CLASS

the Cosmological Linear Anisotropy Solving System[1]

Julien Lesgourgues
TTK, RWTH Aachen University

U. di Padova, 15-16.11.2021

[1] code developed by Julien Lesgourgues & Thomas Tram plus many others

Coding with `class`

1. installation, running, documentation
2. python wrapper, using scripts and notebooks
3. dynamical indexing rules
4. input management
5. error management rules
6. adding features
7. adding parameters in the wrapper
8. interface with samplers

Installation

Installation should be straightforward on Linux, and a bit more tricky but still easy on Mac. We suggest to not even try on other OSs.
We recommend cloning the code from GitHub. The old-fashioned way, i.e. downloading a .tar.gz, also works.
In the ideal case you would just need to type in your terminal

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

and you would be done. To check whether the C code is correctly installed, you can type

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

which should run the code and write some output on the terminal. To check whether the python wrapper installation also worked, try

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

and just check that python does not complain. If any of these steps does not work, please look at the detailed installation instructions at
https://github.com/lesgourg/class_public/wiki/Installation

Once the code is installed, where do I find documentation?

1. Basic information and links:
    - in the historical CLASS webpage `http://class-code.net`
    - in the online documentation page (from the previous page, or from `https://github.com/lesgourg/class_public/wiki`, clik on the link `online html documentation`), in the first two subsections:
        - CLASS: Cosmic Linear Anisotropy Solving System
        - Where to find information and documentation on CLASS?
        - CLASS overview (architecture, input/output, general principles)
2. More advanced:
    - several detailed courses at different levels on my course webpage `https://lesgourg.github.io/courses.html`, especially the New York CCA course (slides+videos); this lecture will be added there too.
    - full automatically-generated documentation (including dependence trees) on the `online html documentation`, in the last sections: Data Structures, Files.

class/ directory

In your class directory (e.g. class_public-3.0.2/), you should see:

```
cpp/      # C++ wrapper of CLASS
doc/      # the automatic documentation in PDF
external/# embedded codes: HyRec, BBN interpolation table,
          # input for CMB distortions, ...
include/ # header files (*.h) containing declarations
main/     # main CLASS function: short, just calls 10 modules
notebooks/ # examples of useful jupyter notebooks
output/   # directory for output files
python/   # python wrapper of CLASS
scripts/ # same as notebooks in python script format
source/   # the 10 modules of CLASS: ALL THE PHYSICS
test/     # other main functions for testing part of the code
tools/    # auxiliary pieces of code (numerical methods):
          # ALL THE MATH (no external C library)
explanatory.ini   # reference input file
+ plenty of other input files (e.g. with Planck best fit)
+ plenty of precision setting files
CPU.py    # small python plotting script
plot_CLASS_output.m # same for MatLab
```

Running in terminal with input file (old fashioned)

Try for instance:

```
> ./class default.ini
```

It gives some output:

```
Reading input parameters
 -> matched budget equations by adjusting Omega_Lambda =
     0.690026
Running CLASS version v3.0.1
# selected lines from the output:
  -> age = 13.797336 Gyr
  -> radiation/matter equality at z = 3406.907947
  -> recombination at z = 1088.798382
  -> reionization  at z = 7.681290
  -> sigma8=0.811718 (computed till k = 8.44246 h/Mpc)
(...)
Writing output files in output/default04_...
```

Running in terminal with input file (old fashioned)

Try for instance:

```
> ./class default.ini
```

It gives some output:

```
Reading input parameters
 -> matched budget equations by adjusting Omega_Lambda =
     0.690026
Running CLASS version v3.0.1
# selected lines from the output:
  -> age = 13.797336 Gyr
  -> radiation/matter equality at z = 3406.907947
  -> recombination at z = 1088.798382
  -> reionization  at z = 7.681290
  -> sigma8=0.811718 (computed till k = 8.44246 h/Mpc)
(...)
Writing output files in output/default04_...
```

- Chatty behavior comes from 10 verbose parameters fixed to 1 in `default.ini`; see them with

  ```
  > tail default.ini
  ```

Running in terminal with input file (old fashioned)

Run with your own input file with (compulsory) extension *.ini:

```
>./class my_model.ini
```

With for instance:

```
output = tCl,pCl,lCl,mPk
lensing = yes                # include CMB lensing effect
non linear = halofit         # non-linear P(k) from HALOFIT
root = output/my_model_
write warnings = yes # will alert you if wrong input syntax
Omega_b = 0.05
more comments, ignored because no equal sign in this line
# comment with an =, still ignored thanks to the sharp
```

Running in terminal with input file (old fashioned)

Run with your own input file with (compulsory) extension `*.ini`:

```
>./class my_model.ini
```

With for instance:

```
output = tCl,pCl,lCl,mPk
lensing = yes                # include CMB lensing effect
non linear = halofit         # non-linear P(k) from HALOFIT
root = output/my_model_
write warnings = yes # will alert you if wrong input syntax
Omega_b = 0.05
more comments, ignored because no equal sign in this line
# comment with an =, still ignored thanks to the sharp
```

- Order of lines doesn't matter at all.
- All parameters not passed are fixed to default, i.e. the most reasonable or minimalistic choice (ΛCDM with Planck 2013 bestfit)
- `./class` can take two input files `*.ini` and `*.pre`:

  ```
  >./class my_model.ini cl_permille.pre
  ```

  But one is enough.

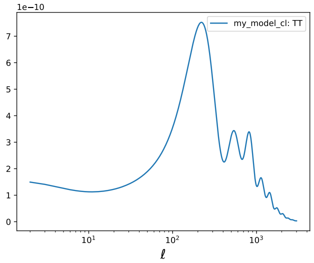
Running in terminal with input file (old fashioned)

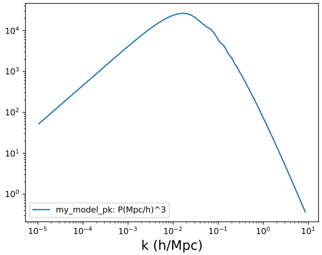
Results are in several files `output/my_model_*.dat`
Can be quickly plotted with provided python script `CPU.py` (Class Plotting Unit):

```
> python CPU.py output/my_model_cl_lensed.dat
> python CPU.py output/my_model_cl.dat -y TT --scale loglin
> python CPU.py output/my_model_pk.dat
```

with options visible with

```
> python CPU.py --help
```





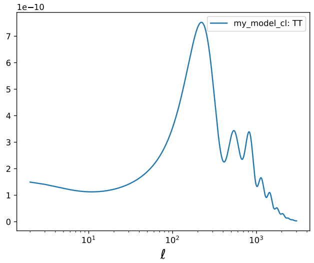
Running in terminal with input file (old fashioned)

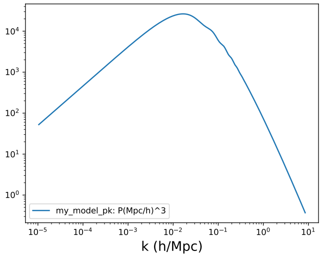
Results are in several files `output/my_model_*.dat`
Can be quickly plotted with provided python script `CPU.py` (Class Plotting Unit):

```
> python CPU.py output/my_model_cl_lensed.dat
> python CPU.py output/my_model_cl.dat -y TT --scale loglin
> python CPU.py output/my_model_pk.dat
```

with options visible with

```
> python CPU.py --help
```





Also provide similar MATLAB script `plot_CLASS_output.m`, get syntax with

```
help plot_class_output
```

Species in public class

- photons: T_cmb or Omega_g or omega_g
- baryons: Omega_b or omega_b
- ultra-relativistic species (massless neutrinos): N_ur or Omega_ur or omega_ur
- cold dark matter: Omega_cdm or omega_cdm (possibly annihilating: annihilation, etc.)
- N_ncdm distinct non-cold dark matter species (massive neutrinos, warm dark matter...): m_ncdm or Omega_ncdm or omega_ncdm plus lots of options
- cold dark matter decaying into dark radiation: Omega_dcdmdr or omega_dcdmdr plus Gamma_dcdm
- cold dark matter interacting with dark radiation: Omega_idm_dr or omega_idm_dr plus cross-section, ...)
- coming soon: dark matter interacting with photons and/or baryons and/or dark radiation
- dark radiation interacting with dark matter and/or with itself: N_idr, plus cross-sections, ...
- spatial curvature Omega_k
- cosmological constant Omega_Lambda
- fluid Omega_fld plus w0_fld, wa_fld, cs2_fld, etc.
- scalar field (quintessence) Omega_scf plus specifications

All details are in explanatory.ini
(densities can all be set to zero, excepted photons and baryons)

Species in public class

Budget equation:

$$\sum_X \Omega_X = 1 + \Omega_k$$

To avoid over-constraining the input, one of the last three (`Omega_Lambda`, `Omega_fld`, `Omega_scf`) must be left unspecified and class will assign it using budget equation.

- default: `Omega_Lambda` is automatically adjusted, assuming `Omega_fld` $=$ `Omega_scf` $= 0$.
- if you pass `Omega_Lambda` $= 0$: `Omega_fld` is automatically adjusted, assuming `Omega_scf` $= 0$.
- if you pass `Omega_Lambda` $= 0$ and `Omega_fld` $= 0$: `Omega_scf` is automatically adjusted.

Allows whatever combination.
E.g. to replace Λ by a dynamical DE fluid with $(w_0, w_a) = (-0.9, 0.1)$:

```
Omega_Lambda=0.
w0_fld=0.9
wa_fld=0.1
```

Running class from python

class as a Python module

- based on wrapper located in python/classy.pyx (developed initially by B. Audren and extended by many others)
- the compilation produces a python module classy.py and installs it on your computer (can be called from anywhere)
- wrapper written in Cython, encapsulates most useful class variables/functions, contains extra functions (e.g. MontePython-motivated)
- goal: obtain, manipulate and plot the results directly within (i)python scripts or notebooks (recommended)

- we will now discuss several examples of scrips/notebooks which are available in scripts/ and notebooks/ (should work with python 2.7 or 3)
- with jupyter installed, open the notebooks with e.g.

```
> jupyter notebook notebooks/warmup.ipnyb
```

- if you can't make it with jupyter, you'll get the same results with

```
> python scripts/warmup.py
```

Python wrapper

First basic example (`notebooks/warmup.ipynb` or `scripts/warmup.py`)

```
# import classy module
from classy import Class
```

```
# create instance of the class "Class"
LambdaCDM = Class()
# pass input parameters
LambdaCDM.set({'omega_b':0.0223828,'omega_cdm':0.1201075,'h':0.67810,'A_s'
     :2.100549e-09,'n_s':0.9660499,'tau_reio':0.05430842})
LambdaCDM.set({'output':'tCl,pCl,lCl,mPk','lensing':'yes','P_k_max_1/Mpc':3.0})
# run class
LambdaCDM.compute()
```

```
# get all C_l output
cls = LambdaCDM.lensed_cl(2500)
# To check the format of cls
cls.keys()
```

```
dict_keys(['pp', 'ell', 'bb', 'ee', 'tt', 'tp', 'te'])
```

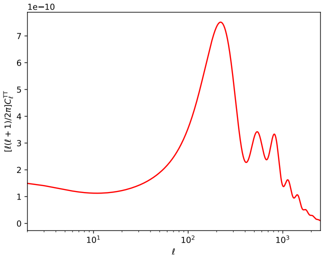
```
ll = cls['ell'][2:]
clTT = cls['tt'][2:]
clEE = cls['ee'][2:]
clPP = cls['pp'][2:]
```

Python wrapper

First basic example (notebooks/warmup.ipynb or scripts/warmup.py)

```
import matplotlib.pyplot as plt
from math import pi
```

```
# plot C_l^TT
plt.figure(1)
plt.xscale('log');plt.yscale('linear');plt.xlim(2,2500)
plt.xlabel(r'$\ell$')
plt.ylabel(r'$[\ell(\ell+1)/2\pi]  C_\ell^\mathrm{TT}$')
plt.plot(ll,clTT*ll*(ll+1)/2./pi,'r-')
```



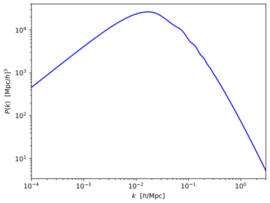
```
plt.savefig('warmup_cltt.pdf')
```

Python wrapper

First basic example (notebooks/warmup.ipynb or scripts/warmup.py)

```
# get P(k) at redhsift z=0
import numpy as np
kk = np.logspace(-4,np.log10(3),1000) # k in h/Mpc
Pk = [] # P(k) in (Mpc/h)**3
h = LambdaCDM.h() # get reduced Hubble for conversions to 1/Mpc
for k in kk:
    Pk.append(LambdaCDM.pk(k*h,0.)*h**3) # function .pk(k,z)
```

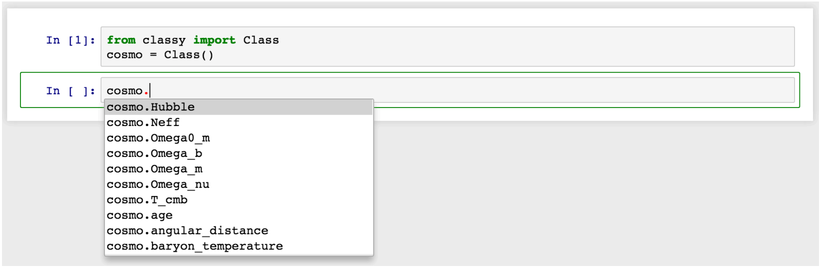
```
# plot P(k)
plt.figure(2)
plt.xscale('log');plt.yscale('log');plt.xlim(kk[0],kk[-1])
plt.xlabel(r'$k \,\,\,\, [h/\mathrm{Mpc}]$')
plt.ylabel(r'$P(k) \,\,\,\, [\mathrm{Mpc}/h]^3$')
plt.plot(kk,Pk,'b-')
```



```
plt.savefig('warmup_pk.pdf')
```

Python wrapper: IPython notebooks

The TAB key after the dot gives you the list of available classy methods (= available functions and quantities) in a scrolling menu:

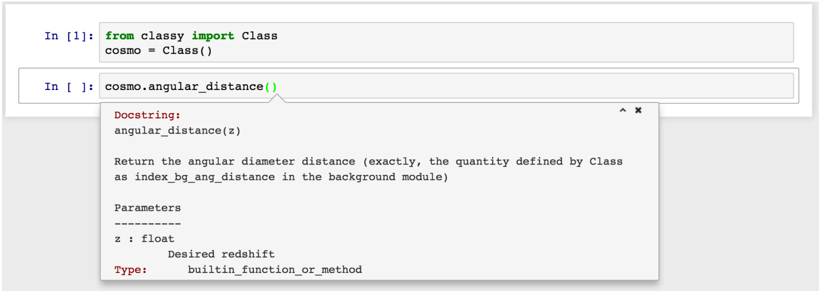


```
In [1]: from classy import Class
        cosmo = Class()

In [ ]: cosmo.
        cosmo.Hubble
        cosmo.Neff
        cosmo.Omega0_m
        cosmo.Omega_b
        cosmo.Omega_m
        cosmo.Omega_nu
        cosmo.T_cmb
        cosmo.age
        cosmo.angular_distance
        cosmo.baryon_temperature
```

Python wrapper: IPython notebooks

The TAB+SHIFT keys after the () gives you a short doc on each method (expand it by clicking +):



```
In [1]: from classy import Class
        cosmo = Class()

In [ ]: cosmo.angular_distance()
```

```
Docstring:
angular_distance(z)

Return the angular diameter distance (exactly, the quantity defined by Class
as index_bg_ang_distance in the background module)

Parameters
----------
z : float
        Desired redshift
Type:      builtin_function_or_method
```

Library of scripts and notebooks downloaded together with public code, meant to let you understand specific aspects:

- `warmup` $\rightarrow$ basic plotting of most common observables
- `cl_ST` $\rightarrow$ tensor contribution to CMB
- `cltt_terms` $\rightarrow$ decomposition of temperature C_l in SW, Doppler, ISW, ...
- `distances` $\rightarrow$ plotting background quantitites
- `thermo` $\rightarrow$ plotting thermodynamical quantitites
- `one_k` $\rightarrow$ transfer functions for given k versus time
- `one_time` $\rightarrow$ transfer functions for given time versus k
- `many_times` $\rightarrow$ colored surface for transfer function versus time and k
- `varying_pann` $\rightarrow$ showing impact of variation of one parameter
- `varying_Neff` $\rightarrow$ similar (more advanced)
- `neutrinohierarchy` $\rightarrow$ playing with multiple neutrino masses
- `check_PPF_approx` $\rightarrow$(technical) check consistency of Parametrized Post-Friedmann description of fluid DE crossing phantom divide
- `Growth_with_w` $\rightarrow$(technical) compute growth factor with fluid DE

Dynamical indexing rules in `class`

- Indexing is very generic in CLASS, same rules apply everywhere.

Dynamical indexing rules in `class`

- Indexing is very generic in CLASS, same rules apply everywhere.
- Example: we want to define the indices of a vector of background quantities (stored in the background table).

Dynamical indexing rules in `class`

- Indexing is very generic in CLASS, same rules apply everywhere.
- Example: we want to define the indices of a vector of background quantities (stored in the background table).
- We choose an abreviation of 2 letters for these indices, `_bg_`.

Dynamical indexing rules in `class`

- Indexing is very generic in CLASS, same rules apply everywhere.
- Example: we want to define the indices of a vector of background quantities (stored in the background table).
- We choose an abreviation of 2 letters for these indices, `_bg_`.
- Then we declare all possible indices `index_bg_<blabla>` in `include/background.h` (more precisely, inside the structure background, because these indices are necessary for manipulating the background table).

Dynamical indexing rules in class

- Indexing is very generic in CLASS, same rules apply everywhere.
- Example: we want to define the indices of a vector of background quantities (stored in the background table).
- We choose an abreviation of 2 letters for these indices, `_bg_`.
- Then we declare all possible indices `index_bg_<blabla>` in `include/background.h` (more precisely, inside the structure background, because these indices are necessary for manipulating the background table).
- We also declare flags saying whether these indices need to be defined or not.

Dynamical indexing rules in class

In include/background.h:

```
struct background {
    /** input parameters with assigned in the input module*
        */
    double Omega0_cdm;
    ...
    /** flags and indices **/
    int has_cdm;    // can take values _TRUE_ or _FALSE_
    ....

    int index_bg_rho_cdm;
    ...

    int bg_size;

    /** interpolation table **/
    double * background_table;
}
```

Dynamical indexing rules in class

In source/background.c, the function background_indices() called at the beginning of background_init() assigns numerical value to indices, that the user will never need to know (quantities always written symbolically as y[pba->index_bg_rho_cdm])

```
int background_indices(pba,...) {
    /* initialize all flags */
    if (pba->Omega0_cdm != 0.)
       pba->has_cdm = _TRUE_;
    ...
    /* initialize all indices */
    index_bg=0;
    class_define_index(pba->index_bg_rho_cdm,
                       pba->has_cdm,
                       index_bg,
                       1);
    class_define_index(pba->index_bg_rho_fld,
                       pba->has_fld,
                       index_bg,
                       1);
    ...
    pba->bg_size = index_bg;
}
```

Dynamical indexing rules in `class`

This logic is followed everywhere for all groups of indices! Examples:

- in `background.c`: `index_bg_...` for all background variables {A,B,C}
- in `background.c`: `index_bi_...` for backg. var. {B,C} integrated over time
- in `thermodynamics.c`: `index_th_...` for all thermodynamics variables
- in `perturbations.c`: `index_pt_...` perturbation var. integrated over time
- in `perturbations.c`: `index_mt_...` metric perturbations
- in `perturbations.c`: `index_md_...` list of modes (scalar, vector, tensor)
- in `perturbations.c`: `index_ic_...` list of initial conditions (AD, CDI, NID...)
- in `perturbations.c`: `index_tp_...` list of type of required source (temperature, polarisation, matter fluctuation...)
- in `perturbations.c`: `index_ap_...` list of approximations that may be used
- etc. etc.

Check in your `include/*.h` files!

Input management in class

Terminal | Python wrapper

$$\begin{tikzcd}
\text{file \texttt{xxx.ini}} \arrow[d] & \\
\text{\texttt{input\_init\_from\_argument(...)}} \\ \text{(parser)} \arrow[d] & \text{\texttt{.set(...)}} \arrow[dl] \\
\text{\texttt{struct file\_content fc;} (all parameter names/values stored as arrays of strings)} \arrow[d] & \\
\text{\texttt{input\_init(...)}} \arrow[d] & \\
\text{\texttt{input\_read\_parameters(...)}} \\ \text{(assign all default values + interprete input + update some parameters)} \arrow[d] & \\
\text{\textit{relevant} parameters only get stored in the structures of each module} &
\end{tikzcd}$$

For special parameters requiring a shooting method: repeated calls of `input_read_parameters(...)` from `input_init(...)` until shooting target is met.

Input management in class

For normal parameters (no shooting): example of CDM density:

```
/** - Omega_0_cdm (CDM) */
class_call(parser_read_double(pfc,"Omega_cdm",&param1,&
    flag1,errmsg),
           errmsg,
           errmsg);
class_call(parser_read_double(pfc,"omega_cdm",&param2,&
    flag2,errmsg),
           errmsg,
           errmsg);
class_test(((flag1 == _TRUE_) && (flag2 == _TRUE_)),
           errmsg,
           "In input file, you can only enter one of
               Omega_cdm or omega_cdm, choose one");
if (flag1 == _TRUE_)
  pba->Omega0_cdm = param1;
if (flag2 == _TRUE_)
  pba->Omega0_cdm = param2/pba->h/pba->h;
```

Input management in class

For shooting parameters, establish mapping between *target parameter*, *unknown parameter* and *level*. Currently:

target parameter	unknown parameter	level
$100 \times \theta_s$	h	thermodynamics
σ_8	A_s	fourier
$\Omega_{\rm dcdm}$	$\rho^{\rm ini}_{\rm dcdm}$	background
...	...	...

... plus a few others (alternative parametrizations of decaying CDM, quintessence parameters).

If you need to add such parameters: see how it is done e.g. for `100*theta_s` and replicate the structure!

Error management rules in `class`

Run with an input file containing only

```
omega_b = 0.07
```

Error management rules in class

By following a few general rules, we get automatically some very informative error messages like:

```
Error in thermodynamics_init
=>thermodynamics_init(L:292) :error in
    thermodynamics_helium_from_bbn(ppr,pba,pth);
=>thermodynamics_helium_from_bbn(L:1031) :condition (omega_b
     > omegab[num_omegab-1]) is true; You have asked for an
    unrealistic high value omega_b = 7.e-02. The
    corrresponding value of the primordial helium fraction
    cannot be found in the interpolation table. If you
    really want this value, you should fix YHe to a given
    value rather than to BBN
```

We only wrote the piece starting with "`You have asked...`". All the rest was generated automatically by the code. This follows from following everywhere 5 rules.

Error management rules in class

Rule 1:

All functions are of type int, and return either _SUCCESS_ or _FAILURE_ (defined internally in include/common.h: #define _SUCCESS_ 0 , #define _FAILURE_ 1)

```
int function(input, &output) {
    ...
    if (something goes wrong) return _FAILURE_;
    ...
    return _SUCCESS_;
}
```

Error management rules in class

Rule 2:

All functions are called with the macro class_call(.,.,.) (all macros class_xxx(...) are defined in include/common.h):

```
class_call(function(input, &output),
           error_message_from_function,
           error_message_output);
```

This is simply a short-cut for

```
if (function == _FAILURE_) {
    ErrorMsg Transmit_Error_Message;
    sprintf(Transmit_Error_Message,"%s(L:%d) : error in %s;\
        n=>%s",__func__,__LINE__,#function,
        error_message_from_function);
    sprintf(error_message_output,"%s",Transmit_Error_Message
        );
    return _FAILURE_;
}
```

Error management rules in class

Rule 3:

Each of the 9 main structures xx has a field called error_message. Any function in the module xxx.c is called xxx_something() and writes its error message in xx.error_message (if pxx is a pointer to xx, in pxx->error_message).

So if we are in perturbations_init() and we call perturbations_indices() we write:

```
class_call(perturbations_indices(...,ppt),
           ppt->error_message,
           ppt->error_message);
```

But if we are in perturbations_init() and we call background_at_tau() we write:

```
class_call(background_at_tau(...,pba),
           pba->error_message,
           ppt->error_message);
```

Error management rules in class

Rule 4:

Whenever an error could occur, we first write a test with the macro `class_test(.,.,.)`:

```
class_test(condition, error_message, "Some text");
```

or

```
class_test(condition, error_message, "Some text and numbers
    %d %e",n,x);
```

Example:

```
class_test(num_points == 0,
           ppt->error_message,
           "this might be caused by ...");
step = (max-min)/((double)num_points);
```

In the text, no need to say in which function we are, or to write that the number of points is zero, or to put a `\n`, all this is done automatically.

Error management rules in class

Rule 5:

Always allocate memory with the macros class_alloc(), class_calloc(), class_realloc().

Instead of

```
malloc(parray, N*sizeof(double));
```

use

```
class_alloc(parray, N*sizeof(double), pxx->error_message);
```

If allocation fails (N too big, null or negative), the function will automatically return a _FAILURE_ and the code will return an appropriate error message:

```
Error running background_init
=>background_init(L:537):error in background_solve(ppr,pba);
=>background_solve(L:1303):could not allocate pvecback with
    size -8
```

Error management rules in class

Useful CLASS macros:

```
class_call(function, errmsg_input, errmsg_output);
class_call_parallel(...);
class_call_except(...,[line of code;line of code;...;]);

class_test(condition, errmsg_output,"message"[,args]);
class_test_parallel(...);
class_test_except(...,[line of code;line of code;...;]);
class_stop(errmsg_ouput,"message"[,args]);

class_alloc(pointer,size);
class_alloc_parallel(...);
class_realloc(...);
class_calloc(...);
```

You can see them in `include/common.h` files!

Error management rules in `class`

Few special cases:

- in `main/class.c` there is no "higher level" so the 10 initialisation functions are called like e.g.:

```
int main(int argc, char **argv) {
    if (background_init(&pr,&ba) == _FAILURE_) {
    printf("\n\nError running background_init \n=>%s\n"
         ,ba.error_message);
    return _FAILURE_;
    }
```

Error management rules in class

Few special cases:

- in main/class.c there is no “higher level” so the 10 initialisation functions are called like e.g.:

```
int main(int argc, char **argv) {
    if (background_init(&pr,&ba) == _FAILURE_) {
    printf("\n\nError running background_init \n=>%s\n"
        ,ba.error_message);
    return _FAILURE_;
    }
```

- the input module does not have an error message attached to its structure, and just uses the local variable errmsg. So inside this module, the calls read e.g.:

```
class_call(background_ncdm_init(ppr,pba),
               pba->error_message,
               errmsg);
class_call(parser_read_file(...,errmsg),
               errmsg,
               errmsg);
```

Error management rules in class

Few special cases:

- in main/class.c there is no "higher level" so the 10 initialisation functions are called like e.g.:

```
int main(int argc, char **argv) {
    if (background_init(&pr,&ba) == _FAILURE_) {
    printf("\n\nError running background_init \n=>%s\n"
         ,ba.error_message);
    return _FAILURE_;
    }
```

- the input module does not have an error message attached to its structure, and just uses the local variable errmsg. So inside this module, the calls read e.g.:

```
class_call(background_ncdm_init(ppr,pba),
               pba->error_message,
               errmsg);
class_call(parser_read_file(...,errmsg),
               errmsg,
               errmsg);
```

- when calling external functions not in the 10 modules we must pass the error message as an argument:

```
class_call(array_interpolate(...,pba->error_message),
               pba->error_message,
               pba->error_message);
```

Implementing new features `class`

If you want to implement:

- a new species
- a new approximation scheme to simplify some equations in some regime
- a new mathematical description of an existing species (switching on more precise corrections, etc.)
- a new observable or output (new source function, new transfer function, new spectrum...)

the logic is always the same:

Implementing new features class

If you want to implement:

- a new species
- a new approximation scheme to simplify some equations in some regime
- a new mathematical description of an existing species (switching on more precise corrections, etc.)
- a new observable or output (new source function, new transfer function, new spectrum...)

the logic is always the same:

1. define an acronym easy to search in the C files (e.g. for early dark energy: `earde` is good, `ede` is bad because it is inside "redefine", "needed", etc.)

Implementing new features `class`

If you want to implement:

- a new species
- a new approximation scheme to simplify some equations in some regime
- a new mathematical description of an existing species (switching on more precise corrections, etc.)
- a new observable or output (new source function, new transfer function, new spectrum...)

the logic is always the same:

1. define an acronym easy to search in the C files (e.g. for early dark energy: `earde` is good, `ede` is bad because it is inside "redefine", "needed", etc.)
2. think of the feature closest to yours, and find its acronym (e.g. for fluid: `fld`)

Implementing new features `class`

If you want to implement:

- a new species
- a new approximation scheme to simplify some equations in some regime
- a new mathematical description of an existing species (switching on more precise corrections, etc.)
- a new observable or output (new source function, new transfer function, new spectrum...)

the logic is always the same:

1. define an acronym easy to search in the C files (e.g. for early dark energy: `earde` is good, `ede` is bad because it is inside "redefine", "needed", etc.)
2. think of the feature closest to yours, and find its acronym (e.g. for fluid: `fld`)
3. grep for all occurences of `fld` in `include/*.h` and `source/*.c` (normally they are all within some "`if (has_fld){ ...}`" and you can search directly for occurences of `has_fld`)

Implementing new features `class`

If you want to implement:

- a new species
- a new approximation scheme to simplify some equations in some regime
- a new mathematical description of an existing species (switching on more precise corrections, etc.)
- a new observable or output (new source function, new transfer function, new spectrum...)

the logic is always the same:

1. define an acronym easy to search in the C files (e.g. for early dark energy: `earde` is good, `ede` is bad because it is inside "redefine", "needed", etc.)
2. think of the feature closest to yours, and find its acronym (e.g. for fluid: `fld`)
3. grep for all occurences of `fld` in `include/*.h` and `source/*.c` (normally they are all within some "`if (has_fld){ ...}`" and you can search directly for occurences of `has_fld`)
4. duplicate these occurences

Implementing new features `class`

If you want to implement:

- a new species
- a new approximation scheme to simplify some equations in some regime
- a new mathematical description of an existing species (switching on more precise corrections, etc.)
- a new observable or output (new source function, new transfer function, new spectrum...)

the logic is always the same:

1. define an acronym easy to search in the C files (e.g. for early dark energy: `earde` is good, `ede` is bad because it is inside "redefine", "needed", etc.)
2. think of the feature closest to yours, and find its acronym (e.g. for fluid: `fld`)
3. grep for all occurences of `fld` in `include/*.h` and `source/*.c` (normally they are all within some "`if (has_fld){ ...}`" and you can search directly for occurences of `has_fld`)
4. duplicate these occurences
5. change `fld` into `earde`

Implementing new features `class`

If you want to implement:

- a new species
- a new approximation scheme to simplify some equations in some regime
- a new mathematical description of an existing species (switching on more precise corrections, etc.)
- a new observable or output (new source function, new transfer function, new spectrum...)

the logic is always the same:

1. define an acronym easy to search in the C files (e.g. for early dark energy: `earde` is good, `ede` is bad because it is inside "redefine", "needed", etc.)
2. think of the feature closest to yours, and find its acronym (e.g. for fluid: `fld`)
3. grep for all occurences of `fld` in `include/*.h` and `source/*.c` (normally they are all within some "`if (has_fld){ ...}`" and you can search directly for occurences of `has_fld`)
4. duplicate these occurences
5. change `fld` into `earde`
6. change some equations to describe the specific properties of your feature

Adding parameters in the wrapper

Example of `python` wrapper:

```
# redeclaration of relevant CLASS variables in cython
python/cclassy.pxd
# wrapper's function (.set(), .compute(), .lensed_cl(), ...)
python/classy.py
```

Don't edit any other! (generated automatically at compilation, or for testing or module installation)

Adding parameters in the wrapper

In `python/cclassy.pxd` relevant variables redeclared inside the structure to which they belong:

```
cdef struct background:
    ...
    double age
    ...
cdef struct thermodynamics
    ...
    double z_reio
    ...
```

Indeed, in the C code, `pba->age`, `pth->z_reio` exist...
When defining new parameter in C code that should be accessible from outside: redeclare them here!

Adding parameters in the wrapper

E.g.: new model of Early Dark Energy.

In `include/background.h`:

```
struct background{
  ...
  double rho_earde;
  ...
  }
```

In `python/cclassy.pxd`:

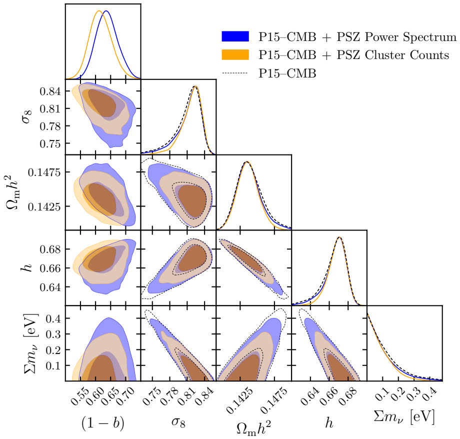
```
cdef struct background:
  ...
  double rho_earde
  ...
```

Recompile after a `make clean` !

Interface with sampler

Many are compatible with CLASS! Non-exhaustive list:

- Bayesian samplers:
  - MontePython (B. Audren, T. Brinckmann, J.L. + others), python, https://github.com/brinckmann/montepython_public



Bolliet et al. 2019

Interface with sampler

Many are compatible with CLASS! Non-exhaustive list:

- Bayesian samplers:
  - MontePython (B. Audren, T. Brinckmann, J.L. + others), python,
    `https://github.com/brinckmann/montepython_public`
  - cobaya (J. Torrado, A. Lewis), python,
    `https://cobaya.readthedocs.io`
  - cosmosis (J. Zuntz), python,
    `https://bitbucket.org/joezuntz/cosmosis`
  - NumCosmo, (S. Dias Pinto Vitenti, M. Penna-Lima, C. Doux), C with GObject framework (callable from Perl, Python, etc.),
    `https://numcosmo.github.io/`
- Frequentist minimizers:
  - CAMEL (LAL Orsay), C++, uses MINUIT,
    `http://camel.in2p3.fr`

Interface with sampler

Many are compatible with CLASS! Non-exhaustive list:

- Bayesian samplers:
  - MontePython (B. Audren, T. Brinckmann, J.L. + others), python, `https://github.com/brinckmann/montepython_public`
  - cobaya (J. Torrado, A. Lewis), python, `https://cobaya.readthedocs.io`
  - cosmosis (J. Zuntz), python, `https://bitbucket.org/joezuntz/cosmosis`
  - NumCosmo, (S. Dias Pinto Vitenti, M. Penna-Lima, C. Doux), C with GObject framework (callable from Perl, Python, etc.), `https://numcosmo.github.io/`
- Frequentist minimizers:
  - CAMEL (LAL Orsay), C++, uses MINUIT, `http://camel.in2p3.fr`

At least in MontePython, Cobaya and CAMEL: no declaration of cosmological parameters in the sampler! No need to modify anything if you add new parameters! (whatever parameter `'name'` read in input file just passed directly through `class.set('name':,...)`