a file (`data.write_step`). Typically, you will need a rather low number here, e.g. `data.write_step = 5` or `10`. The reason for not setting this parameter to one is just to save a bit of time in writing on the disk.

In general, you will want to specify the number of steps in the command line, with the option `-N` (see section~ref{commands}). This will overwrite the value passed in the input parameter file. The value by default in the parameter file, `data.N = 10`, is intentionnaly low, simply to prevent doing any mistake while testing the program on a cluster.

2.3 Output directory

You are assumed to use the code in the following way: for every set of experiments and parameters you want to test, including different priors, some parameters fixed, etcldots you should use one output folder. This way, the folder will

keep track of the exact calling of the code, allowing you to reproduce the data at later times, or to complete the existing chains. All important data are stored in your `folder/log.param` file.

Incidentaly, if you are starting the program in an existing folder, already containing a `log.param` file, then you do not even have to specify a parameter file: the code will use it automatically. This will avoid mixing things up. If you are using one anyway, the code will warn you that it did not read it: it will always only use the `log.param` file.

In the folder `montepython`, you can create a folder `chains` where you will organize your runs e.g. in the following way:

```
montepython/chains/set_of_experiments1/model1
montepython/chains/set_of_experiments1/model2
...
montepython/chains/set_of_experiments2/model1
montepython/chains/set_of_experiments2/model2
...
```

The minimum amount of command lines for running *Monte Python* is an input file, an output directory and a configuration file: if you have already edited `defaut.conf` or copied it to your own `my-machine.conf`, you may already try a mini-run with the command

```
montepython]$ montepython/MontePython.py -conf my-machine.conf -p example.param -o test
```

2.4 Analyzing chains and plotting

Once you have accumulated a few chains, you can analyse the run to get convergence estimates, best-fit values, minimum credible intervals, a covariance matrix and some plots of the marginalised posterior probability. You can run again *Monte Python* with the `info` prefix followed by the name of a directory or of several chains, e.g. `info chains/myrun/` or `info chains/myrun/2012-10-26* chains/myrun/2012-10-27*`. There is no need to pass an input file with parameter names since they have all been stored in the `log.param`.

Information on the acceptance rate and minimum $-\log\mathcal{L} = \chi^2_{\rm eff}/2$ is written in `chains/myrun/myrun.log`. Information on the convergence (Gelman-Rubin test for each chain parameter), on the best fit, mean and minimum credible interval for each parameter at the 68.26%, 95.4%, 99.7% level are written in horizontal presentation in `chains/myrun/myrun.h_info`, and in vertical presentation in `chains/myrun/myrun.v_info` (without 99.7% in the vertical one). A latex file to produce a table with parameter names, means and 68% errors in written in `chains/myrun/myrun.tex`.

The covariance matrix of the run is written in `chains/myrun/myrun.covmat`. It can be used as an input for the proposal density in a future run. The first line, containing the parameter name, will be read when the covariance matrix will be passed in input. This means that the list of parameters in the input covariance matrix and in the run don't need to coincide: the code will automatically eliminate, add and reorder parameters (see `mcmc.get_covariance_matrix()`). Note that the rescaling factors passed in the input file are used internally during the run and also in the presentation of results in the `.h_info`, `.v_info`, `.tex` files, but not in the covariance matrix file, which refers to the true parameters.

The 1D posteriors and 2D posterior contours are plotted in `chains/myrun/plots/myrun_1D.pdf` and `chains/myrun/plots/myrun_triangle.pdf`. You will find in the Parser module documentation a list of commands to customize the plots.

When the chains are not very converged and the posterior probability has local maxima, the code will fail to compute minimum credible intervals and say it in a warning. The two solutions are either to re-run and increase the number of samples, or maybe just to decrease the number of bins with the `--bins` option.

2.5 Global running strategy

In the current version of *Monte Python*, we deliberately choose not to use MPI communication between instances of the code. Indeed the use of MPI usually makes the installation step more complicated, and the gain is, in our opinion, not worth it. Several chains are launched as individual serial runs (if each instance of *Monte Python* is launched on several cores, *Class* and the WMAP likelihood will parallelize since they use OpenMP). They can be run with the same command since chain names are created automatically with different numbers for each chain: the chain names are in the form `yyyy-mm-dd_N__i.txt` where `yyyy` is the year, `mm` the month, `dd` the day, `N` the requested number of steps and `i` the smallest available integer at the time of starting a new run.

However the absence of communication between chains implies that the proposal density cannot be updated automatically during the initial stage of a run. Hence the usual strategy consists in launching a first run with a poor (or no) covariance matrix, and a low acceptance rate; then to analyze this run and produce a better covariance matrix; and then to launch a new run with high acceptance rate, leading to nice plots. Remember that in order to respect strictly markovianity and the Metropolis Hastings algorithm, one should not mix up chains produced with different covariance matrices: this is easy if one takes advantage of the `info` syntax, for example `info chains/myrun/2012-10-26_10000*`. However mixing runs that started from very similar covariance matrices is harmless.

It is also possible to run on several desktops instead of a single cluster. Each desktop should have a copy of the output folder and with the same `log.param` file, and after running the chains can be grouped on a single machine and analyse. In this case, take care of avoiding that chains are produced with the same name (easy to ensure with either the `-N` or `--chain-number` options). This is a good occasion to keep the desktops of your department finally busy.

CHAPTER 3

Example of a complete work session

I just downloaded and installed *Monte Python*, read the previous pages, and I wish to launch and analyse my first run.

I can first create a few folders in order to keep my `montepython` directory tidy in the future. I do a

`$ mkdir chains` for storing all my chains

`$ mkdir chains/planck` if the first run I want to launch is based on the fake planck likelihood proposed in the `example.param` file

`$ mkdir input` for storing all my input files

`$ mkdir scripts` for storing all my scripts for running the code in batch mode

I then copy `example.param` in my input folder, with a name of my choice, e.g. `lcdm.param`, and edit it if needed:

```
$ cp example.param input/lcdm.param
```

I then launch a short chain with

```
$ montepython/Montepython.py run -p input/lcdm.param -o chains/planck/lcdm -N 5
```

I can see on the screen the evolution of the initialization of the code. At the end I check that I have a chain and a `log.param` written in my `chains/planck/lcdm/log.param` directory. I can immediately repeat the experience with the same command. The second chain is automatically created with number 2 instead of 1. I can also run again without the input file:

```
$ montepython/Montepython.py run -o chains/planck/lcdm -N 5
```

This works equally well because all information is taken from the `log.param` file.

In some cases, initally, I don't have a covariance matrix to pass in input [1] . But in this particular example I can try the one delivered with the *Monte Python* package, in the `covmat/` directory:

```
$ montepython/Montepython.py run -p input/lcdm.param\
       -o chains/planck/lcdm -c covmat/fake_planck_lcdm.covmat -N 5
```

I don't have yet a covariance matrix to pass in input, otherwise I would have run with

```
$ montepython/Montepython.py run -p input/lcdm.param -o chains/planck/lcdm -c mycovmat.covmat -N 5
```

I now wish to launch longer runs on my cluster or on a powerful desktop. The syntax of the script depends on the cluster. In the simplest case it will only contain some general commands concerning the job name, wall time limit etc., and the command line above (I can use the one without input file, provided that I made already one short interactive

[1] If I am also a CosmoMC user, I might have an adequate covmat to start with, before using the covmat that *Monte Python* will produce. Fot this I just need to edit the first line, add comas between paramater names, and for parameter that are identical to those in my run, replace CosmoMC parameter names with equivalent *Class* parameter names.}

run, and that the `log.param` already exists; but I can now increase the number of steps, e.g. to 5000 or 10000). On some cluster, the chain file is created immediately in the output directory at start up. In this case, the automatic numbering of chains proposed by *Monte Python* will be satisfactory.

Warning: On some clusters, the automatic numbering will conflict when the chains are created too fast. Please look at the section on how to use `mpi_run` for guidance

In other clusters, the chains are created on a temporary file, and then copied at the end to the output file. In this case, if I do nothing, there is a risk that chain names are identical and clash. I should then relate the chain name to the job number, with an additional command line `--chain_number $JOBID`. Some clusters, `$JOBID` is a string, but the job number can be extracted with a line like `export JOBNUM="$(echo $PBS_JOBID|cut -d'.' -f1)"`, and passed to *Monte Python* as `--chain_number $JOBNUM`.

If I use in a future run the Planck likelihood, I should not forget to add in the script (before calling *Monte Python*) the line

```
source /path/to/my/plc/bin/clik_profile.sh
```

I then launch a chain by submitting the script, with e.g. `qsub scripts/lcdm.sh`. I can launch many chains in one command with

```
$ for i in {1..10}; do qsub scripts/lcdm.sh;done
```

If you cluster creates the chains too fast, there might be conflicts in the chain names. One way to go around this issue is to run with `mpi`, which is a parallelization process. The chains will be initialised one after the other, each one sending a **go** signal to the next in line.

To launch a job with `mpi`, the syntax is exactly the same than without, except that you will start the whole command with, depending on your installation, `mpirun` or `mpiexec`:

```
mpirun -np 4 python montepython/MontePython.py run -o chains/...
```

will simply launch 4 chains, each using the environment variable `$OMP_NUM_THREADS` for the number of cores to compute *Class*.

When the runs have stopped, I can analyse them with

```
$ montepython/Montepython.py info chains/planck/lcdm
```

If I had been running without a covariance matrix, the results would probably be bad, with a very low acceptance rate and few points. It would have however created a covariance matrix `chains/planck/lcdm/lcdm.covmat`. I can decide to copy it in order to keep track of it even after analysing future runs,

```
cp chains/planck/lcdm/lcdm.covmat chains/planck/lcdm/lcdm_run1.covmat
```

I now add to my script, in the line starting with `montepython/Montepyhton.py`, the option

```
-c chains/planck/lcdm/lcdm_run1.covmat
```

run on the same day as the previous one, it might be smart to change also a bit the number of steps (e.g. from 5000 to 5001) in order to immediately identify chains belonging to the same run.

When this second run is finished, I analyse it with e.g.

```
montepython/Montepython.py info chains/planck/lcdm/2012-10-27_5001*
```

If all R-1 numbers are small (typically < 0.05) and plots look nice, I am done. If not, there can be two reasons: the covariance matrix is still bad, or I just did not get enough samples.

I can check the acceptance rate of this last run by looking at the `chains/planck/lcdm/lcdm.log` file. If I am in a case with nearly gaussian posterior (i.e. nearly ellipsoidal contours), an acceptance rate < 0.2 or > 0.3

can be considered as bad. In other cases, even 0.1 might be the best that I can expect. If the acceptance rate is bad, I must re-run with an improved covariance matrix in order to converge quicker. I copy the last covariance matrix to `lcdm_run2.covmat` and use this one for the next run. If the acceptance rate is good but the chains are not well converged because they are simply too short, then I should better rerun with the same covariance matrix `lcdm_run1.covmat`: in this way, I know that the proposal density is frozen since the second run, and I can safely analyse the second and third runs altogether.

If I do two or three runs in that way, I always loose running time, because each new chain will have a new burn-in phase (i.e. a phase when the log likelihood is very bad and slowly decreasing towards values close to the minimum). If this is a concern, I can avoid it in three ways:

- before launching the new run, I set the input mean value of each parameter in the input file to the best-fit value found in the previous run. The runs will then start from the best-fit value plus or minus the size of the first jump drown from the covariance matrix, and avoid burn-in. Since I have changed the input file, I must rerun with a new output directory, e.g. `chain/lcdm2`. This is a clean method.
- I might prefer a less clean but slightly quicker variant: I modify the mean values, like in the previous item, but directly in the `log.param` file, and I rerun in the same directory without an input file. This will work, but it is advisable not to edit the `log.param` manually, since it is supposed to keep all the information from previous runs.
- I may restart the new chains from the previous chains using the `-r` command line option. The name of previous chains can be written after `-r` manually or through a script.
- I can also restart from the best-fit found previously, using the `-bf` command line option, specifying the `.bestfit` file to use.

When I am pleased with the final plots and result, I can customize the plot content and labels by writing a short file `plot_files/lcdm.plot` passed through the `-extra` command line option, and paste the latex file produced by *Monte Python* in my paper.

CHAPTER 4

Using MultiNest with Monte Python

Monte Python can easily use the implementation of MultiNest by F. Feroz and M. Hobson [1], through the Python wrapper PyMultiNest by J. Buchner [2].

Some hints about why and how to use MultiNest can be found in **'A Basic usage and parameters'_**. A more thorough description of the MultiNest sampler can be found in the MultiNest papers [1]. The PyMultiNest tutorial is also worth checking out, as well as the respective `README` files of both MultiNest and PyMultiNest.

Note: By using MultiNest and PyMultiNest, you agree to their respective licenses, that can be found in the `LICENSE` (or `LICENCE`) files into the respective installation folders.

4.1 Installation

This basically follows the installation procedure in the PyMultiNest documentation.

4.1.1 1. MultiNest

Download MultiNest from here, either using the releases page or cloning with git

```
$ git clone http://github.com/JohannesBuchner/MultiNest
```

Note: MultiNest requires the libraries `lapack` and `mpi` (optional), and the compilation tool `cmake`.

We now follow the instructions in the `README` file to compile:

```
$ cd /path/to/MultiNest  # Folder where MultiNest was downloaded
$ cd build
$ cmake ..  # -DCMAKE_Fortran_COMPILER=gfortran
$ make
```

It is not necessary to install (`make install`), but it is so to add the folder in which the library was installed to the list of paths in which compilers will looll for libraries to link. In Linux (and other systems using a bash shell), this consists simply of adding at the end of the file `\home\<your user>\.bashrc` the line

```
export LD_LIBRARY_PATH=/path/to/MultiNest/lib${LD_LIBRARY_PATH:+:$LD_LIBRARY_PATH}
```

[1] arXiv:0704.3704, arXiv:0809.3437 and arXiv:1306.2144.
[2] arXiv:1402.0004.

By default, MultiNest will try to use the Intel Fortran Compiler if installed. If you want to use `gfortran` instead, add the flag `-DCMAKE_Fortran_COMPILER=gfortran` to `cmake`. If you want to force compilation with MPI support (though `gfortran` should autodetect it and implement it in most systems), use `mpif90` as compiler name.

4.1.2 2. PyMultiNest

Note: In a future implementation (hopefully soon), PyMultiNest will be installed automatically. Right now, the installations has to be done manually as it follows.

Download PyMultiNest from here, either using the releases page or cloning with git

```
$ git clone http://github.com/JohannesBuchner/PyMultiNest
```

Now go to the installation directory and install:

```
$ cd PyMultiNest
$ python setup.py install  # --user
```

You may need the flag `--user` in the last command if you do not have admin privileges.

If everything went ok, you should be able to run `import pymultinest` in a python console without getting any output. In that case, you are good to go!

4.2 Basic usage and parameters

The MultiNest sampling is invoked with the command line option `-m NS`. As in the MCMC case, the parameter file is read and the sampling is launched. The output files are created inside a subfolder `NS` inside the chain folder. This will create the expected `log.param` file inside the chain's root folder, and the expected raw MultiNest files in the `NS` subfolder (see MultiNest's `README`), along with two more files: `[chain name].paramnames`, which contains the ordering of the parameters in the nested sampling chain files (not necessarily the ordering in which they appear in the `log.param`, since clustering parameters must go first), and `[chain name].arguments`, which contains the user defined MultiNest arguments and their values (see below).

Note: If the sampling has been interrupted, simply run it again and MultiNest should be able to restart where it finished. If you intend to start a new sampling with different parameters for MultiNest, it is safer to delete the `NS` subfolder (otherwise, the behaviour is not well defined).

Note: MultiNest can benefit greatly from being run in parallel with MPI. If it has been correctly compiled with MPI (try to run the examples distributed with the MultiNest code with MPI), it is possible to take advantage of it using Monte Python: simply run the sampler `python MontePython.py` preceded by the appropriate MPI runner (`mpirun` for Open MPI, `mpiexec` for MPICH, etc.).

Once the sampling has finished, the output of it can be analised as in the MCMC case with `MontePython.py -info [chain_folder]/NS` (notice that one must specify the `NS` subfolder). This will create a chain file in the chain root folder containing the (accepted) points of the nested sampling, and it will be automatically analysed as a MCMC chain, producing the expected files and plots.

The MultiNest parameters are added after the `-m NS` flag in the command line. They are described in the next section (more thorough descriptions are to be looked for within the MultiNest documentation).

4.2.1 Automatic parameters

(Technical section, you can skip)

The following parameters are defined automatically by the content of the `.param` file, and you should not care about them:

- `ndims | n_dims` : number of varying parameters.
- `nPar | n_params` : number of varying parameters.
- `root | outputfiles_basename` : prefix of the MultiNest output files: name of the chain plus a hyphen.
- `outfile | write_output` : whether to write output files (yes, of course).
- `resume | resume` : whether to allow for resuming a previously killed run, enabled by default.
- `initMPI | init_MPI` : initialise MPI within MultiNest (disabled: MPI, if requested, is initialised by Monte Python).
- `feedback | verbose (True)` : print information periodically.

4.2.2 Manually set parameters

The following parameters can be changed by hand to adjust the sampling to one's needs. In the following, they are presented as

```
[MultiNest name] | [PyMultiNest name] (default value)
```

and are set in every run by command line options as

```
--NS_[PyMultiNest name] [value]
```

E.g. to set the number of "live points" to 100, one should add to the command `python MontePython.py [...] -m NS` the option

```
--NS_n_live_points 100
```

Note: The default values are those defined in PyMultiNest (at least most of them), and are not hard-coded in Monte Python.

Note: The parameters not appearing in the following lists are not managed in the current implementation.

General sampling options

- `nlive | n_live_points (400)` : number of points used in every iteration.
- `IS | importance_nested_sampling (True)` : whether to use Importance Nested Samplin (see arXiv:1306.2144).
- `efr | sampling_efficiency (0.8)` : defines the sampling efficiency (see 'Use cases' below).
- `ceff | const_efficiency_mode (True)` : constant efficiency mode – slower, but more accurate evidence estimation.
- `seed | seed (-1)`: seed of the random number generator (if negative, uses system clock).
- `logZero | log_zero (-1e90)` : if the log-likelihood of a sample is smaller than this value, the sample is ignored.

- `updInt | n_iter_before_update (100)` : number of iteration after which the output files are updated.

Ending conditions

- `tol | evidence_tolerance (0.5)`
- `maxiter | max_iter (0)`

The sampling ends after `maxiter` iterations, or when the tolerance condition on the evidence defined by `tol` is fulfilled, whatever happens first.

Multi-modal sampling

- `mmodal | multimodal (False)` : whether to try to find separate modes in the posterior.
- `maxModes | max_modes (100)` : maximum number of separate modes to consider.
- `Ztol | mode_tolerance (-1e90)` : if the local log-evidence is greater than this value, a mode is created.

Note: Here, multi-modal sampling is disabled by default. If enabled, Imporance Nested Sampling will be automatically disabled, since both modes are not compatible.

We left out the option concerning the *clustering parameters*, i.e. on which parameters's subspace is MultiNest to look for posterior mode separation:

```
nCdims | n_clustering_params
```

In (Py)MultiNest, clustering parameters are specified as the `n` first ones, which **must** be at the beginning of the parameters list. Here, instead, we override that limitation, and the clustering parameters are specified as

```
--NS_clustering_params param1 param2 ...
```

The reason for doing it this way is giving more flexibility to the user, being able to change the clustering parameters without having to modify the ordering of the parameters in the `param` file to put the clustering parameters at the beginnig. But this comes at a price: the raw MultiNest chain files have the parameters ordered with the clustering parameters at the beginning, and then the rest as they appear in the `.param` file. The ordering of the parameters is save to a file `[chain name].paramnames` in the `NS` subfolder. If you intend to use MustiNest's raw output files, you must take this into account! If, instead, you use nested sampling simply as a means to get a covariance matrix and some sample points (saved in `chain_NS__[accepted/rejected].txt`), you do not need to care about this.

4.3 References

CHAPTER 5

Using the Cosmo Hammer with Monte Python

Monte Python can now use the software Cosmo Hammer written by J. Akeret and S. Seehars, which is based on the emcee sampler, itself based on the Affine Invariant Markov Chain Monte Carlo

Note: By using the Cosmo Hammer, you agree to abide by the GNU General Public License v3.0 or higher (see their website)

Please look at their website for specifics about the installation.

5.1 Using with Monte Python

you can choose to use the Cosmo Hammer by specifying the argument: *-m CH*

You should probably always set the environment variable OMP_NUM_THREADS to your maximum number of cores:

```
$] export OMP_NUM_THREADS=4
```

before running.

CHAPTER 6

Existing likelihoods, and how to create new ones

This page is intended to explain in more concrete terms the information contained in the Likelihood class module documentation. More specifically, you should be able to write new likelihood files and understand the structure of existing ones.

6.1 One likelihood is one directory, one `.py` and one `.data` file

We have seen already that cosmological parameters are passed directly from the input file to *Class*, and do not appear anywhere in the code itself, i.e. in the files located in the `montepython/` directory. The situation is the same for likelihoods. You can write the name of a likelihood in the input file, and *Monte Python* will directly call one of the external likelihood codes implemented in the `montepython/likelihoods/` directory. This means that when you add some new likelihoods, you don't need to declare them in the code. You implement them in the `likelihoods` directory, and they are ready to be used if mentioned in the input file.

For This to work, a precise syntax must be respected. Each likelihood is associated to a name, e.g. `hst`, `wmap`, `WiggleZ` (the name is case-sensitive). This name is used:

- for calling the likelihood in the input file, e.g. `data.experiments = ['hst', ...]`,
- for naming the directory of the likelihood, e.g. `montepython/likelihoods/hst/`,
- for naming the input data file describing the characteristics of the experiment, `montepython/likelihoods/hst/hst.data` (this file can point to raw data files located in the `data` directory)
- for naming the class declared in `montepython/likelihoods/hst/__init__.py` and used also in `montepython/likelihoods/hst/hst.data`

Warning: Note that since release 2.0.0, the likelihood python source is not called any longer `hst.py`, but `__init__.py`. The reason was for packaging and ease of use when calling from a Python console.

When implementing new likelihoods, you will have to follow this rule. You could already wish to have two Hubble priors/likelihoods in your folder. For instance, the distributed version of `hst` corresponds to a gaussian prior with standard deviation $h = 0.738 \pm 0.024$. If you want to change these numbers, you can simply edit `montepython/likelihoods/hst/hst.data`. But you could also keep `hst` unchanged and create a new likelihood called e.g. `spitzer`. We will come back to the creation of likelihoods later, but just to illustrate the structure of likelihoods, let us see how to create such a prior/likelihood:

```
$ mkdir likelihoods/spitzer
$ cp likelihoods/hst/hst.data likelihoods/spitzer/spitzer.data
$ cp likelihoods/hst/__init__.py likelihoods/spitzer/__init__.py
```

Then edit `montepython/likelihoods/spitzer/__init__.py` and replace in the initial declaration the class name `hst` by `spitzer`:

```
class spitzer(Likelihood_prior):
```

Edit also `montepython/likelihoods/spitzer/spitzer.data`, replace the class name `hst` by `spitzer`, and the numbers by your constraint:

```
spitzer.h = 0.743
spitzer.sigma = 0.021
```

You are done. You can simply add `data.experiments = [...,'spitzer', ...]` to the list of experiments in the input parameter file and the likelihood will be used.

6.2 Existing likelihoods

We release the first version of *Monte Python* with the likelihoods:

- `spt`, `bicep`, `cbi`, `acbar`, `bicep`, `quad`, the latest public versions of CMB data from SPT, Bicep, CBI, ACBAR, BICEP and Quad; for the SPT likelihoods we include three nuisance parameters obeying to gaussian priors, like in the original SPT paper, and for ACBAR one nuisance parameter with top-hat prior. These experiments are described by the very same files as in a ComsoMC implementation. They are located in the `data/` directory. For each experiment, there is a master file `xxx.dataset` containing several variables and the names of other files with the raw data. In the files `likelihoods/xxx/xxx.data`, we just give the name of the different `xxx.dataset` files, that *Monte Python* is able to read just like CosmoMC.
- `wmap`, original likelihood file accessed through the wmap wrapper. The file `likelihoods/wmap/wmap.data` allows you to call this likelihood with a few different options (e.g. switching on/off Gibbs sampling, choosing the minimum and maximum multipoles to include, etc.) As usual, we implemented the nuisance parameter `A_SZ` with a flat prior. In the input parameter file, you can decide to vary this parameter in the range 0-2, or to fix it to some value.
- `hst` is the HST Key Project gaussian prior on h,
- `sn` constains the luminosity distance-redhsift relation using the Union 2 data compilation,
- `WiggleZ` constraints the matter power spectrum $P(k)$ in four different redshift bins using recent WiggleZ data,

plus a few other likelihoods referring to future experiments, described in the next subsection. All these likelihoods are strictly equivalent to those in the CosmoMC patches released by the various experimental collaborations.

6.3 Mock data likelihoods

We also release simplified likelihoods `fake_planck_bluebook`, `euclid_lensing` and `euclid_pk` for doing forecasts for Planck, Euclid (cosmic shear survey) and Euclid (redshift survey).

In the case of Planck, we use a simple gaussian likelihood for TT, TE, EE (like in astro-ph/0606227 with no lensing extraction) with sensitivity parameters matching the numbers published in the Planck bluebook. In the case of Euclid, our likelihoods and sensitivity parameters are specified in the Euclid Red Book. The sensitivity parameters can always be modified by the user, by simply editing the `.data` files.

These likelihoods compare theoretical spectra to a fiducial spectrum (and **not** to random data generated given the fiducial model: this approach is simpler and leads to the same forecast error bars, see this paper again).

Let us illustrate the way in which this works with `fake_planck_bluebook`, although the two Euclid likelihoods obey exactly to the same logic.

When you download the code, the file `montepython/likelihoods/fake_planck_bluebook/fake_planck_bluebook` has a field `fake_planck_bluebook.fiducial_file` pointing to the file `'fake_planck_bluebook_fiducial.dat'`. You downloaded this file together with the code: it is located in `data` and it contains the TT/TE/EE spectrum of a particular fiducial model (with parameter values logged in the first line of the file). If you launch a run with this likelihood, it will work immediately and fit the various models to this fiducial spectrum.

But you probably wish to choose your own fiducial model. This is extremely simple with *Monte Python*. You can delete the provided fiducial file\:code:*'fake_planck_bluebook_fiducial.dat'*, or alternatively, you can change the name of the fiducial file in `likelihoods/fake_planck_bluebook/fake_planck_bluebook.data`. When you start the next run, the code will notice that there is no input fiducial spectrum. It will then generate one automatically, write it in the correct file with the correct location, and stop after this single step. Then, you can launch new chains, they will fit this fiducial spectrum.

When you generate the fiducial model, you probably want to control exactly fiducial parameter values. If you start from an ordinary input file with no particular options, *Monte Python* will perform one random jump and generate the fiducial model. Fiducial parameter values will be logged in the first line of the fiducial file. But you did not choose them yourself. However, when you call *Monte Python* with the intention of generating a fiducial spectrum, you can pass the command line option `-f 0`. This sets the variance of the proposal density to zero. Hence the fiducial model will have precisely the parameter values specified in the input parameter file. The fiducial file is even logged in the `log.param` of all the runs that have been using it.

6.4 Creating new likelihoods belonging to pre-defined category

A likelihood is a class (let's call it generically `xxx`), declared and defined in `montepython/likelihoods/xxx/__init__.py`, using input numbers and input files names specified in `montepython/likelihoods/xxx/xxx.data`. The actual data files should usually be placed in the `data/` folder (with the exception of WMAP data). Such a class will always inherit from the properties of the most generic class defined inside `montepython/likelihoods_class.py`. But it may fall in the category of some pre-defined likelihoods and inherit more properties. In this case the coding will be extremely simple, you won't need to write a specific likelihood code.

In the current version, pre-defined classes are:

Likelihood_newdat suited for all CMB experiments described by a file in the `.newdat` format (same files as in CosmoMC).

Likelihood_mock_cmb suited for all CMB experiments dexcribed with a simplified gaussian likelihood, like our `fake_planck_bluebook` likelihood.

Likelihood_mpk suited for matter power spectrum data that would be described with a `.dataset` file in CosmoMC. This generic likelihood contains a piece of code following closely the routine `mpk` developped for CosmoMC. In the released version of *Monte Python*, this likelihood type is only used by each of the four redshift bins of the WiggleZ data, but it is almost ready for being used with other data set in this format.

Suppose, for instance, that a new CMB dataset `nextcmb` is released in the `.newdat` format. You will then copy the `.newdat` file and other related files (with window functions, etc.) in the folder `data/`. You will then create a new likelihood, starting from an existing one, e.g cbi:

```
$ mkdir likelihoods/nextcmb
$ cp likelihoods/cbi/cbi.data likelihoods/nextcmb/nextcmb.data
$ cp likelihoods/cbi/__init__.py likelihoods/nextcmb/__init__.py
```

The python file should only be there to tell the code that nextcmb is in the `.newdat` format. Hence it should only contain:

```
from montepython.likelihood_class import Likelihood_newdat
class nextcmb(Likelihood_newdat):
    pass
```

This is enough: the likelihood is fully defined. The data file should only contain the name of the `.newdat` file:

```
nextcmb.data_directory  = data.path['data']
nextcmb.file            = 'next-cmb-file.newdat'
```

Once you have edited these few lines, you are done! No need to tell *Monte Python* that there is a new likelihood! Just call it in your next run by adding `data.experiments = [...,'nextcmb', ...]` to the list of experiments in the input parameter file, and the likelihood will be used.

You can also define nuisance parameters, contamination spectra and nuisance priors for this likelihood, as explained in the next section.

6.5 Creating new likelihoods from scratch

The likelihood `sn` is an example of individual likelihood code: the actual code is explicitly written in `sn.py`. To create your own likelihood files, the best to is look at such examples and follow them. We do not provide a full tutorial here, and encourage you to ask for help if needed. Here are however some general indications.

Your customised likelihood should inherit from generic likelihood properties through:

```
from montepython.likelihood_class import Likelihood
class my-likelihood(Likelihood):
```

Implementing the likelihood amounts in developing in the python file `my-likelihood.py` the properties of two essential functions, `__init__` and `loglkl`. But you don't need to code everything from scratch, because the generic *likelihood* already knows the most generic steps. The previous link will give you all the functions defined from this base class, that your daughter class will inherit from. Here follows a detailled explanation about how to use these.

One thing is that you don't need to write from scratch the parser reading the `.data` file: this will be done automatically at the beginning of the initialization of your likelihood. Consider that any field defined with a line in the `.data` file, e.g. `my-likelihood.variance = 5`, are known in the likelihood code: in this example you could write in the python code something like `chi2+=result**2/self.variance`.

You don't need either to write from scratch an interface with *Class*. You just need to write somewhere in the initialization function some specific parameters that should be passed to *Class*. For instance, if you need the matter power spectrum, write

```
self.need_cosmo_arguments(data,{'output':'mPk'})
```

that uses the method *need_cosmo_arguments*. If this likelihood is used, the field `mPk` will be appended to the list of output fields (e.g. `output=tCl,pCl,mPk`), unless it was already there. If you write

```
self.need_cosmo_arguments(data,{'l_max_scalars':3300})
```

the code will check if `l_max_scalars` was already set at least to 3300, and if not, it will increase it to 3300. But if another likelihood needs more it will be more.

You don't need to redefine functions like for instance those defining the role of nuisance parameters (especially for CMB experiments). If you write in the `.data` file

```
my-likelihood.use_nuisance          = ['N1','N2']
```

the code will know that this likelihood cannot work if these two nuisance parameters are not specified in the parameter input file (they can be varying or fixed; fix them by writing a 0 in the sigma entry). If you try to run without them, the

code will stop with an explicit error message. If the parameter `N1` has a top-hat prior, no need to write it: just specify prior edges in the input parameter file. If `N2` has a gaussian prior, specify it in the `.data` file, e.g.:

```
my-likelihood.N2_prior_center  = 1
my-likelihood.N2_prior_variance = 2
```

Since these fields refer to pre-defined properties of the likelihood, you don't need to write explicitly in the code something like `chi2 += (N2-center)**2/variance`, adding the prior is done automatically. Finally, if these nuisance parameters are associated to a CMB dataset, they may stand for a multiplicative factor in front of a contamination spectrum to be added to the theoretical C_ℓ's. This is the case for the nuisance parameters of the `acbar`, `spt` and `wmap` likelihoods delivered with the code, so you can look there for concrete examples. To assign this role to these nuisance parameters, you just need to write

```
my-likelihood.N1_file = 'contamination_corresponding_to_N1.data'
```

and the code will understand what it should do with the parameter `N1` and the file `data/contamination_corresponding_to_N1.data`. Optionally, the factor in front of the contamination spectrum can be rescaled by a constant number using the syntax:

```
my-likelihood.N1_scale = 0.5
```

Creating new likelihoods requires a basic knowledge of python. If you are new in python, once you know the basics, you will realise how concise a code can be. You can compare the length of the likelihood codes that we provide with their equivalent in Fortran in the CosmoMC package.

CHAPTER 7

Documentation

This documentation was extracted directly from the code. The comments written as docstrings are automatically read and processed. This should prevent the documentation from straying too far from the current shape of the code.

Contents:

7.1 run Module

run.**add_covariance_matrix**(*command*)
Make sure that the command uses the covariance matrix from the folder

run.**from_run_to_info**(*command*)
Translate a command corresponding to a run into one for analysis

run.**mock_update_run**(*custom_command=''*)
Tentative covmat update run

Not reachable yet by any option.

run.**mpi_run**(*custom_command=''*)
Launch a simple MPI run, with no communication of covariance matrix

Each process will make sure to initialise the folder if needed. Then and only then, it will send the signal to its next in line to proceed. This allows for initialisation over an arbitrary cluster geometry (you can have a single node with many cores, and all the chains living there, or many nodes with few cores). The speed loss due to the time spend checking if the folder is created should be negligible when running decently sized chains.

Each process will send the number that it found to be the first available to its friends, so that the gathering of information post-run is made easier. If a chain number is specified, this will be used as the first number, and then incremented afterwards with the rank of the process.

run.**run**(*custom_command=''*)
Main call of the function

It recovers the initialised instances of cosmo Class, `Data` and the NameSpace containing the command line arguments, feeding into the sampler.

Parameters **custom_command** (*str*) – allows for testing the code

run.**safe_initialisation**(*custom_command=''*, *comm=None*, *nprocs=1*)
Wrapper around the init function to handle errors

Keyword Arguments

- **custom_command** (*str*) – testing purposes

- **comm** (*MPI.Intracomm*) – object that helps communicating between the processes
- **nprocs** (*int*) – number of processes

7.2 Initialise Module

initialise.**initialise**(*custom_command=''*)
Initialisation routine

This function recovers the input from the command line arguments, from *parser_mp*, the parameter files.

It then extracts the path of the used Monte Python code, and proceeds to initialise a *data* instance, a cosmological code instance.

Parameters **custom_command** (*str*) – allows for testing the code

initialise.**recover_cosmological_module**(*data*)
From the cosmological module name, initialise the proper Boltzmann code

Note: Only CLASS is currently wrapped, but a python wrapper of CosmoMC should enter here.

initialise.**recover_local_path**(*command_line*)
Read the configuration file, filling a dictionary

Returns **path** (*dict*) – contains the absolute path to the location of the code, the data, the cosmological code, and potential likelihood codes (clik for Planck, etc)

7.3 Parser module

Defines the command line options and their help messages in *create_parser()* and read the input command line in *parse()*, dealing with different possible configurations.

The fancy short/long help formatting, as well as the automatic help creation from docstrings is entirely due to Francesco Montesano.

class parser_mp.**MpArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *version=None*, *parents=[]*, *formatter_class=<class 'argparse.HelpFormatter'>*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*)
Bases: argparse.ArgumentParser

Extension of the default ArgumentParser

error(*message*)
Override method to raise error :Parameters: **message** (*string*) –

error message

safe_parse_args(*args=None*)
Allows to set a default subparser

This trick is there to maintain the previous way of calling MontePython.py

set_default_subparser(*default*, *args=None*)
If no subparser option is found, add the default one

Note: This function relies on the fact that all calls to MontePython will start with a -. If this came to

change, this function should be revisited

---

parser_mp.**add_subparser**(*sp*, *name*, ***kwargs*)

Add a parser to the subparser *sp* with *name*.

All the logic common to all subparsers should go here

Parameters

- **sp** (*subparser instance*)
- **name** (*str*) – name of the subparser
- **kwargs** (*dict*) – keywords to pass to the subparser
- **output**
- **——**
- **sparser** (*Argparse instance*) – new subparser

parser_mp.**create_parser**()

Definition of the parser command line options

The main parser has so far two subparsers, corresponding to the two main modes of operating the code, namely *run* and *info*. If you simply call `python montepython/MontePython.py -h`, you will find only this piece of information. To go further, and find the command line options specific to these two submodes, one should then do: `python montepython/MontePython.py run -h`, or `info -h`.

All command line arguments are defined below, for each of the two subparsers. This function create the automatic help command.

Each flag outputs the following argument to a destination variable, specified by the *dest* keyword argument in the source code. Please check there to understand the variable names associated with each option.

run

-N [int] number of steps in the chain (**OBL**). Note that when running on a cluster, your run might be stopped before reaching this number.

-o [str] output folder (**OBL**). For instance `-o chains/myexperiments/mymodel`. Note that in this example, the folder `chains/myexperiments` must already exist.

-p [str] input parameter file (**OBL**). For example `-p input/exoticmodel.param`.

-c [str] input covariance matrix (*OPT*). A covariance matrix is created when analyzing previous runs.

Note that the list of parameters in the input covariance matrix and in the run do not necessarily coincide.

-j [str] jumping method (*global* (default), *sequential* or *fast*) (*OPT*).

With the *global* method the code generates a new random direction at each step, with the *sequential* one it cycles over the eigenvectors of the proposal density (= input covariance matrix).

The *global* method the acceptance rate is usually lower but the points in the chains are less correlated. We recommend using the sequential method to get started in difficult cases, when the proposal density is very bad, in order to accumulate points and generate a covariance matrix to be used later with the *default* jumping method.

The *fast* method implements the Cholesky decomposition presented in http://arxiv.org/abs/1304.4473 by Antony Lewis.

-m [str] sampling method, by default 'MH' for Metropolis-Hastings, can be set to 'NS' for Nested Sampling (using Multinest wrapper PyMultiNest), 'CH' for Cosmo Hammer (using the Cosmo Hammer wrapper to emcee algorithm), and finally 'IS' for importance sampling.

Note that when running with Importance sampling, you need to specify a folder to start from.

–update [int] update frequency for Metropolis Hastings. If greater than zero, number of steps after which the proposal covariance matrix is updated automatically (*OPT*).

-f [float] jumping factor (>= 0, default to 2.4) (*OPT*).

The proposal density is given by the input covariance matrix (or a diagonal matrix with elements given by the square of the input sigma's) multiplied by the square of this factor. In other words, a typical jump will have an amplitude given by sigma times this factor.

The default is the famous factor 2.4, advertised by Dunkley et al. to be an optimal trade-off between high acceptance rate and high correlation of chain elements, at least for multivariate gaussian posterior probabilities. It can be a good idea to reduce this factor for very non-gaussian posteriors.

Using `-f 0 -N 1` is a convenient way to get the likelihood exactly at the starting point passed in input.

–conf [str] configuration file (default to *default.conf*) (*OPT*). This file contains the path to your cosmological module directory.

–chain-number [str] arbitrary number of the output chain, to overcome the automatic one (*OPT*).

By default, the chains are named `yyyy-mm-dd_N__i.txt` with year, month and day being extracted, `N` being the number of steps, and `i` an automatically updated index.

This means that running several times the code with the same command will create different chains automatically.

This option is a way to enforce a particular number `i`. This can be useful when running on a cluster: for instance you may ask your script to use the job number as `i`.

-r [str] restart from last point in chain, to avoid the burn-in stage (*OPT*).

At the beginning of the run, the previous chain will be deleted, and its content transfered to the beginning of the new chain.

-b [str] start a new chain from the bestfit file computed with analyze. (*OPT*)

–fisher [None] Calculates the inverse of the fisher matrix to use as proposal distribution

–silent [None] silence the standard output (useful when running on clusters)

–Der-target-folder [str] Add additional derived params to this folder. It has to be used in conjunction with *Der-param-list*, and the method set to Der: `-m Der`. (*OPT*)

–Der-param-list [str] Specify a number of derived parameters to be added. A complete example would be to add Omega_Lambda as a derived parameter: `python montepython/MontePython.py run -o existing_folder -m Der --Der-target-folder non_existing_folder --Der-param-list Omega_Lambda`

–IS-starting-folder [str] Perform Importance Sampling from this folder or set of chains (*OPT*)

For Nested Sampling and Cosmo Hammer arguments, see `nested_sampling` and `cosmo_hammer`.

info

Replaces the old **-info** command, which is deprecated but still available.

files [string/list of strings] you can specify either single files, or a complete folder, for example `info chains/my-run/2012-10-26*`, or `info chains/my-run`.

If you specify several folders (or set of files), a comparison will be performed.

–minimal [None] use this flag to avoid computing the posterior distribution. This will decrease the time needed for the analysis, especially when analyzing big folders.

–bins [int] number of bins in the histograms used to derive posterior probabilities and credible intervals (default to 20). Decrease this number for smoother plots at the expense of masking details.

–no-mean [None] remove the mean likelihood from the plot. By default, when plotting marginalised 1D posteriors, the code also shows the mean likelihood per bin with dashed lines; this flag switches off the dashed lines.

–extra [str] extra file to customize the output plots. You can actually set all the possible options in this file, including line-width, ticknumber, ticksize, etc... You can specify four fields, *info.redefine* (dict with keys set to the previous variable, and the value set to a numerical computation that should replace this variable), *info.to_change* (dict with keys set to the old variable name, and value set to the new variable name), *info.to_plot* (list of variables with new names to plot), and *info.new_scales* (dict with keys set to the new variable names, and values set to the number by which it should be multiplied in the graph). For instance,

```
info.to_change={'oldname1':'newname1','oldname2':'newname2',...}
info.to_plot=['name1','name2','newname3',...]
info.new_scales={'name1':number1,'name2':number2,...}
```

–noplot [bool] do not produce any plot, simply compute the posterior (*OPT*) (flag)

–noplot-2d [bool] produce only the 1d posterior plot (*OPT*) (flag)

–contours-only [bool] do not fill the contours on the 2d plots (*OPT*) (flag)

–all [None] output every subplot and data in separate files (*OPT*) (flag)

–ext [str] change the extension for the output file. Any extension handled by `matplotlib` can be used. (*pdf* (default), *png* (faster))

–fontsize [int] desired fontsize (default to 16)

–ticksize [int] desired ticksize (default to 14)

–line-width [int] set line width (default to 4)

–decimal [int] number of decimal places on ticks (default to 3)

–ticknumber [int] number of ticks on each axis (default to 3)

–legend-style [str] specify the style of the legend, to choose from *sides* or *top*.

–keep-non-markovian [bool] Use this flag to keep the non-markovian part of the chains produced at the beginning of runs with –update mode This option is only relevant when the chains were produced with –update (*OPT*) (flag)

–keep-fraction [float] after burn-in removal, analyze only last fraction of each chain. (between 0 and 1). Normally one would not use this for runs with –update mode, unless –keep-non-markovian is switched on (*OPT*)

–want-covmat [bool] calculate the covariant matrix when analyzing the chains. Warning: this will interfere with ongoing runs utilizing update mode (*OPT*) (flag)

Returns **args** (*NameSpace*) – parsed input arguments

parser_mp.**custom_help**(*split_string='<++>'*)
Create a custom help action.

It expects *split_string* to appear in groups of three. If the option string is '-h', then uses the short description between the first two *split_string*. If the option string is '-h', then uses all that is between the first and the third *split_string*, stripping the first one.

Parameters

- **split_string** (*str*) – string to use to select the help string and how to select them. They must appear in groups of *3*
- **output**
- **——**
- **CustomHelp** (*class definition*)

parser_mp.**existing_file**(*fname*)
Check if the file exists. If not raise an error

Parameters **fname** (*string*) – file name to parse

Returns **fname** (*string*)

parser_mp.**get_dict_from_docstring**(*key_symbol='<**>'*, *description_symbol='<++>'*)
Create the decorator

Parameters

- **key_symbol** (*str*) – identifies the key of a argument/option
- **description_symbol** (*str*) – identify the description of a argument/option
- **Returns**
- **——**
- **wrapper** (*function*)

parser_mp.**initialise_parser**(***kwargs*)
Create the argument parser and returns it :Parameters: * **kwargs** (*dictionary*) –

keyword to pass to the parser

•**output**

•**——**

•**p** (*MpArgumentParser instance*) – parser with some keyword added

parser_mp.**parse**(*custom_command=''*)
Check some basic organization of the folder, and exit the program in case something goes wrong.

Keyword Arguments **custom_command** (*str*) – For testing purposes, instead of reading the command line argument, read instead the given string. It should ommit the start of the command, so e.g.: '-N 10 -o toto/'

parser_mp.**parse_docstring**(*docstring*, *key_symbol='<**>'*, *description_symbol='<++>'*)
Extract from the docstring the keys and description, return it as a dict

Parameters

- **docstring** (*str*)
- **key_symbol** (*str*) – identifies the key of an argument/option

- **description_symbol** (*str*) – identify the description of an argument/option
- **output**
- **——**
- **helpdic** (*dict*) – help strings for the parser

parser_mp.**positive_int**(*string*)

Check if the input is integer positive :Parameters: * **string** (*string*) –

string to parse

•**output** (*int*) – return the integer

7.4 Data module

class data.**Data**(*command_line*, *path*)

Bases: object

Store all relevant data to communicate between the different modules.

The Data class holds the cosmological information, the parameters from the MCMC run, the information coming from the likelihoods. It is a wide collections of information, with in particular two main dictionaries: cosmo_arguments and mcmc_parameters.

It defines several useful **methods**. The following ones are called just once, at initialization:

- *fill_mcmc_parameters()*
- *read_file()*
- *read_version()*
- *group_parameters_in_blocks()*

On the other hand, these two following functions are called every step.

- *check_for_slow_step()*
- *update_cosmo_arguments()*

Finally, the convenient method *get_mcmc_parameters()* will be called in many places, to return the proper list of desired parameters.

It has a number of different **attributes**, and the more important ones are listed here:

- *boundary_loglike*
- *cosmo_arguments*
- *mcmc_parameters*
- *need_cosmo_update*
- *log_flag*

Note: The *experiments* attribute is extracted from the parameter file, and contains the list of likelihoods to use

Note: The path argument will be used in case it is a first run, and hence a new folder is created. If starting from an existing folder, this dictionary will be compared with the one extracted from the log.param, and will use the latter while warning the user.

Warning: New in version 2.0.0, you can now specify an oversampling of the nuisance parameters, to hasten the execution of a run with likelihoods that have many of them. You should specify a new field in the parameter file, *data.over_sampling = [1, ...]*, that contains a 1 on the first element, and then the over sampling of the desired likelihoods. This array must have the same size as the number of blocks (1 for the cosmo + 1 for each likelihood with varying nuisance parameters). You need to call the code with the flag *-j jast* for it to be used.

To create an instance of this class, one must feed the following parameters and keyword arguments:

Parameters

- **command_line** (*NameSpace*) – NameSpace containing the input from the *parser_mp*. It stores the input parameter file, the jumping methods, the output folder, etc... Most of the information extracted from the command_file will be transformed into *Data* attributes, whenever it felt meaningful to do so.
- **path** (*dict*) – Contains a dictionary of important local paths. It is used here to find the cosmological module location.

boundary_loglike = None
Define the boundary loglike, the value used to defined a loglike that is out of bounds. If a point in the parameter space is affected to this value, it will be automatically rejected, hence increasing the multiplicity of the last accepted point.

cosmo_arguments = None
Simple dictionary that will serve as a communication interface with the cosmological code. It contains all the parameters for the code that will not be set to their default values. It is updated from *mcmc_parameters*.

Return type dict

mcmc_parameters = None
Ordered dictionary of dictionaries, it contains everything needed by the *mcmc* module for the MCMC procedure. Every parameter name will be the key of a dictionary, containing the initial configuration, role, status, last accepted point and current point.

Return type ordereddict

NS_arguments = None
Dictionary containing the parameters needed by the PyMultiNest sampler. It is filled just before the run of the sampler. Those parameters not defined will be set to the default value of PyMultiNest.

Return type dict

over_sampling = None
List storing the respective over sampling of the parameters. The first entry, applied to the cosmological parameters, will always be 1. Setting it to anything else would simply rescale the whole process. If not specified otherwise in the parameter file, all other numbers will be set to 1 as well.

Return type list

need_cosmo_update = None
added in version 1.1.1. It stores the truth value of whether the cosmological block of parameters was changed from one step to another. See *group_parameters_in_blocks()*

Return type bool

log_flag = None
Stores the information whether or not the likelihood data files need to be written down in the log.param file. Initially at False.

Return type bool

fill_mcmc_parameters()
Initializes the ordered dictionary *mcmc_parameters* from the input parameter file.

It uses *read_file()*, and initializes instances of `parameter` to actually fill in *mcmc_parameters*.

initialise_likelihoods(*experiments*)
Given an array of experiments, return an ordered dict of instances

---

Note: in the __init__ method, experiments is naturally self.experiments, but it is useful to keep it as a parameter, for the case of importance sampling.

---

read_file(*param*, *structure*, *field=''*, *separate=False*)
Execute all lines concerning the Data class from a parameter file

All lines starting with *data.* will be replaced by *self.*, so the current instance of the class will contain all the information.

Note: A rstrip() was added at the end, because of an incomprehensible bug on some systems that imagined some inexistent characters at the end of the line... Now should work

---

Note: A security should be added to protect from obvious attacks.

Parameters

- **param** (*str*) – Name of the parameter file
- **structure** (*str*) – Name of the class entries we want to execute (mainly, data, or any other likelihood)

Keyword Arguments

- **field** (*str*) – If nothing is specified, this routine will execute all the lines corresponding to the *structure* parameters. If you specify a specific field, like *path*, only this field will be read and executed.
- **separate** (*bool*) – If this flag is set to True, a container class will be created for the structure field, so instead of appending to the namespace of the data instance, it will append to a sub-namespace named in the same way that the desired structure. This is used to extract custom values from the likelihoods, allowing to specify values for the likelihood directly in the parameter file.

group_parameters_in_blocks()
Regroup mcmc parameters by blocks of same speed

This method divides all varying parameters from *mcmc_parameters* into as many categories as there are likelihoods, plus one (the slow block of cosmological parameters).

It creates the attribute `block_parameters`, to be used in the module *mcmc*.

Note: It does not compute by any mean the real speed of each parameter, instead, every parameter belonging to the same likelihood will be considered as fast as its neighbour.

> **Warning:** It assumes that the nuisance parameters are already written sequentially, and grouped together (not necessarily in the order described in `experiments`). If you mix up the different nuisance parameters in the .param file, this routine will not method as intended. It also assumes that the cosmological parameters are written at the beginning of the file.

assign_over_sampling_indices()
Create the list of varied parameters given the oversampling

read_version(*param_file*)
Extract version and subversion from an existing log.param

get_mcmc_parameters(*table_of_strings*)
Returns an ordered array of parameter names filtered by *table_of_strings*.

Parameters **table_of_strings** (*list*) – List of strings whose role and status must be matched by a parameter. For instance,

```
>>> data.get_mcmc_parameters(['varying'])
['omega_b', 'h', 'amplitude', 'other']
```

will return a list of all the varying parameters, both cosmological and nuisance ones (derived parameters being *fixed*, they wont be part of this list). Instead,

```
>>> data.get_mcmc_parameters(['nuisance', 'varying'])
['amplitude', 'other']
```

will only return the nuisance parameters that are being varied.

check_for_slow_step(*new_step*)
Check whether the value of cosmological parameters were changed, and if no, skip computation of the cosmology.

update_cosmo_arguments()
Put in *cosmo_arguments* the current values of *mcmc_parameters*

This method is called at every step in the Markov chain, to update the dictionary. In the Markov chain, the scale is not remembered, so one has to apply it before giving it to the cosmological code.

Note: When you want to define new parameters in the Markov chain that do not have a one to one correspondance to a cosmological name, you can redefine its behaviour here. You will find in the source several such examples.

Note: For complex CLASS parameters, that expect a string of numbers separated with commas, you can now use the name of the argument, for instance `m_ncdm`, then append a double underscore and a number. So if you run with two cosmological parameters, `m_ncdm__1` and `m_ncdm__2`, this function will automatically concatenate the two and feed class `m_ncdm`. You still have to make sure that the other variables are properly set, like `N_ncdm` to 2, in this example.

---

static folder_is_initialised(*folder*)
Static method to call for checking if a folder was already initialised

This method can be used to speed up the mpi initialisation in *run*. If a process finds that the folder is already a proper Monte Python one, it sends directly a 'go' signal to its next in line.

> **Warning:** This method assumes that the last lines of the log.param are the path indication. If this would ever change, adjust this method accordingly.

__cmp__(*other*)
Redefinition of the 'compare' method for two instances of this class.

It will decide which basic operations to perform when the code asked if two instances are the same (in case you want to launch a new chain in an existing folder, with your own parameter file) Comparing cosmological code versions (warning only, will not fail the comparison)

__call__(*ctx*)
Interface layer with CosmoHammer

Store quantities to a the context, to be accessed by the Cosmo Module and each of the likelihoods.

Parameters **ctx** (*context*) – Contains several dictionaries storing data and cosmological information

class data.**Parameter**(*array*, *key*)
Bases: dict

Store all important fields, and define a few convenience methods

This class replaces the old function defined in the Data class, called *from_input_to_mcmc_parameters*. The traduction is now done inside the Parameter class, which interprets the array given as an input inside the parameter file, and returns a dictionary having all relevant fields initialized.

Warning: This used to be an ordered dictionary, for no evident reason. It is now reverted back to an ordinary dictionary. If this broke anything, it will be reverted back

At the end of this initialization, every field but one is filled for the specified parameter, be it fixed or varying. The missing field is the 'last_accepted' one, that will be filled in the module *mcmc*.

Note: The syntax of the parameter files is defined here - if one wants to change it, one should report the changes in there.

The other fields are

Variables

- **initial** (*array*) – Initial array of input values defined in the parameter file. Contains (in this order) *mean*, *minimum*, *maximum*, *1-sigma*. If the min/max values (**TO CHECK** proposal density boundaries) are unimportant/unconstrained, use *None* or *-1* (without a period !)
- **scale** (*float*) – 5th entry of the initial array in the parameter file, defines the factor with which to multiply the values defined in *initial* to give the real value.
- **role** (*str*) – 6th entry of the initial array, can be *cosmo*, *nuisance* or *derived*. A *derived* parameter will not be considered as varying, but will be instead recovered from the cosmological code for each point in the parameter space.
- **prior** (*Prior*) – defined through the optional 7th entry of the initial array, can be ommited or set to *flat* (same), or set to *gaussian*. An instance of the *prior* defined in *prior* will be initialized and set to this value.
- **tex_name** (*str*) – A tentative tex version of the name, provided by the function *io_mp.get_tex_name()*.
- **status** (*str*) – Depending on the *1-sigma* value in the initial array, it will be set to *fixed* or *varying* (resp. zero and non-zero)
- **current** (*float*) – Stores the value at the current point in parameter space (*not allowed initially*)

Parameters

- **value** (*list*) – Array read from the parameter file
- **key** (*str*) – Name of the parameter

7.5 Prior module

class prior.**Prior**(*array*)
Bases: object

Store the type of prior associated to a parameter

It takes as an optional input argument the array of the input parameters defined in the parameter file.

The current implemented types are 'flat' (default), and 'gaussian', which expect also a mean and sigma. Possible extension would take a 'external', needing to read an external file to read for the definition.

The entry 'prior' of the dictionary mcmc_parameters will hold an instance of this class. It defines one main function, called *draw_from_prior()*, that returns a number within the prior volume.

draw_from_prior()
Draw a random point from the prior range considering the prior type

Returns **value** (*float*) – A random sample inside the prior region

value_within_prior_range(*value*)
Check for a value being in or outside the prior range

is_bound()
Checks whether the allowed parameter range is finite

map_from_unit_interval(*value*)
Linearly maps a value of the interval [0,1] to the parameter range.

For the sake of speed, assumes the parameter to be bound to a finite range, which should have been previously checked with *is_bound()*

7.6 Likelihood class module

Contains the definition of the base likelihood class *Likelihood*, with basic functions, as well as more specific likelihood classes that may be reused to implement new ones.

class likelihood_class.**Likelihood**(*path*, *data*, *command_line*)
Bases: object

General class that all likelihoods will inherit from.

It copies the content of self.path from the initialization routine of the *Data* class, and defines a handful of useful methods, that every likelihood might need.

If the nuisance parameters required to compute this likelihood are not defined (either fixed or varying), the code will stop.

Parameters

- **data** (*class*) – Initialized instance of *Data*
- **command_line** (*NameSpace*) – NameSpace containing the command line arguments

add_contamination_spectra(*cl*, *data*)

add_nuisance_prior(*lkl*, *data*)

computeLikelihood(*ctx*)
Interface with CosmoHammer

Parameters **ctx** (*Context*) – Contains several dictionaries storing data and cosmological information

get_cl(*cosmo*, *l_max=-1*)
Return the C_ℓ from the cosmological code in $\mu\mathrm{K}^2$

loglkl(*cosmo*, *data*)
Placeholder to remind that this function needs to be defined for a new likelihood.

Raises `NotImplementedError`

need_cosmo_arguments(*data*, *dictionary*)
Ensure that the arguments of dictionary are defined to the correct value in the cosmological code

Warning: So far there is no way to enforce a parameter where *smaller is better*. A bigger value will always overried any smaller one (*cl_max*, etc...)

Parameters

- **data** (*dict*) – Initialized instance of *data*
- **dictionary** (*dict*) – Desired precision for some cosmological parameters

read_contamination_spectra(*data*)

read_from_file(*path*, *data*, *command_line*)
Extract the information from the log.param concerning this likelihood.

If the log.param is used, check that at least one item for each likelihood is recovered. Otherwise, it means the log.param does not contain information on the likelihood. This happens when the first run fails early, before calling the likelihoods, and the program did not log the information. This check might not be completely secure, but it is better than nothing.

Warning: This checks relies on the fact that a likelihood should always have at least **one** line of code written in the likelihood.data file. This should be always true, but in case a run fails with the error message described below, think about it.

Warning: As of version 2.0.2, you can specify likelihood options in the parameter file. They have complete priority over the ones specified in the *likelihood.data* file, and it will be reflected in the *log.param* file.

class likelihood_class.**Likelihood_clik**(*path*, *data*, *command_line*)
Bases: *likelihood_class.Likelihood*

loglkl(*cosmo*, *data*)

class likelihood_class.**Likelihood_clocks**(*path*, *data*, *command_line*)
Bases: *likelihood_class.Likelihood*

Base implementation of H(z) measurements

loglkl(*cosmo*, *data*)

class likelihood_class.**Likelihood_mock_cmb**(*path*, *data*, *command_line*)
Bases: *likelihood_class.Likelihood*

compute_lkl(*cl*, *cosmo*, *data*)

loglkl(*cosmo*, *data*)

class likelihood_class.**Likelihood_mpk**(*path*, *data*, *command_line*, *common=False*, *common_dict={}*)
Bases: *likelihood_class.Likelihood*

add_common_knowledge(*common_dictionary*)
Add to a class the content of a shared dictionary of attributes

The purpose of this method is to set some attributes globally for a Pk likelihood, that are shared amongst all the redshift bins (in WiggleZ.data for instance, a few flags and numbers are defined that will be transfered to wigglez_a, b, c and d

loglkl(*cosmo*, *data*)

class likelihood_class.**Likelihood_newdat**(*path*, *data*, *command_line*)
Bases: *likelihood_class.Likelihood*

compute_lkl(*cl*, *cosmo*, *data*)

loglkl(*cosmo*, *data*)

class likelihood_class.**Likelihood_prior**(*path*, *data*, *command_line*)
Bases: *likelihood_class.Likelihood*

It copies the content of self.path from the initialization routine of the *Data* class, and defines a handful of useful methods, that every likelihood might need.

If the nuisance parameters required to compute this likelihood are not defined (either fixed or varying), the code will stop.

Parameters

- **data** (*class*) – Initialized instance of *Data*
- **command_line** (*NameSpace*) – NameSpace containing the command line arguments

loglkl()

class likelihood_class.**Likelihood_sn**(*path*, *data*, *command_line*)
Bases: *likelihood_class.Likelihood*

read_configuration_file()
Extract Python variables from the configuration file

This routine performs the equivalent to the program "inih" used in the original c++ library.

read_light_curve_parameters()
Read the file jla_lcparams.txt containing the SN data

Note: the length of the resulting array should be equal to the length of the covariance matrices stored in C00, etc...

---

read_matrix(*path*)
extract the matrix from the path

This routine uses the blazing fast pandas library (0.10 seconds to load a 740x740 matrix). If not installed, it uses a custom routine that is twice as slow (but still 4 times faster than the straightforward numpy.loadtxt method)

Note: the length of the matrix is stored on the first line... then it has to be unwrapped. The pandas routine read_table understands this immediatly, though.

7.7 Sampler module

This module defines one key function, *run()*, that distributes the work to the desired actual sampler (Metropolis Hastings, or Nested Sampling so far).

It also defines a serie of helper functions, that aim to be generically used by all different sampler methods:

- *get_covariance_matrix()*
- *read_args_from_chain()*
- *read_args_from_bestfit()*
- *accept_step()*
- *compute_lkl()*

sampler.**accept_step**(*data*)
Transfer the 'current' point in the varying parameters to the last accepted one.

sampler.**check_flat_bound_priors**(*parameters*, *names*)
Ensure that all varying parameters are bound and flat

It is a necessary condition to use the code with Nested Sampling or the Cosmo Hammer.

sampler.**compute_fisher**(*data*, *cosmo*, *center*, *step_size*)

sampler.**compute_fisher_element**(*data*, *cosmo*, *center*, *one*, *two=None*)

sampler.**compute_lkl**(*cosmo*, *data*)
Compute the likelihood, given the current point in parameter space.

This function now performs a test before calling the cosmological model (**new in version 1.2**). If any cosmological parameter changed, the flag `data.need_cosmo_update` will be set to `True`, from the routine *check_for_slow_step*.

Returns

loglike (*float*) – The log of the likelihood ($\frac{-\chi^2}{2}$) computed from the sum of the likelihoods of the experiments specified in the input parameter file.

This function returns `data.boundary_loglkie`, defined in the module *data* if *i)* the current point in the parameter space has hit a prior edge, or *ii)* the cosmological module failed to compute the model. This value is chosen to be extremly small (large negative value), so that the step will always be rejected.

sampler.**get_covariance_matrix**(*cosmo*, *data*, *command_line*)
Compute the covariance matrix, from an input file or from an existing matrix.

Reordering of the names and scaling take place here, in a serie of potentially hard to read methods. For the sake of clarity, and to avoid confusions, the code will, by default, print out a succession of 4 covariance matrices at the beginning of the run, if starting from an existing one. This way, you can control that the paramters are set properly.

Note: The set of parameters from the run need not to be the exact same set of parameters from the existing covariance matrix (not even the ordering). Missing parameter from the existing covariance matrix will use the sigma given as an input.

---

sampler.**read_args_from_bestfit**(*data*, *bestfit*)
Deduce the starting point either from the input file, or from a best fit file.

Parameters **bestfit** (*str*) – Name of the bestfit file from the command line.

sampler.**read_args_from_chain**(*data*, *chain*)
Pick up the last accepted values from an input chain as a starting point

Function used only when the restart flag is set. It will simply read the last line of an input chain, using the tail command from the extended *io_mp.File* class.

Warning: That method was not tested since the adding of derived parameters. The method *read_args_from_bestfit()* is the prefered one.

Warning: This method works because of the particular presentation of the chain, and the use of tabbings (not spaces). Please keep this in mind if you are having difficulties

Parameters **chain** (*str*) – Name of the input chain provided with the command line.

sampler.**run**(*cosmo*, *data*, *command_line*)
Depending on the choice of sampler, dispatch the appropriate information

The *mcmc* module is used as previously, except the call to *mcmc.chain()*, or `nested_sampling.run()` is now within this function, instead of from within `MontePython`.

In the long term, this function should contain any potential hybrid scheme.

7.8 Mcmc module

This module defines one key function, *chain()*, that handles the Markov chain. So far, the code uses only one chain, as no parallelization is done.

The following routine is also defined in this module, which is called at every step:

- *get_new_position()* returns a new point in the parameter space, depending on the proposal density.

The *chain()* in turn calls several helper routines, defined in *sampler*. These are called just once:

- *compute_lkl()* is called at every step in the Markov chain, returning the likelihood at the current point in the parameter space.
- *get_covariance_matrix()*
- *read_args_from_chain()*
- *read_args_from_bestfit()*
- *accept_step()*

Their usage is described in *sampler*. On the contrary, the following routines are called at every step:

The arguments of these functions will often contain **data** and/or **cosmo**. They are both initialized instances of respectively *data* and the cosmological class. They will thus not be described for every function.

mcmc.**chain**(*cosmo*, *data*, *command_line*)
Run a Markov chain of fixed length with a Metropolis Hastings algorithm.

Main function of this module, this is the actual Markov chain procedure. After having selected a starting point in parameter space defining the first **last accepted** one, it will, for a given amount of steps :

•choose randomnly a new point following the *proposal density*,

•compute the cosmological *observables* through the cosmological module,

•compute the value of the *likelihoods* of the desired experiments at this point,

•*accept/reject* this point given its likelihood compared to the one of the last accepted one.

Every time the code accepts `data.write_step` number of points (quantity defined in the input parameter file), it will write the result to disk (flushing the buffer by forcing to exit the output file, and reopen it again.

Note: to use the code to set a fiducial file for certain fixed parameters, you can use two solutions. The first one is to put all input 1-sigma proposal density to zero (this method still works, but is not recommended anymore). The second one consist in using the flag "-f 0", to force a step of zero amplitude.

---

mcmc.**get_new_position**(*data*, *eigv*, *U*, *k*, *Cholesky*, *Rotation*)

Obtain a new position in the parameter space from the eigen values of the inverse covariance matrix, or from the Cholesky decomposition (original idea by Anthony Lewis, in Efficient sampling of fast and slow cosmological parameters)

The three different jumping options, decided when starting a run with the flag **-j** are **global**, **sequential** and **fast** (by default) (see *parser_mp* for reference).

Warning: For running Planck data, the option **fast** is highly recommended, as it speeds up the convergence. Note that when using this option, the list of your likelihoods in your parameter file **must match** the ordering of your nuisance parameters (as always, they must come after the cosmological parameters, but they also must be ordered between likelihood, with, preferentially, the slowest likelihood to compute coming first).

- **global**: varies all the parameters at the same time. Depending on the input covariance matrix, some degeneracy direction will be followed, otherwise every parameter will jump independently of each other.
- **sequential**: varies every parameter sequentially. Works best when having no clue about the covariance matrix, or to understand which estimated sigma is wrong and slowing down the whole process.
- **fast**: privileged method when running the Planck likelihood. Described in the aforementioned article, it separates slow (cosmological) and fast (nuisance) parameters.

Parameters

- **eigv** (*numpy array*) – Eigenvalues previously computed
- **U** (*numpy_array*) – Covariance matrix.
- **k** (*int*) – Number of points so far in the chain, is used to rotate through parameters
- **Cholesky** (*numpy array*) – Cholesky decomposition of the covariance matrix, and its inverse
- **Rotation** (*numpy_array*) – Not used yet

7.9 Nested Sampling module

7.10 Cosmo Hammer module

7.11 Analyze module

Collection of functions needed to analyze the Markov chains.

This module defines as well a class *Information*, that stores useful quantities, and shortens the argument passing between the functions.

Note: Some of the methods used in this module are directly adapted from the CosmoPmc code from Kilbinger et. al.

---

analyze.**analyze**(*command_line*)
Main function, does the entire analysis.

It calls in turn all the other routines from this module. To limit the arguments of each function to a reasonnable size, a *Information* instance is used. This instance is initialized in this function, then appended by the other routines.

analyze.**prepare**(*files*, *info*)
Scan the whole input folder, and include all chains in it.

Since you can decide to analyze some file(s), or a complete folder, this function first needs to separate between the two cases.

Warning: If someday you change the way the chains are named, remember to change here too, because this routine assumes the chains have a double underscore in their names.

Note: Only files ending with .txt will be selected, to keep compatibility with CosmoMC format

Note: New in version 2.0.0: if you ask to analyze a Nested Sampling sub-folder (i.e. something that ends in *NS* with capital letters), the analyze module will translate the output from Nested Sampling to standard chains for Monte Python, and stops. You can then run the *– info* flag on the whole folder. **This procedure is not necessary if the run was complete, but only if the Nested Sampling run was killed before completion**.

Parameters

- **files** (*list*) – list of potentially only one element, containing the files to analyze. This can be only one file, or the encompassing folder, files
- **info** (*Information instance*) – Used to store the result

analyze.**convergence**(*info*)
Compute convergence for the desired chains, using Gelman-Rubin diagnostic

Chains have been stored in the info instance of *Information*. Note that the G-R diagnostic can be computed for a single chain, albeit it will most probably give absurd results. To do so, it separates the chain into three subchains.

analyze.**compute_posterior**(*information_instances*)
computes the marginalized posterior distributions, and optionnally plots them

Parameters **information_instances** (*list*) – list of information objects, initialised on the given folders, or list of file, in input. For each of these instance, plot the 1d and 2d posterior distribution, depending on the flags stored in the instances, comming from command line arguments or read from a file.

analyze.**ctr_level**(*histogram2d*, *lvl*, *infinite=False*)
Extract the contours for the 2d plots (Karim Benabed)

analyze.**minimum_credible_intervals**(*info*)
Extract minimum credible intervals (method from Jan Haman) FIXME

analyze.**write_h**(*info_file*, *indices*, *name*, *string*, *quantity*, *modifiers=None*)
Write one horizontal line of output

analyze.**cubic_interpolation**(*info*, *hist*, *bincenters*)
Small routine to accomodate the absence of the interpolate module

analyze.**write_histogram**(*hist_file_name*, *x_centers*, *hist*)
Store the posterior distribution to a file

analyze.**read_histogram**(*histogram_path*)
Recover a stored 1d posterior

analyze.**write_histogram_2d**(*hist_file_name*, *x_centers*, *y_centers*, *extent*, *hist*)
Store the histogram information to a file, to plot it later

analyze.**read_histogram_2d**(*histogram_path*)
Read the histogram information that was stored in a file.

To use it, call something like this:

```
x_centers, y_centers, extent, hist = read_histogram_2d_from_file(path)
fig, ax = plt.subplots()
ax.contourf(
    y_centers, x_centers, hist, extent=extent,
    levels=ctr_level(hist, [0.68, 0.95]),
    zorder=5, cma=plt.cm.autumn_r)
plt.show()
```

analyze.**clean_conversion**(*module_name*, *tag*, *folder*)
Execute the methods "convert" from the different sampling algorithms

Returns True if something was made, False otherwise

analyze.**separate_files**(*files*)
Separate the input files in folder

Given all input arguments to the command line files entry, separate them in a list of lists, grouping them by folders. The number of identified folders will determine the number of information instances to create

analyze.**recover_folder_and_files**(*files*)
Distinguish the cases when analyze is called with files or folder

Note that this takes place chronologically after the function *separate_files*

analyze.**extract_array**(*line*)

Return the array on the RHS of the line

```
>>> extract_array("toto = ['one', 'two']
```

") ['one', 'two'] >>> extract_array('toto = ["one", 0.2]

') ['one', 0.2]

analyze.**extract_dict**(*line*)
Return the key and value of the dictionary element contained in line

```
>>> extract_dict("something['toto'] = [0, 1, 2, -2, 'cosmo']")
'toto', [0, 1, 2, -2, 'cosmo']
```

analyze.**extract_parameter_names**(*info*)
Reading the log.param, store in the Information instance the names

analyze.**find_maximum_of_likelihood**(*info*)
Finding the global maximum of likelihood

min_minus_lkl will be appended with all the maximum likelihoods of files, then will be replaced by its own maximum. This way, the global maximum likelihood will be used as a reference, and not each chain's maximum.

analyze.**remove_bad_points**(*info*)
Create an array with all the points from the chains, after removing non-markovian, burn-in and fixed fraction

analyze.**compute_mean**(*mean*, *spam*, *total*)

analyze.**compute_variance**(*var*, *mean*, *spam*, *total*)

analyze.**compute_covariance_matrix**(*info*)

analyze.**adjust_ticks**(*param*, *information_instances*)

analyze.**store_contour_coordinates**(*info*, *name1*, *name2*, *contours*)
docstring

analyze.**iscomment**(*s*)
Define what we call a comment in MontePython chain files

class analyze.**Information**(*command_line*, *other=None*)
Bases: object

Hold all information for analyzing runs

The following initialization creates the three tables that can be customized in an extra plot_file (see *parser_mp*).

Parameters **command_line** (*Namespace*) – it contains the initialised command line arguments

has_interpolate_module = False

cm = [(0.0, 0.0, 0.0, 1.0), (0.30235, 0.15039, 0.74804, 1.0), (0.99843, 0.25392, 0.14765, 1.0), (0.9, 0.75353, 0.10941, 1.0)]

cmaps = [<Mock id='139873176374928'>, <Mock id='139873176375120'>, <Mock id='139873176465488'>, <Mock id='1

alphas = [1.0, 0.8, 0.6, 0.4]

to_change = None
Dictionary whose keys are the old parameter names, and values are the new ones. For instance `{'beta_plus_lambda':'beta+lambda'}`

to_plot = None
Array of names of parameters to plot. If left empty, all will be plotted.

> **Warning:** If you changed a parameter name with *to_change*, you need to give the new name to this array

new_scales = None
Dictionary that redefines some scales. The keys will be the parameter name, and the value its scale.

remap_parameters(*spam*)
Perform substitutions of parameters for analyzing

> **Note:** for arbitrary combinations of parameters, the prior will not necessarily be flat.

define_ticks()

write_information_files()

write_h_info()

write_v_info()
Write vertical info file

write_tex()
Write a tex table containing the main results

7.12 Io module

Input-Output handling

Handles all the input/output of the code (at least most of it). If something is printed that does not satisfy you (number of decimals, for instance, in the output files), you only have to find the called function and change a number.

Whenever the arguments of the functions are `command_line` or `data`, no mention of them will be done - as it is now clear. On the contrary, if there are more arguments, they will be detailled.

This module also defines a new class *`File`*, that extends `file`, which provides a tail function. It is used in *`sampler.read_args_from_chain()`*.

Finally, the way the error messages are displayed is set there, along with ascii-art for the exclamation mark sign.

`io_mp.`**`log_parameters`**(*data*, *command_line*)
Write the first lines of the log.param

Writes the beginning of log.param, starting with the header with the cosmological code version and potential git hash and branch name, and then recopies entirely the input parameter file.

`io_mp.`**`log_likelihood_parameters`**(*likelihood*, *command_line*)
Write down the interpreted .data file of the input likelihood to log.param

> **Warning:** Since version 2.0.2, the lines are not copied verbatim, they are first interpreted, then copied. This allows for overriding of parameters from the input.param file.

`io_mp.`**`log_cosmo_arguments`**(*data*, *command_line*)
Write down the *cosmo_arguments* used to log.param

Third function called when writing log.param. It is understood here that all the other parameters for the cosmological modules are set to their default value directly in the program.

It is written as an update for the dictionary cosmo_arguments (i.e. as `dict.update()` and not as `dict =`) in order not to erase previously initialized data.

`io_mp.`**`log_default_configuration`**(*data*, *command_line*)
Log the .conf file to log.param

Fourth and last function called when writing log.param. Only useful if you have several versions of your cosmological code installed in different locations, or different versions of Clik. But, as you never know what might go wrong, it is logged everytime !

TODO: should the root be still logged? (@packaging)

`io_mp.`**`print_parameters`**(*out*, *data*)
Will print the parameter names. In the code, `out` is simply the standard output, as this information will not be printed on the output file.

Indeed, you will be able to recover these information from the log.param.

> **Warning:** Please pay attention to the fact that, once launched, the order of the parameters in log.param is crucial, as is it the only place where it is stored.

`io_mp.`**`print_vector`**(*out*, *N*, *loglkl*, *data*)
Print the last accepted values to `out`

Parameters

- **out** (*list*) – Array containing both standard output and the output file.

  This way, if you run in interactive mode, you will be able to monitor the progress of the chain.

- **N** (*int*) – Multiplicity of the point, *i.e.* number of times the code stayed at this particular place.

- **loglkl** (*float*) – Value of the (- log likelihood) at this point

- **.. note** (*:*) –

  It is the *last_accepted* point that is printed, and **not** the *current* one (obviously, as one does not know yet the multiplicity of the current one !)

io_mp.**refresh_file**(*data*)
Closes and reopen the output file to write any buffered quantities

io_mp.**create_output_files**(*command_line*, *data*)
Automatically create a new name for the chain.

This routine takes care of organising the folder for you. It will automatically generate names for the new chains according to the date, number of points chosen.

Warning: The way these names are generated (with the proper number of _, __, -, and their placement) is exploited in the rest of the code in various places. Please keep that in mind if ever you are in the mood of changing things here.

io_mp.**get_tex_name**(*name*, *number=1*)
Simplistic tex name transformer.

Essentially tries to add a backslash in front of known possible greek letters, and insert curly brackets { } around statement following an _ or a ^. It will also try to include the scale into the name in a nice way.

Note: This might easily fail on simple names, like *beta_plus_lambda*. In this case, please use an extra plot file with the command line option `-extra plot_file`, or come up with a better function !

Note: This function returns immediatly with the unmodified name if it already contains the LaTeX symbol for math, $.

Parameters **name** (*str*) – Input name

Keyword Arguments **number** (*float*) – Scale

io_mp.**write_covariance_matrix**(*covariance_matrix*, *names*, *path*)
Store the covariance matrix to a file

io_mp.**write_bestfit_file**(*bestfit*, *names*, *path*)
Store the bestfit parameters to a file

io_mp.**pretty_print**(*string*, *status*, *return_string=False*)
Return the string formatted according to its status

The input is a potentially long message, describing the problem. According to the severity of its status (so far, 'error' will exit the program, whereas 'warning' and 'info' will go through anyway).

Standard length has been defined globally, as well as the ascii-art dictionary of arrays START_LINE.

io_mp.**safe_exec**(*string*)
Attempt at executing a string from file in a secure way

class io_mp.**File**
Bases: file

New class of file, to provide an equivalent of the tail command (on linux).

It will be used when starting from an existing chain, and avoids circling through an immense file.

tail(*lines_2find=1*)
Imitates the classic tail command

exception io_mp.**LockError**
Bases: exceptions.Exception

Warning: in the process of being tested

LOCK_FAILED = 1

io_mp.**lock**(*file*, *flags*)
Lock a given file to prevent other instances of the code to write to the same file.

Warning: in the process of being tested

io_mp.**unlock**(*file*)
Unlock a previously locked file.

Warning: in the process of being tested

io_mp.**warning_message**(*message*, **args*)
Custom implementation of *showwarning* from warnings

exception io_mp.**MyError**(*message*)
Bases: exceptions.Exception

Base class defining the general presentation of error messages

Reformat the name of the class for easier reading

__str__()
Define the behaviour under the print statement

exception io_mp.**CosmologicalModuleError**(*message*)
Bases: *io_mp.MyError*

For all problems linked to the cosmological module

Reformat the name of the class for easier reading

exception io_mp.**ConfigurationError**(*message*)
Bases: *io_mp.MyError*

Missing files, libraries, etc...

Reformat the name of the class for easier reading

exception io_mp.**MissingLibraryError**(*message*)
Bases: *io_mp.MyError*

Missing Cosmo module, Planck, ...

Reformat the name of the class for easier reading

exception io_mp.**LikelihoodError**(*message*)
Bases: *io_mp.MyError*

Problems when computing likelihood, missing nuisance, etc...

Reformat the name of the class for easier reading

exception io_mp.**FiducialModelWritten**(*message*)
Bases: *io_mp.MyError*

Used to exit the code in case of writing a fiducial file

Reformat the name of the class for easier reading

exception io_mp.**AnalyzeError**(*message*)
Bases: *io_mp.MyError*

Used when encountering a fatal mistake in analyzing chains

Reformat the name of the class for easier reading

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

a

d

i

l

m

p

r

s

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W