Monte Python,
(CLASSy Coding)

Benjamin Audren

February 10, 2012

Contents

1 Installing 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, you should have a full **scipy module** (at least version 0.9.0), with the method *i*nterpolate. 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.

1.2 `CLASS`

Next in line, you must execute the python wrapper of `CLASS`. Download the latest version at `http://class-code.net`, and follow the basic instruction. Instead of the `make class`, do a `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 you get no error message from this line, you know everything is fine (be carefull not to execute python in the class/python/ folder, for it would not mean it is installed on the whole system.

1.3 Clik

Download the latest Clik public release, with mock planck data from `blabla`. Follow the instruction to install it. This step is not compulsory, as for now, Monte Python can run on its own. However, if you want to use the WMAP7 likelihood, and/or planck mock data, you need to install it.

Once it is installed, everytime you will want to use it with Monte Python, you will have to run the following command:

```
~]$ source path/to/your/clik/bin/profile.sh
```

Alternatively, you can add the previous command to your ~/.bashrc file.

2 Installation

Move the latest release of Monte Python to your code/ folder, and untar its content:

```
code]$ tar -xzvf montepython-x.y.tgz
code]$ cd montepython
```

You will have to edit two files (one for every new distribution of Monte Python, and the other one, once and for all). The first to edit is *code/MontePython.py*. Its first line reads:

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

You have to change this line to wherever your python executable is installed. This modification is not crucial, since it will only affect the program if executed. 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 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 two lines, filled with the relevant information for your machine:

```
path['class']   = 'path/to/your/class/'
path['data']    = 'path/to/your/montepython/data/'
```

If you have installed Clik, then you should also add:

```
path['clik']    = 'path/to/your/clik/examples/'
```

To run Monte Python, simply type `python code/MontePython.py`. Of course, you will need to provide more options than that, that will be explained below.

3 Summarized behaviour and philosophy of Monte Python

The principle is the following. You run Monte Python giving him, at least, a parameter file and an output folder. This will create a new folder, store the parameter file (in *folder/log.param*), and start a Markov Chain there. You can then launch new runs in this existing folder, as long as you are using the same parameter file. Otherwise, the program will simply not launch.

All the configuration files for all the likelihoods, Class parameters and mcmc parameters will be stored in this folder. This will allow you to make reference to this file, rerun anything with the exact same settings as much as you want.

After analysis, the folder will also contain three more files:

- *folder.info*: with summarized convergence properties, mean and best-fit values, and 1-sigma deviations.
- *folder.log*: with the detailed properties of all files used to create folder.info
- *folder.covmat*: contains the covariance matrix of the parameters

as well as a new subfolder: *plots/*, with the triangle and 1d plot.

3.1 Parameter Files

This file contains the following lines:

```
data.exp = ['clik_wmap_full','sn','acbar']

data.params[cosmo_name]       = [mean,min,max,1-sigma prior,scale,'cosmo']
...

data.params[nuisance_name]    = [mean,min,max,1-sigma prior,scale,'nuisance']
...

data.Class_arguments['perturbation_verbose'] = 3
data.Class_arguments['non linear']           = 'one-loop'

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. The way they work will be explained in subsection 3.4

In `data.params`, you will list all the Class and nuisance parameter you want to vary (if they differ from the default values). You must give an array with six elements in it, in this order:

- **mean value**,
- **minimum value** (set to -1 if irrelevant or unimportant),
- **maximum value** (same),
- **1-sigma prior** (only used when there are no covariance matrix as an input),
- **scale** (most of the time, it will be 1, but occasionnaly can be $1e-9$ for instance) and finally

- **role** (so far, the only valid options are 'cosmo' and 'nuisance')

The procedure for `data.Class_arguments` differs only slightly. What you pass here as argument is in fact the same value you would use in the `explanatory.ini` of your `CLASS`runs. So it is no array, but either a number or a string.

For the moment, remember that all elements you input with 'cosmo' role will be interprated by the code as arguments for the cosmological code (only `CLASS`so far). It exists a function that you can edit that will allow you not to have a one to one correspondance between these two things. It is located in `code/data.py` and called `update_Class_arguments`. A detailled exemple on how you should modify the function to suit your needs is alreay implemented with the case of Ω_Λ

The two last elements are the number of steps you want your chain to be (`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, if you are running on a cluster where your jobs might easily get killed, or when you are testing. If instead you are afraid of loosing time with writing on disk tasks, you can put it higher. Anyway, I would recommend not to go much higher than 5, for it really is not the most time consuming part of the code.

Note: you will want to overwrite the `data.N` in the command line, with the option `-N 10000`. The value by default in the parameter file is intentionnaly low (though you can change it of course). The reason is simply to prevent doing any mistake while testing the program on a cluster.

3.2 Folder = set of experiments and parameters

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 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 these 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 says so.

3.3 Input covariance matrix

From any folder containing chains with more than 20 accepted points, you can launch the analyze part of the code by invoking the command:

```
python code/MontePython.py -info chains/wmap7_sn/ -bins 15
```

This will create, among other, a `chains/wmap7_sn/wmap7_sn.covmat` file, that will contain on the first line all the names of the varying parameters of this run, and immediatly below the deduced covariance matrix.

You can then feed this as an input for a new run, with the following command line argument :

```
python code/MontePython.py -o new_folder -p new.param -c wmap7_sn/wmap7_sn.covmat
```

Here, `new.param` is supposed to have different parameters (for instance, using a varying z_{reio} instead of a fixed one). The program will then deduce the correct starting covariance matrix of the already computed

elements, only using the 1σ prior for new parameters. Do not bother to have the same ordering: the code is doing it all for you.

3.4 Likelihood = a .py and a .data file

To finish this summarized presentation, let us look on the likelihood files. They are defined in a very specific way, that you will have to respect whenever you want to implement new ones.

Every likelihood is defined in the `path/to/MontePython/likelihoods/name_of_likelihood/` folder. This folder contains at least two files, `name_of_likelihood.py` and `name_of_likelihood.data`

The `.py` file defines the class (in python sense) of your likelihood. Fortunately for you, parent classes have already been defined in the `code/likelihood_class.py`. For instance, for all the newdat type of likelihoods (acbar, boomerang, bicep, etc.), their `.py` file will be painfully simple:

```
from likelihood_class import likelihood_newdat

class acbar(likelihood_newdat):
  pass
```

Yes, that is all. Now you have to fill in the `.data` file. Continuing on the example with acbar, which requires a nuisance parameter (either fixed or varying), let us take a look at the entirity of the file `acbar.data`:

```
acbar.data_directory = data.path['data']
acbar.file           = 'acbar2007_v3_corr.newdat'
acbar.use_nuisance   = ['A_SZ']

acbar.A_SZ_file      = 'WMAP_SZ_VBand.dat'
acbar.A_SZ_scale     = 0.28
```

The only particular thing is there: the `.use_nuisance` option. You have to pass here the entire list of the nuisance parameters the likelihood is expecting before running. After, for each nuisance parameter, you need to specify the band file, and the scale. Here, you could also have a line that force Class to have certain parameters (a certain l_max for instance). To enforce this behaviour, just add:

```
acbar.need_Class_arguments(data,{'lensing':'yes', 'output':'tCl lCl pCl'})
```

Now in your parameter file, simply input the line `data.exp=['acbar']`, and the code will compute the new likelihood. You have a new experiment, already in the newdat format ? Create the folder, the two files, put your data in `data.path['data']` and the name in the list of experiments and here you go. It really is this simple.

3.5 General guidelines to keep in mind

Here follows some rules about being sure everything work fine with MontePython.

- Be sure to launch only one experiment by folder. If, after some month, you want to relaunch a run from this, simply call the program without any parameter file, but just the existing output folder. Everything was stored for you.
- When launching a new run by extending a previous one (adding a free $N_{\rm eff}$ to ΛCDM for instance), it is often better not to take any starting covariance matrix than one that might not be adapted.

- Also, when launching a new run where you are not sure of your priors, use the `-j sequential` flag. It creates much more correlated chains, but this helps get started, produce a meaningful covariance matrix, and start again from a better point.

4 Creating (completely new) likelihoods

We already saw that adding likelihoods that already come into a certain format (from Clik, or with the newdat format) is a matter of three minutes. However, if one ones to code a new likelihood, one has to understand a bit the way classes work in this code.

The file `code/likelihood_class.py` contains all the basic definitions of likelihoos already implemented. The basic one, `likelihood` will be useful to you if you want to create a new one from scratch. Just let your new class inherit from likelihood, and build over that. The following functions are already coded for you:

- `read_from_file`, it will only read your .data file and extract properly the information
- `get_cl`, returns the properly normalized C_l from Class, in μK^2.
- `need_Class_arguments`, enforce the definition at a certain value of certain Class parameters. Typically, this will enforce that the C_l gets computed until a certain l.
- `read_contamination_spectra`, TODO julien ?
- `add_contamination_spectra`,
- `add_nuisance_prior`,