The Cosmic Linear Anisotropy Solving System (CLASS)

Julien Lesgourgues

November 22, 2018

Where to find information and documentation on `CLASS` ?

- **for what the code can actually compute:** all possible input parameters, all coded cosmological models, all functionalities, all observables, etc.: read the file `explanatory.ini` in the main `CLASS` directory: it is a reference file where we keep track of all possible input.
- **for the physics and equations used in the code:** mainly, the following papers:
  - **"Cosmological perturbation theory in the synchronous and conformal Newtonian gauges"**
    C. P. Ma and E. Bertschinger.
    astro-ph/9506072
    10.1086/176550
    Astrophys. J. **455**, 7 (1995)
  - **"The Cosmic Linear Anisotropy Solving System (CLASS) II: Approximation schemes"**
    D. Blas, J. Lesgourgues and T. Tram.
    arXiv:1104.2933 [astro-ph.CO]
    10.1088/1475-7516/2011/07/034
    JCAP **1107**, 034 (2011)
  - **"The Cosmic Linear Anisotropy Solving System (CLASS) IV: efficient implementation of non-cold relics"**
    J. Lesgourgues and T. Tram.
    arXiv:1104.2935 [astro-ph.CO]
    10.1088/1475-7516/2011/09/032
    JCAP **1109**, 032 (2011)
  - **"Optimal polarisation equations in FLRW universes"**
    T. Tram and J. Lesgourgues.
    arXiv:1305.3261 [astro-ph.CO]
    10.1088/1475-7516/2013/10/002
    JCAP **1310**, 002 (2013)

  - **"Fast and accurate CMB computations in non-flat FLRW universes"**
    J. Lesgourgues and T. Tram.
    arXiv:1312.2697 [astro-ph.CO]
    10.1088/1475-7516/2014/09/032
    JCAP **1409**, no. 09, 032 (2014)
  - **"The CLASSgal code for Relativistic Cosmological Large Scale Structure"**
    E. Di Dio, F. Montanari, J. Lesgourgues and R. Durrer.
    arXiv:1307.1459 [astro-ph.CO]
    10.1088/1475-7516/2013/11/044
    JCAP **1311**, 044 (2013)

- **for the structure, style, and concrete aspects of the code:** this documentation; plus the slides of our `CLASS` lectures, for instance those from Tokyo 2014 available at

  `https://www.dropbox.com/sh/ma5muh76sggwk8k/AABl_DDUBEzAjjdywMjeTya2a?dl=0`

  in the folder `CLASS_Lecture_slides/`.

- **for the python wrapper of `CLASS` :** at the moment, the best is the slides from these lectures, for instance following the previous link and looking into

  `CLASS_Lecture_slides/lecture7_wrapper.pdf`

  and into

  `IPython_Notebooks/`

  for example of python sessions. We will expand soon the documentation on this part with a dedicated webpage.

1 Overall architecture of CLASS

1.1 Files and directories

After downloading `CLASS` , one can see the following files in the root directory contains:

- some example of input files, the most important being `explanatory.ini`. a reference input file containing all possible flags, options and physical input parameters. While this documentation explains the structure and

use of the code, `explanatory.ini` can be seen as the *physical* documentation of `CLASS` . The other input file are alternative parameter input files (ending with `.ini`)and precision input files (ending with `.pre`)

- the `Makefile`, with which you can compile the code by typing `make clean; make;` this will create the executable `class` and some binary files in the directory `build/`. The `Makefile` contains other compilation options that you can view inside the file.
- `CPU.py` is a python script designed for plotting the `CLASS` output; for documentation type `python CPU.py --help`
- `plot_CLASS_output.m` is the counterpart of `CPU.py` for MatLab
- there are other input files for various applications: an example of a non-cold dark matter distribution functions (`psd_FD_single.dat`), and examples of evolution and selection functions for galaxy number count observables (`myevolution.dat, myselection.dat`).

Other files are split between the following directories:

- `source/` contains the C files for each `CLASS` module, i.e. each block containing some part of the physical equations and logic of the Boltzmann code.
- `tools/` contains purely numerical algorithms, applicable in any context: integrators, simple manipulation of arrays (derivation, integration, interpolation), Bessel function calculation, quadrature algorithms, parser, etc.
- `main/` contains the main module `class.c` with the main routine `class(...)`, to be used in interactive runs (but not necessarily when the code is interfaced with other ones).
- `test/` contains alternative main routines which can be used to run only some part of the code, to test its accuracy, to illustrate how it can be interfaced with other codes, etc.
- `include/` contains all the include files with a `.h` suffix.
- `output/` is where the output files will be written by default (this can be changed to another directory by adjusting the input parameter `root = <...>`)
- `python/` contains the python wrapper of `CLASS` , called classy (see `python/README`)
- `cpp/` contains the C++ wrapper of `CLASS` , called ClassEngine (see `cpp/README`)
- `doc/` contains the automatic documentation (manual and input files required to build it)

- `external_Pk/` contains exemples of external codes that can be used to generate the primordial spectrum and be interfaced with `CLASS` , when one of the many options already built inside the code are not sufficient.
- `bbn/` contains interpolation tables produced by BBN codes, in order to predict e.g. $Y_{\rm He}(\omega_b, \Delta N_{\rm eff})$.
- `hyrec/` contains the recombination code HyRec of Yacine Ali-Haïmoud and Chris Hirata, that can be used as an alternative to the built-in Recfast (using the input parameter `recombination = <...>`).

1.2 The ten-module backbone

1.2.1 Ten tasks

The purpose of CLASS consists in computing some power spectra for a given set of cosmological parameters. This task can be decomposed in few steps or modules:

1. set input parameter values.
2. compute the evolution of cosmological background quantitites.
3. compute the evolution of thermodynamical quantitites (ionization fractions, etc.)
4. compute the evolution of source functions $S(k,\tau)$ (by integrating over all perturbations).
5. compute the primordial spectra.
6. eventually, compute non-linear corrections at small redshift/large wavenumber.
7. compute transfer functions in harmonic space $\Delta_l(k)$ (unless one needs only Fourier spectra $P(k)$'s and no harmonic spectra C_l's).
8. compute the observable power spectra C_l's (by convolving the primordial spectra and the harmonic transfer functions) and/or $P(k)$'s (by multiplying the primordial spectra and the appropriate source functions $S(k,\tau)$).
9. eventually, compute the lensed CMB spectra (using second-order perturbation theory)
10. write results in files (when `CLASS` is used interactively. The pyhton wrapper would not go to this step and just keep the output stored internally).

1.2.2 Ten structures

In CLASS, each of these steps is associated with a structure:

1. `struct precision` for input precision parameters (input physical paramaters are dispatched among the other structures listed below)
2. `struct background` for cosmological background,
3. `struct thermo` for thermodynamics,
4. `struct perturbs` for source functions,
5. `struct primordial` for primordial spectra,
6. `struct nonlinear` for nonlinear corrections,
7. `struct transfers` for transfer functions,
8. `struct spectra` for observable spectra,
9. `struct lensing` for lensed CMB spectra,
10. `struct output` for auxiliary variable describing the output format.

A given structure contains "everything concerning one step that the subsequent steps need to know" (for instance, `struct perturbs` contains everything about source functions that the transfer module needs to know). In particular, each structure contains one array of tabulated values (for `struct background` , background quantitites as a function of time, for `struct thermo`, thermodynamical quantitites as a function of redshift, for `struct perturbs`, sources $S(k, \tau)$, etc.). It also contains information about the size of this array and the value of the index of each physical quantity, so that the table can be easily read and interpolated. Finally, it contains any derived quantity that other modules might need to know. Hence, the comunication from one module A to another module B consists in passing a pointer to the structure filled by A, and nothing else.

All "precision parameters" are grouped in the single structure `struct precision`. The code contains *no other arbitrary numerical coefficient.*

1.2.3 Ten modules

Each structure is defined and filled in one of the following modules (and precisely in the order below):

1. `input.c`
2. `background.c`
3. `thermodynamics.c`
4. `perturbations.c`

5. `primordial.c`
6. `nonlinear.c`
7. `transfer.c`
8. `spectra.c`
9. `lensing.c`
10. `output.c`

Each of these modules contains at least three functions:

- *module*`_init(...)`
- *module*`_free(...)`
- *module_something*`_at_`*somevalue*`(...)`

where *module* is one of `input, background, thermodynamics, perturb, primordial, nonlinear, transfer, spectra, lensing, output`.

The first function allocates and fills each structure. This can be done provided that the previous structures in the hierarchy have been already allocated and filled. In summary, calling one of *module*`_init(...)` amounts in solving entirely one of the steps 1 to 10.

The second function deallocates the fields of each structure. This can be done optionally at the end of the code (or, when the code is embedded in a sampler, this *must* be done between each execution of CLASS, and especially before calling *module*`_init(...)` again with different input parameters).

The third function is able to interpolate the pre-computed tables. For instance, `background_init()` fills a table of background quantitites for discrete values of conformal time τ, but `background_at_tau(tau, * values)` will return these values for any arbitrary τ.

Note that functions of the type *module_something*`_at_`*somevalue*`(...)` are the only ones which are called from another module, while functions of the type *module*`_init(...)` and *module*`_free(...)` are the only one called by the main executable. All other functions are for internal use in each module.

When writing a C code, the ordering of the functions in the *.c file is in principle arbitrary. However, for the sake of clarity, we always respected the following order in each `CLASS` module:

1. all functions that may be called by other modules, i.e. "external functions", usually named like *module_something*`_at_`*somevalue*`(...)`
2. then, *module*`_init(...)`
3. then, *module*`_free()`
4. then, all functions used only internally by the module

1.3 `main()` function(s)

1.3.1 The `main.c` file

The main executable of CLASS is the function `main()` located in the file `main/main.c`. This function consist only in the following lines (not including comments and error-management lines explained later):

```
main() {
   struct precision pr;
   struct background ba;
   struct thermo th;
   struct perturbs pt;
   struct primordial pm;
   struct nonlinear nl;
   struct transfers tr;
   struct spectra sp;
   struct lensing le;
   struct output op;

   input_init_from_arguments(argc, argv,&pr,&ba,&th,&pt,&tr,&pm,&sp,&nl,&le,&op,errmsg);
   background_init(&pr,&ba);
   thermodynamics_init(&pr,&ba,&th);
   perturb_init(&pr,&ba,&th,&pt);
   primordial_init(&pr,&pt,&pm);
   nonlinear_init(&pr,&ba,&th,&pt,&pm,&nl);
   transfer_init(&pr,&ba,&th,&pt,&nl,&tr);
   spectra_init(&pr,&ba,&pt,&pm,&nl,&tr,&sp);
   lensing_init(&pr,&pt,&sp,&nl,&le);
   output_init(&ba,&th,&pt,&pm,&tr,&sp,&nl,&le,&op)

   /****** done ******/

   lensing_free(&le);
   spectra_free(&sp);
   transfer_free(&tr);
   nonlinear_free(&nl);
   primordial_free(&pm);
   perturb_free(&pt);
   thermodynamics_free(&th);
   background_free(&ba);
}
```

We can come back on the role of each argument. The arguments above are all pointers to the 10 structures of the code, excepted `argc, argv` which contains the input files passed by the user, and `errmsg` which contains the output error

message of the input module (error management will be described below).

`input_init_from_arguments` needs all structures, because it will set the precision parameters inside the `precision` structure, and the physical parameters in some fields of the respective other structures. For instance, an input parameter relevant for the primordial spectrum calculation (like the tilt n_s) will be stored in the `primordial` structure. Hence, in `input_init_from_arguments`, all structures can be seen as output arguments.

Other *module*`_init()` functions typically need all previous structures, which contain the result of the previous modules, plus its own structures, which contain some relevant input parameters before the function is called, as well as all the result form the module when the function has been executed. Hence all passed structures can be seen as input argument, excepted the last one which is both input and output. An example is `perturb_init(&pr,&ba,&th,&pt)`.

Each function *module*`_init()` does not need *all* previous structures, it happens that a module does not depend on a *all* previous one. For instance, the primordial module does not need information on the background and thermodynamics evolution in order to compute the primordial spectra, so the dependency is reduced: `primordial_init(&pr,&pt,&pm)`.

Each function *module*`_init()` only deallocates arrays defined in the structure of their own module, so they need only their own structure as argument. (This is possible because all structures are self-contained, in the sense that when the structure contains an allocated array, it also contains the size of this array). The first and last module, `input` and `output`, have no `input_free()` or `output_free()` functions, because the structures `precision` and `output` do not contain arrays that would need to be de-allocated after the execution of the module.

1.3.2 The `test_<...>.c` files

For a given purpose, somebody could only be interested in the intermediate steps (only background quantities, only the thermodynamics, only the perturbations and sources, etc.) It is then straightforward to truncate the full hierarchy of modules 1, ... 10 at some arbitrary order. We provide several "reduced executables" achieving precisely this. They are located in `test/test_`*module*`.c` (like, for instance, `test/test_perturbations.c`) and they can be complied using the Makefile, which contains the appropriate commands and definitions (for instance, you can type `make test_perturbations`).

The `test/` directory contains other useful example of alternative main functions, like for instance `test_loops.c` which shows how to call `CLASS` within a loop over different parameter values. There is also a version `test/test_loops_omp.c` using a double level of openMP parallelisation: one for running several `CLASS` instances in parallel, one for running each `CLASS` instance on several cores. The comments in theses files are self-explanatory.

2 Input/output

2.1 Input

There are two types of input:

1. "precision parameters" (controlling the precision of the output and the execution time),

2. "input parameters" (cosmological parameters, flags telling to the code what it should compute, ...)

The code can be executed with a maximum of two input files, e.g.

```
./class explanatory.ini cl_permille.pre
```

The file with a `.ini` extension is the cosmological parameter input file, and the one with a `.pre` extension is the precision file. Both files are optional: all parameters are set to default values corresponding to the "most usual choices", and are eventually replaced by the parameters passed in the two input files. For instance, if one is happy with default accuracy settings, it is enough to run with

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

Input files do not necessarily contain a line for each parameter, since many of them can be left to default value. The example file `explanatory.ini` is very long and somewhat indigestible, since it contains all possible parameters, together with lengthy explanations. We recommend to keep this file unchanged for reference, and to copy it in e.g. `test.ini`. In the latter file, the user can erase all sections in which he/she is absolutely not interested (e.g., all the part on isocurvature modes, or on tensors, or on non-cold species, etc.). Another option is to create an input file from scratch, copying just the relevant lines from `explanatory.ini`. For the simplest applications, the user will just need a few lines for basic cosmological parameters, one line for the `output` entry (where one can specifying which power spectra must be computed), and one line for the `root` entry (specifying the prefix of all output files).

The syntax of the input files is explained at the beginning of `explanatory.ini`. Typically, lines in those files look like:

```
parameter1 = value1
free comments
parameter2 = value2 # further comments
# commented_parameter = commented_value
```

and parameters can be entered in arbitrary order. This is rather intuitive. The user should just be careful not to put an "`=`" sign not preceded by a "`#`" sign

inside a comment: the code would then think that one is trying to pass some unidentified input parameter.

The syntax for the cosmological and precision parameters is the same. It is clearer to split these parameters in the two files `.ini` and `.pre`, but there is no strict rule about which parameter goes into which file: in principle, precision parameters could be passed in the `.ini`, and vice-versa. The only important thing is not to pass the same parameter twice: the code would then complain and not run.

The `CLASS` input files are also user-friendly in the sense that many different cosmological parameter bases can be used. This is made possible by the fact that the code does not only read parameters, it "interprets them" with the level of logic which has been coded in the `input.c` module. For instance, the Hubble parameter, the photon density, the baryon density and the ultra-relativistic neutrino density can be entered as:

```
h = 0.7
T_cmb = 2.726    # Kelvin units
omega_b = 0.02
N_eff = 3.04
```

(in arbitrary order), or as

```
H0 = 70
omega_g = 2.5e-5    # g is the label for photons
Omega_b = 0.04
omega_ur = 1.7e-5    # ur is the label for ultra-relativistic species
```

or any combination of the two. The code knows that for the photon density, one should pass one (but not more than one) parameter out of `T_cmb, omega_g, Omega_g` (where small omega's refer to $\omega_i \equiv \Omega_i h^2$). It searches for one of these values, and if needed, it converts it into one of the other two parameters, using also other input parameters. For instance, `omega_g` will be converted into `Omega_g` even if h is written later in the file than `omega_g`: the order makes no difference. Lots of alternatives have been defined. If the code finds that not enough parameters have been passed for making consistent deductions, it will complete the missing information with in-built default values. On the contrary, if it finds that there is too much information and no unique solution, it will complain and return an error.

In summary, the input syntax has been defined in such way that the user does not need to think too much, and can pass his preferred set of parameters in a nearly informal way.

Let us mention a two useful parameters defined at the end of `explanatory.ini`, that we recommend setting to `yes` in order to run the code in a safe way:

`write parameters` = [`yes` or `no`] *(default:* `no`*)*

When set to yes, all input/precision parameters which have been read are written in a file `<root>parameters.ini`, to keep track all the details of this execution; this file can also be re-used as a new input file. Also, with this option, all parameters that have been passed and that the code did not read (because the syntax was wrong, or because the parameter was not relevant in the context of the run) are written in a file `<root>unused_parameters`. When you have doubts about your input or your results, you can check what is in there.

`write warnings` = [`yes` or `no`] *(default:* `no`*)*

When set to yes, the parameters that have been passed and that the code did not read (because the syntax was wrong, or because the parameter was not relevant in the context of the run) are written in the standard output as `[Warning:]...`.

There is also a list of "verbose" parameters at the end of `explanatory.ini`. They can be used to control the level of information passed to the standard output (0 means silent; 1 means normal, e.g. information on age of the universe, etc.; 2 is useful for instance when you want to check on how many cores the run is parallelised; 3 and more are intended for debugging).

This part of the documentation will be expanded with details on default precision and on the proposed alternative precision files.

2.2 Output

The input file may contain a line

`root = <root>`

where `<root>` is a path of your choice, e.g. `output/test_`. Then all output files will start like this, e.g. `output/test_cl.dat`, `output/test_cl_lensed.dat`, etc. Of course the number of output files depends on your settings in the input file. There can be input files for CMB, LSS, background, thermodynamics, transfer functions, primordial spectra, etc. All this is documented in `explanatory.ini`.

If you do not pass explicitly a `root = <root>`, the code will name the output in its own way, by concatenating `output/`, the name of the input parameter file, and the first available integer number, e.g.

`output/explanatory03_cl.dat`, etc.

3 General principles

3.1 Error management

Error management is based on the fact that all functions are defined as integers returning either _SUCCESS_ or _FAILURE_. Before returning _FAILURE_, they write an error message in the structure of the module to which they belong. The calling function will read this message, append it to its own error message, and return a _FAILURE_; and so on and so forth, until the main routine is reached. This error management allows the user to see the whole nested structure of error messages when an error has been met. The structure associated to each module contains a field for writing error messages, called `structure_i.error_message`, where `structure_i` could be one of `background`, `thermo`, `perturbs`, etc. So, when a function from a module i is called within module j and returns an error, the goal is to write in `structure_j.error_message` a local error message, and to append to it the error message in `structure_i.error_message`. These steps are implemented in a macro `class_call()`, used for calling whatever function:

```
class_call(module_i_function(...,structure_i),
           structure_i.error_message,
           structure_j.error_message);
```

So, the first argument of `call_call()` is the function we want to call; the second argument is the location of the error message returned by this function; and the third one is the location of the error message which should be returned to the higher level. Usually, in the bulk of the code, we use pointer to structures rather than structure themselves; then the syntax is

```
class_call(module_i_function(...,pi),
           pi->error_message,
           pj->error_message);
```

where in this generic example, `pi` and `pj` are assumed to be pointers towards the structures `structure_i` and `structure_j`.

The user will find in `include/common.h` a list of additional macros, all starting by `class_...()`, which are all based on this logic. For instance, the macro `class_test()` offers a generic way to return an error in a standard format if a condition is not fulfilled. A typical error message from `CLASS` looks like:

```
Error in module_j_function1
=> module_j_function1 (L:340) :  error in module_i_function2(...)
=> module_i_function2 (L:275) :  error in module_k_function3(...)
...
=> module_x_functionN (L:735) :  your choice of input parameter blabla=30
is not consistent with the constraint blabla<1
```

where the L's refer to line numbers in each file. These error messages are very informative, and are built almost entirely automatically by the macros. For instance, in the above example, it was only necessary to write inside the function `module_x_functionN()` a test like:

```
class_test(blabla >= 1,
           px->error_message,
       "your choice of input parameter blabla=%e is not consistent
with the constraint blabla<%e",
           blabla,blablamax);
```

All the rest was added step by step by the various `class_call()` macros.

3.2 Dynamical allocation of indices

On might be tempted to decide that in a given array, matrix or vector, a given quantity is associated with an explicit index value. However, when modifying the code, extra entries will be needed and will mess up the initial scheme; the user will need to study which index is associated to which quantity, and possibly make an error. All this can be avoided by using systematically a dynamical index allocation. This means that all indices remain under a symbolic form, and in each, run the code attributes automatically a value to each index. The user never needs to know this value. *This part of the documentation will be expanded with concrete guidelines.*

3.3 No hard coding

Any feature or equation which could be true in one cosmology and not in another one should not be written explicitly in the code, and should not be taken as granted in several other places. Discretization and integration steps are usually defined automatically by the code for each cosmology, instead of being set to something which might be optimal for minimal models, and not sufficient for other ones. *This part of the documentation will be expanded with concrete examples.*

3.4 Modifying the code

This part of the documentation will be expanded with concrete guidelines.

4 Units and equations

This part of the documentation will be expanded with concrete guidlines.