Monte Python Documentation

Release v1.0

Benjamin Audren & Julien Lesgourgues

October 29, 2012

Contents

1 Prerequisites

1.1 Python

First of all, you need a clean installation of **python** (version 2.x, $x \geq 7$, but < 3.0), 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 honour to have the root password on your machine, an `apt-get install python-numpy`, `python-scipy` and `cython` will do the trick.).

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.

1.2 CLASS

Next in line, you must compile the python wrapper of `CLASS`. Download the latest version ($\geq$ 1.5.0) at `http://class-code.net`, and follow the basic instruction. Instead of `make class`, type `make`. This will also create 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 successfull, this should work within any directory. If you get no error message from this line, you know everything is fine.

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 section 2: Installation), then it will use this one.

2 Installation

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 edit two files (the first, once for every new distribution of `Monte Python`, and the second, once and for all). The first to edit is `code/MontePython.py`. Its first line reads:

```
#!/usr/bin/python
```

You should eventually replace this path with the one of your python 2.7 executable, if different. This modification is not crucial, it simply allows to run the code by simply typing `code/Montepython.py`. If, instead, you run it through python (*i.e.*: `python code/MontePython.py`), then this line will be disregarded.

The second file to change, and this one is crucial, is `default.conf`, in the root directory of the code. This file 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 code/MontePython.py --help` (or just `code/MontePython.py --help`). This will provide you with a short description of the available command line arguments, explained in section 4.

2.2 WMAP 7 likelihood

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`.

3 Getting started

3.1 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 `likelihood/` directory. Likelihoods will be explained in section 6

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 for unbounded prior edge),
- **maximum value** (set to -1 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 `1.e-9` if you are dealing with `A_s` or `0.01` if you are dealing with `omega_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, located in `code/data.py`, 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 4). 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.

3.2 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, etc... 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 first check whether the two parameter files are in agreement, and if not, will stop and say so. In that way, you are sure not to put together some chains obtained under different physical assumptions.

In the folder `montepyhton`, 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]$ code/MontePython.py -conf my-machine.conf -p example.param -o test
```

3.3 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 stores 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 (Raferty & Lewis test for each chain paramater), 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. 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 section 4 a list of commands to customize the plots. Besides, you may wish to change font sizes: for the moment this has to be done by editing the very last lines of `code/analyse.py` and writing your own values for the `fontsize` and `ticksize` variables.

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.

3.4 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 `yyy` 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.

4 Command line arguments

The command `code/MontePython.py --help` lists all command line arguments understood by `Monte Python`. They fall in two categories:

- arguments useful for running chains:

`-N steps`	number of steps in the chain (when running on a cluster, your run might be stopped before reaching this number)
`-o output_folder`	for example `-o chains/myexperiments/mymodel`
`-p input_param_file`	for example `-p input/exoticmodel.param`
`-c input_cov_matrix`	name of a covariance matrix (created when analyzing a previous run, and used in input to initialise the proposal density). List of parameters in the input covariance matrix and in the run don't need to coincide.
`-j jumping_method`	can take two values, `global` (default) and `sequential`. 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). With 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.
`-f jumping_factor`	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, found by Dunkely 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 configuration_file`	as explained before, this command is crucial to tell `Monte Python` about the location of `CLASS`.
`-chain_number chain_number`	chains are named automatically `yyyy-mm-dd_N__i.txt` where `yyy` 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: so running `Monte Python` several times with exactly the same command will automatically lead to different chain names. 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 chain_name`	restart from the last point of a previous chain, to avoid a new burn-in stage. At the beginning of the run, the previous chain will be deleted, and its content transfered to the beginning of the new chain.

- arguments useful for analyzing chains and plotting:

`-info [file [file ...]]`	chains to analyze; give the folder name to analyze all chains in this folder, or individual chains, for example `-info chains/myrun/` or `-info chains/myrun/2012-10-26* chains/myrun/2012-10-27*`
`-bins number_of_bins`	number of bins in the histograms used to derive posterior probabilities and credible intervals, default is 20; reduce this number for smoother plots at the expense of masking details
`-no_mean`	by default, when plotting marginalised 1D posteriors, the code also shows the mean likelihood per bin with dashed lines; this option switches off the dashed lines
`-comp comparison_folder`	pass the name of another folder (or another set of chains, same syntax as `-info`) if you want to compare 1D posteriors on the same plot. The lists of parameters in the two folders to compare do not need to coincide. Limited so far to two folders to compare in total.
`-extra plot_file`	name of an optional file containing a few lines for customizing the plots, `info.to_change={'oldname1':'newname1','oldname2':'newname2',...}` `info.to_plot=['name1','name2','newname3',...]` `infot.new_scales={'name1':number1,'name2':number2,...}`
`-noplot`	analyze chains without drawing plots
`-all`	by default, produces only full 1D plot and triangle plots, with this option all individual 1D plot and 2D contour plot are stored

5 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
`$ mkdir plot_files`	to store files used to customize plots

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

```
$ code/Montepython.py -conf my-conf.conf -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:

```
$ code/Montepython.py -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:

[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.

```
$ code/Moncode/Montepython.py -conf my-conf.conf -p input/lcdm.param
   -o chains/planck/lcdm -c covmat/fake_planck_lcdm.covmat -N 5
```

I now wish to launch longer runs on my cluster or 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 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. 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 WMAP likelihood, I should not forget to add in the script (before calling `Monte Python`) the line

```
source /path/to/my/wrapper_wmap/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
```

Advanced cluster users may even find ways to launch several serial runs appearing as a single job on the cluster with the `mpirun` command.

When the runs have stopped, I can analyse them with

```
$ code/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 `code/Montepyhton.py`, the option

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

and I launch a new run with several chains. If I launch this second 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.

```
code/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 previous chains using the `-r` command line option. The name of previous chains can be written after `-r` manually or through a script.

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.

6 Existing and new likelihoods

6.1 One likelihood = 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 `code/` 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 `likelihood/` 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 `likelihood` directory, and they are ready to be used if mentioned in the input file.

This can work because 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. `likelihoods/hst/`,
- for naming the input data file describing the characteristics of the experiment, `likelihoods/hst/hst.data` (this file can point to raw data files located in the `data` directory)
- for naming the python file containing the likelihood code, `likelihoods/hst/hst.py`
- for naming the class declared in `likelihoods/hst/hst.py` and used also in `likelihoods/hst/hst.data`

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 $h = 0.72 \pm 0.02$. If you want to change these numbers, you can simply edit `likelihoods/hst/hst.data` and change the numbers 0.72 and 0.02. 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/hst.py likelihoods/spitzer/spitzer.py
```

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

```
class spitzer(likelihood_prior):
```

Edit also `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 `arXiv:1210.219`. 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 astro-ph/0606227).

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 `likelihoods/fake_planck_bluebook/fake_planck_bluebook.data` 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 `'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 `likelihoods/xxx/xxx.py`, using input numbers and input files names specified in `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 `code/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/cbi.py likelihoods/nextcmb/nextcmb.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 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 tou 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 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.

This means 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')
```

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_associated_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
```

To know exactly what is pre-defined in the generic likelihood class, open the file `code/likelihood_class.py`. You will find the following set of functions:

- `read_from_file`, to read your `.data` file and extract properly the information
- `get_cl`, to return the properly normalized C_l's from `CLASS`, in μK^2.
- `need_cosmo_arguments`, to enforce the definition of a certain value of certain cosmo parameters (e.g. to enforce that the C_l's get computed until a certain l),
- `read_contamination_spectra`,
- `add_contamination_spectra`,
- `add_nuisance_prior`.

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.