A quick guide for CLASS and MontePython

Diogo Castelão

2022

CLASS

You can find informations about the code, including lectures and exercises, in the offical webpage: https://lesgourg.github.io/class_public/class.html.

Installing CLASS:

CLASS comes with a python wrapper, required to work with MontePython. Make sure you have the following libraries installed: numpy, scipy, cython (also matplotlib).

You can clone the code from GitHub, or download a .tar.gz file. (from https://github.com/lesgourg/class_public):

```
> git clone https://github.com/lesgourg/class_public.git class
> cd class/
> make clean
>make -j
```

If everything went well, you should be able to run the code without any error:

```
>./class explanatory.ini
```

Check if python wrapper installation also worked:

```
>python
>>> from classy import Class
```

if python does not complain, you're fine.

Note: for most of Linux distro, CLASS will install easily. For MacOS, you may need to modify the Makefile. One main modifications is the clang incompatibility with OpenMP. For that, comment line 41: #OMPFLAG   = -fopenmp and uncomment line 43: OMPFLAG   = -openmp.
For other problems, find more information in:
https://github.com/lesgourg/class_public/wiki/Installation

Useful informations about CLASS:

- In the CLASS directory you have:

- main /	# main CLASS function: basically calls 10 modules
- source/	# the 10 modules of CLASS : ALL PHYSICS IN HERE
- include/	# header file (*.h) for each module containing all declarations.
- external/	# External codes: HyRec, BBN interpolation table …
- python/	# Python wrapper
- explanatory.ini	# input file with all the descriptions you need. (USEFUL)
- notebook/	# Examples of jupyter notebooks (USEFUL for running and plotting)

- You have two tools for plotting (recommend Jupyter notebook):

- CPU.py (check for info: > python CPU.py -h) # Python plotting script.
When you run for example:
> python CPU.py output/something_background.dat -y rho –scale loglog
you will generate a file called output/something_background.py which you can check.

- plot_CLASS_output.m (check for info: help plot_CLASS_output.m) #Matlab script

- If you want to modify CLASS you can easily trace how the physics of a specific DE or DM specie is written in the code by following a similar component already implemented in the CLASS.
Example: for a DE scalar field type search for, "has_scf", in each module of CLASS. Make a copy of each definition and modify it as you desire (also modify the headers where you define new variables).

-Check in the end of the document for the Appendix. A more complete example of modifying CLASS for a new DE equation of state is given.

MontePython

You can find lectures and exercises in the webpage of CLASS (https://lesgourg.github.io/class_public/class.html), and useful information and documentation in the MontePython repository: https://github.com/brinckmann/montepython_public.

Installing MontePython:

You can clone the code from GitHub, or download a .tar.gz file. (from https://github.com/brinckmann/montepython_public):

```
> git clone https://github.com/brinckmann/montepython_public.git
```

You will need to modify only two files to your local configuration:

- In the montepython/MontePython.py file, you have in the first line:
#!/usr/bin/python - change it to your preferred python distribution.

- Make a copy of the default.conf.template file locate in the MontePython folder and rename it to default.conf. There you should define the path to your CLASS code:

```
    path['cosmo']        = '/something/class_folder/'
```

and if you want to use the planck likelihood, the path where you have it installed:

```
    path['clik']       = 'something/code/plc_3.0/plc-3.01/'
```

That's it!

Now you can call:

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

to get a list of all commands.

Useful informations about MontePython:

- For the run command, there are two important commands: you should define an output folder (-o) and a parameter file (-p). You can find in the input/ folder, several examples of parameter files.

A typical run would be:

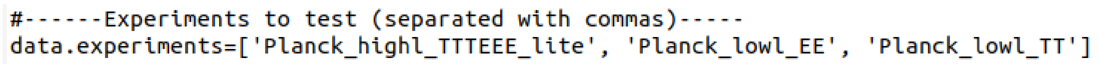
```
> python montepython/MontePython.py run -o chains/example -p example.param -N 100
```

with -N being the number of steps.

To analyse the chain, type:

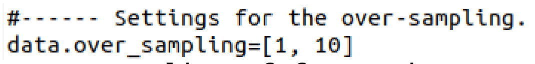
```
python montepython/MontePython.py info chain/example/
```

- A quick guide through a parameter file:



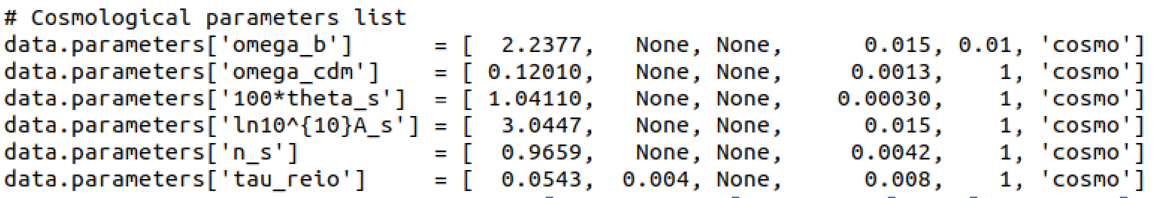
```
#------Experiments to test (separated with commas)-----
data.experiments=['Planck_highl_TTTEEE_lite', 'Planck_lowl_EE', 'Planck_lowl_TT']
```

List of experiments: must match the names in the /montepython/likelihoods folder.



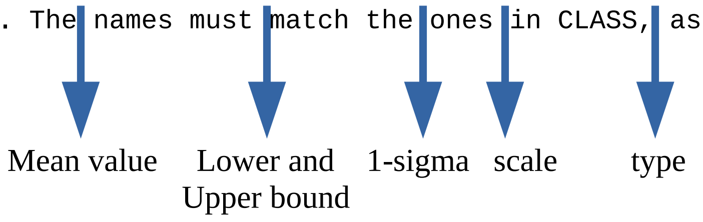
```
#------ Settings for the over-sampling.
data.over_sampling=[1, 10]
```

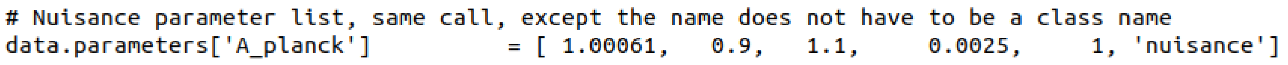
Over-sampling of fast nuisance parameters. First digit: cosmological parameters. Second digit: Nuisance parameters. In this case, nuisance parameters are sampled 10x more in comparison to cosmological parameters.



```
# Cosmological parameters list
data.parameters['omega_b']      = [  2.2377,   None, None,     0.015, 0.01, 'cosmo']
data.parameters['omega_cdm']    = [ 0.12010,   None, None,    0.0013,    1, 'cosmo']
data.parameters['100*theta_s']  = [ 1.04110,   None, None,   0.00030,    1, 'cosmo']
data.parameters['ln10^{10}A_s'] = [  3.0447,   None, None,     0.015,    1, 'cosmo']
data.parameters['n_s']          = [  0.9659,   None, None,    0.0042,    1, 'cosmo']
data.parameters['tau_reio']     = [  0.0543,  0.004, None,     0.008,    1, 'cosmo']
```

Cosmological parameters. The names must match the ones in CLASS, as in explanatory.ini.





```
# Nuisance parameter list, same call, except the name does not have to be a class name
data.parameters['A_planck']          = [ 1.00061,    0.9,    1.1,     0.0025,     1, 'nuisance']
```

Nuisance parameters, related with the experiment in use. Only used in MontePython and not related with CLASS. Use of over-sampling useful for faster convergence, since nuisance parameters often faster to vary.

```
# Derived parameters
data.parameters['z_reio']          = [1, None, None, 0,     1,   'derived']
data.parameters['Omega_Lambda']    = [1, None, None, 0,     1,   'derived']
data.parameters['sigma8']          = [0, None, None, 0,     1,   'derived']
```

Derived parameters are computed from other parameters and do not affect the run.

```
# Other cosmo parameters (fixed parameters, precision parameters, etc.)
data.cosmo_arguments['sBBN file'] = data.path['cosmo']+'/bbn/sBBN.dat'
data.cosmo_arguments['k_pivot'] = 0.05
```

Arguments passed to CLASS. For example, fixed parameters or precision parameters.

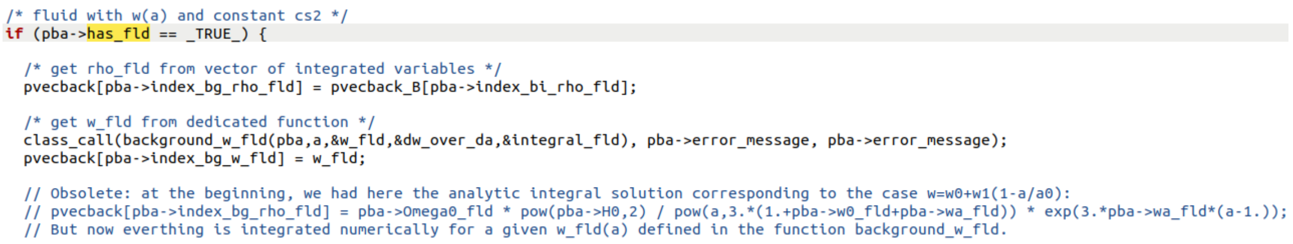
- To install the Planck likelihood check MontePython repository or check https://cosmologist.info/cosmomc/readme_planck.html (made for COSMOMC but commands are the same).

- To use the JLA likelihood you need to download the dataset. In the MontePython folder go to /data/JLA/ and open the readme file, there you find the location to download the dataset. You should put the content of /jla_likelihood_v4/data/ in the folder where the readme file is.

- To install and use other sampling method besides Metropolis-Hastings check in MontePython documentations. Example, for Nested Sampling follow: https://monte-python.readthedocs.io/en/latest/nested.html

Appendix

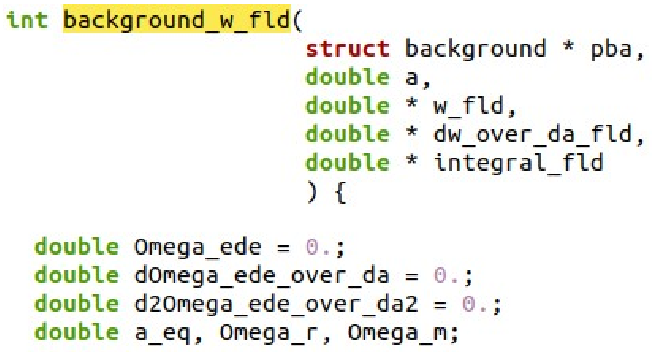
CLASS modification for a new fluid equation of state: Looking at the background.c we can find (searching for "has_fld"):

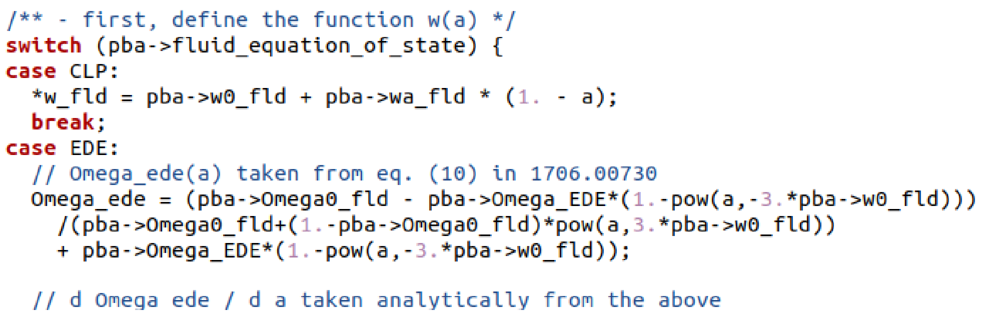


```
/* fluid with w(a) and constant cs2 */
if (pba->has_fld == _TRUE_) {

  /* get rho_fld from vector of integrated variables */
  pvecback[pba->index_bg_rho_fld] = pvecback_B[pba->index_bi_rho_fld];

  /* get w_fld from dedicated function */
  class_call(background_w_fld(pba,a,&w_fld,&dw_over_da,&integral_fld), pba->error_message, pba->error_message);
  pvecback[pba->index_bg_w_fld] = w_fld;

  // Obsolete: at the beginning, we had here the analytic integral solution corresponding to the case w=w0+w1(1-a/a0):
  // pvecback[pba->index_bg_rho_fld] = pba->Omega0_fld * pow(pba->H0,2) / pow(a,3.*(1.+pba->w0_fld+pba->wa_fld)) * exp(3.*pba->wa_fld*(a-1.));
  // But now everthing is integrated numerically for a given w_fld(a) defined in the function background_w_fld.
```

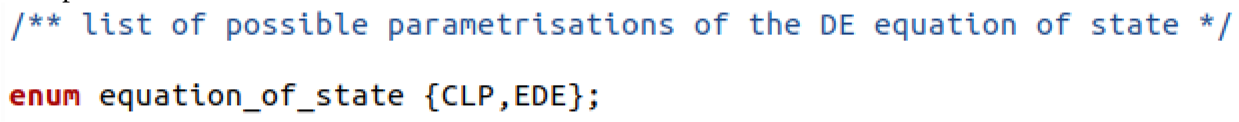
In this part of the code we can see that pvecback[pba→index_bg_rho_fld] (where the energy density of the fluid is stored) is integrated numerically.

Also, we can see that the equation of state w_fld comes from the function background_w_fld(…). In CLASS the DE fluid implemented allows diferent parametrisations of w(a). In the current version the CLP and the EDE parametrisation are included in the code. To choose which parametrisation we want to use, we should define in our input file the parameter:
fluid_equation_of_state: CLP or EDE.
We need to modify this function in the background module if we have a DE fluid with a particular equation of state that needs to be implemented:

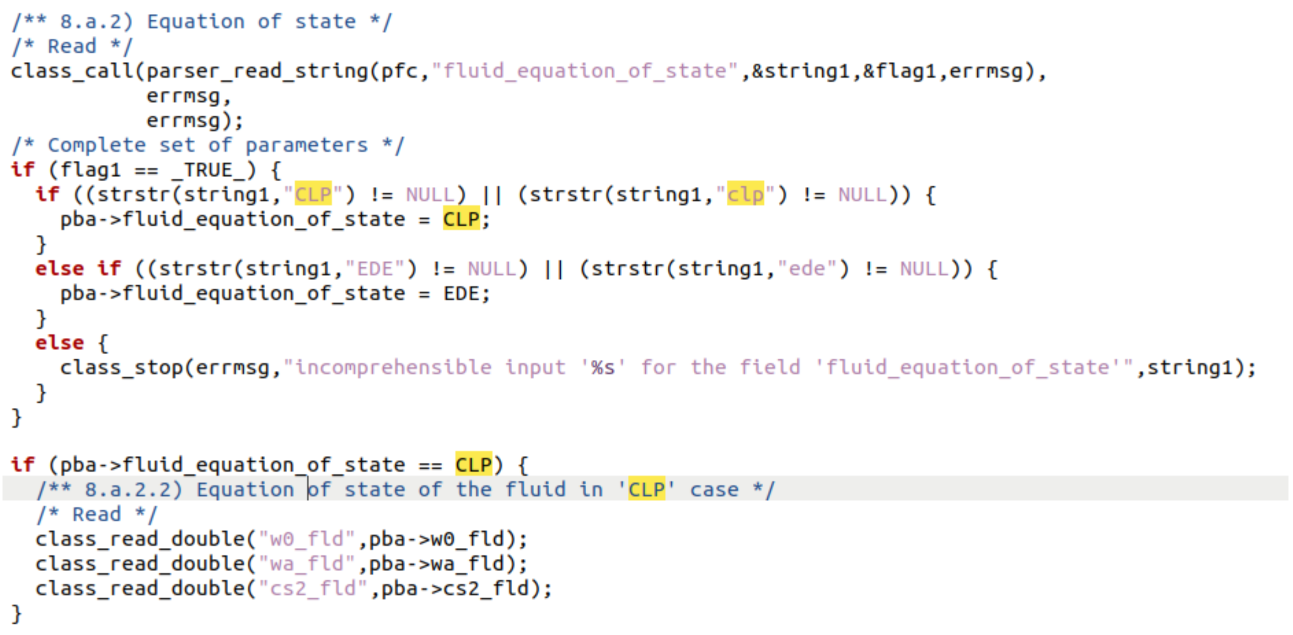

```
int background_w_fld(
                     struct background * pba,
                     double a,
                     double * w_fld,
                     double * dw_over_da_fld,
                     double * integral_fld
                     ) {

  double Omega_ede = 0.;
  double dOmega_ede_over_da = 0.;
  double d2Omega_ede_over_da2 = 0.;
  double a_eq, Omega_r, Omega_m;
```
Here we can find the function used to calculate the equation of state for the DE fluid.


```
  /** - first, define the function w(a) */
  switch (pba->fluid_equation_of_state) {
  case CLP:
    *w_fld = pba->w0_fld + pba->wa_fld * (1. - a);
    break;
  case EDE:
    // Omega_ede(a) taken from eq. (10) in 1706.00730
    Omega_ede = (pba->Omega0_fld - pba->Omega_EDE*(1.-pow(a,-3.*pba->w0_fld)))
      /(pba->Omega0_fld+(1.-pba->Omega0_fld)*pow(a,3.*pba->w0_fld))
      + pba->Omega_EDE*(1.-pow(a,-3.*pba->w0_fld));

    // d Omega ede / d a taken analytically from the above
```
Here you can set the new parametrisation that you wish to implement in CLASS creating a new case. In the rest of the function background_w_fld you will also have to define your new parametrisation case two more times: one where it is ask to write dw/da and other to write the analytical solution for the integral \int_{a}^{a0} da 3(1+w_{fld})/a. You may calculate the derivative of w(a) and the integral for your specific parametrisation by hand or using another code (e.g. Mathematica).

We then need add our new variables of the model in the background.h file. This includes the name for the new parametrisation of the equation of state and the new parameters of the model. For example:


```
/** list of possible parametrisations of the DE equation of state */

enum equation_of_state {CLP,EDE};
```
Here, you can set the name of the new parametrisation that you want to implement. Continue in the header file and check for example where wa_fld is defined and add your new parameter wb_fld.

You will also need to modify the input.c file. Where you will tell CLASS which parameters he should look for in the *.ini file. Search in the file for “CLP” and add your new case following what has been done for this parametrisation (remember that you use wb_fld instead of wa_fld in your new parametrisation). Example of parts of the input.c code to be modified:



```
/** 8.a.2) Equation of state */
/* Read */
class_call(parser_read_string(pfc,"fluid_equation_of_state",&string1,&flag1,errmsg),
           errmsg,
           errmsg);
/* Complete set of parameters */
if (flag1 == _TRUE_) {
  if ((strstr(string1,"CLP") != NULL) || (strstr(string1,"clp") != NULL)) {
    pba->fluid_equation_of_state = CLP;
  }
  else if ((strstr(string1,"EDE") != NULL) || (strstr(string1,"ede") != NULL)) {
    pba->fluid_equation_of_state = EDE;
  }
  else {
    class_stop(errmsg,"incomprehensible input '%s' for the field 'fluid_equation_of_state'",string1);
  }
}

if (pba->fluid_equation_of_state == CLP) {
  /** 8.a.2.2) Equation of state of the fluid in 'CLP' case */
  /* Read */
  class_read_double("w0_fld",pba->w0_fld);
  class_read_double("wa_fld",pba->wa_fld);
  class_read_double("cs2_fld",pba->cs2_fld);
}
```

Not required but recommended: In the input.c is where you also define the default values for each parameter used in CLASS. If you wish to set a default value for wb_fld, search for wa_fld in the end of the code and set you default value for wb_fld.

With these modifications you should be able to run your new model with a new w(a) parametrisation.

Enjoy!