# Tools for cosmology: the Cosmological Linear Anisotropy Solving System (CLASS)[1]

Julien Lesgourgues

TTK, RWTH Aachen University

euclid-school16, Narbonne, 23.08.2016

[1] code developed together with Thomas Tram plus many others

## Introduction

Double challenge:

## Introduction

Double challenge:

- My first CLASS course in 4 hours (usually, our course “tools for cosmology” includes 13 hours on CLASS + 5 hours on MontePython)

## Introduction

Double challenge:

- My first CLASS course in 4 hours (usually, our course "tools for cosmology" includes 13 hours on CLASS + 5 hours on MontePython)
- Ziad $\Longrightarrow$ 4 "wallclock" hours = 2 "effective" hours

## Context

CLASS is the 5th public Boltzmann code covering all basic cosmology:

1. COSMICS package in f77 (Bertschinger 1995)
2. CMBFAST in f77 (Seljak & Zaldarriaga 1996)
3. CAMB in f90/2000 (Lewis & Challinors 1999)
4. CMBEASY in C++ (Doran 2003)
5. CLASS in C (Lesgourgues & Tram 2011)

... and there will probably be 1 or 2 more! But only CAMB and CLASS are still developed and kept to high precision level.

## Context

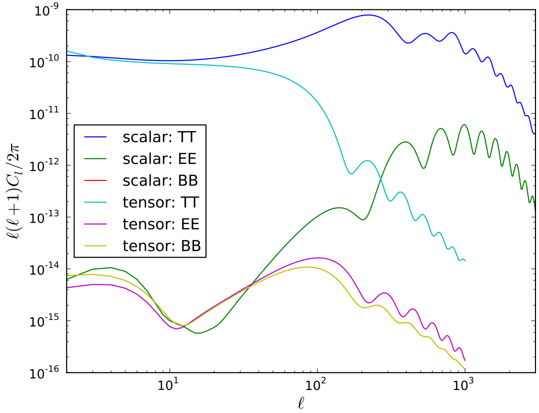
Project started on request of Planck science team, in order to have a tool independent from CAMB, and check for possible Boltzmann-code-induced bias in parameter extraction. The CLASS-CAMB comparison has triggered progress in the accuracy of both codes. Agreement established at $10^{-4}$ (0.01%) level for CMB observables, using highest-precision settings in both codes. But the CLASS projected expandeded and went much further the initial Planck purposes.

CLASS is meant to be

- more general (more models, more output/observables)
- more modern (structured, modular, flexible, wrap-able: wrapper for python, C++, automatic precision test code)
- more friendly (documented, structured, easy to understand) and hence easier to modify (coding additional models/observables)
- equally accurate and fast (in principle, better structure offers more possibilities to make it more accurate and fast than competitors in future)

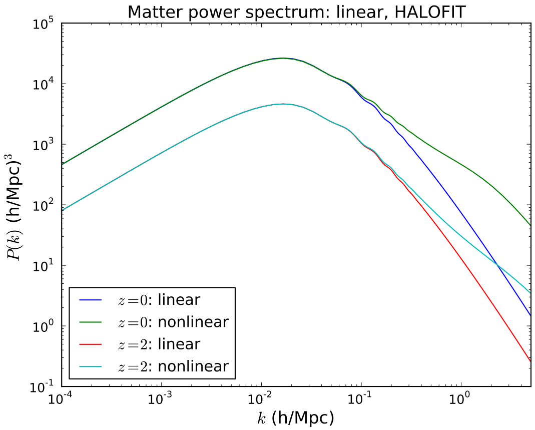
# With CLASS you can get:

The CMB anisotropy spectra:



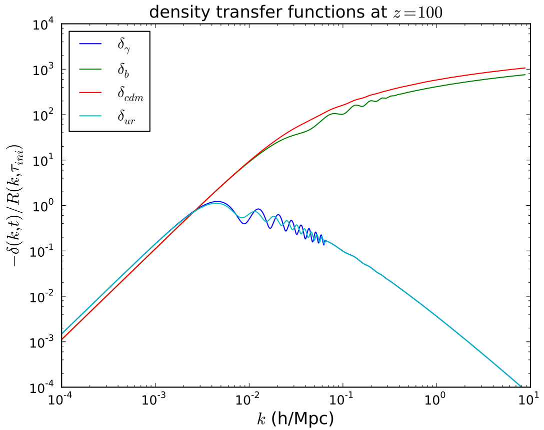
## With CLASS you can get:

The matter power spectrum:



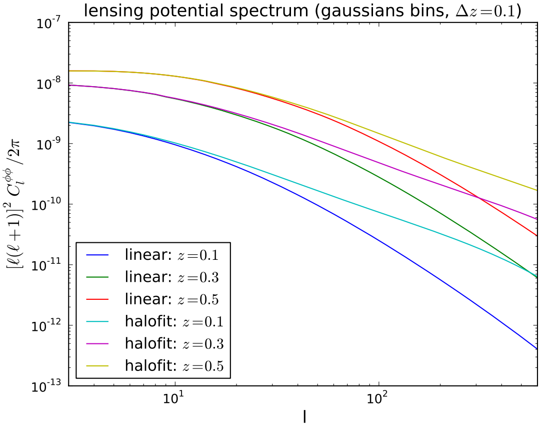
# With CLASS you can get:

The transfer functions at a given time/redshift (e.g. initial conditions for N-body):



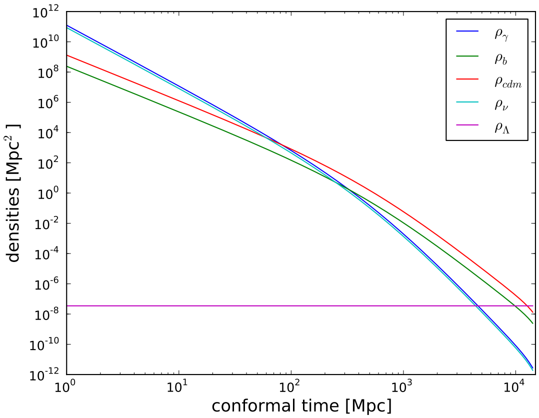
# With CLASS you can get:

The matter density (number count) $C_l$'s, or the lensing $C_l$'s (with arbitrary selection/window functions):



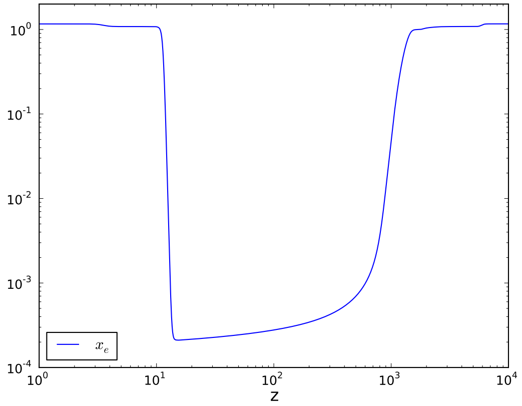
# With CLASS you can get:

The background evolution in a given cosmological model:



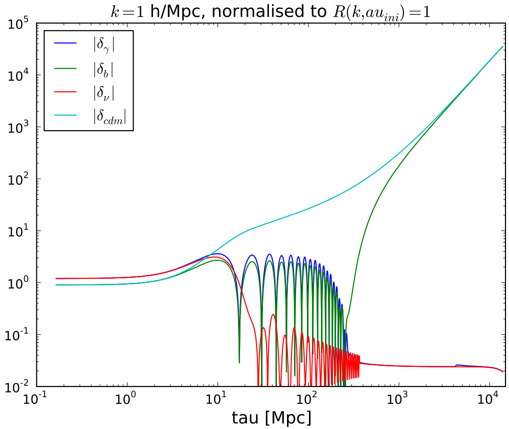
## With CLASS you can get:

The thermal history in a given cosmological model:



# With CLASS you can get:

The time evolution of perturbations for individual Fourier modes:



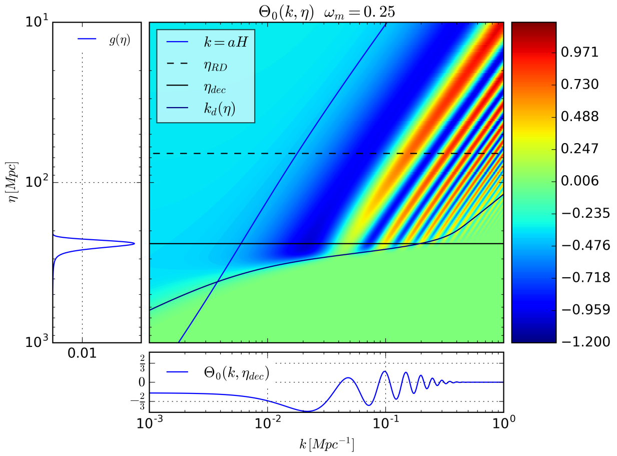
## With CLASS you can get:

... and several other quantities, for instance:

- distance–redshift relations, sound horizon, characteristic redshifts;
- primordial spectrum for given inflationary potential;
- decomposition of CMB $C_l$'s in intrinsic, Sachs-Wolfe, Doppler, ISW, etc.;
- decomposition of galaxy number count $C_l$'s in density, RSD, lensing, etc.;
- ...

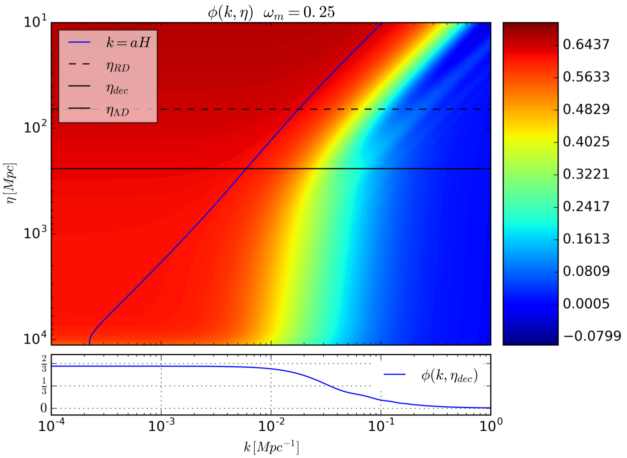
## With CLASS you can get:

... if you use CLASS as a python module you can extract all kind of output or intermediate quantities, manipulate them in various way and make all kinds of computations or nice plots:



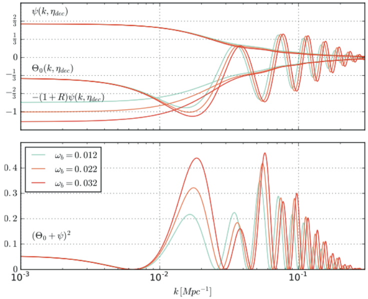
## With CLASS you can get:

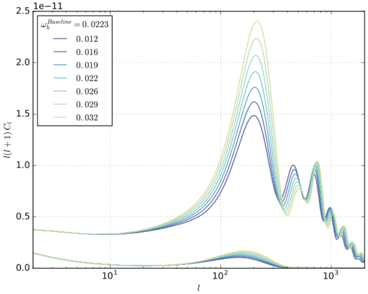
… if you use CLASS as a python module you can extract all kind of output or intermediate quantities, manipulate them in various way and make all kinds of computations or nice plots:



# With CLASS you can get:

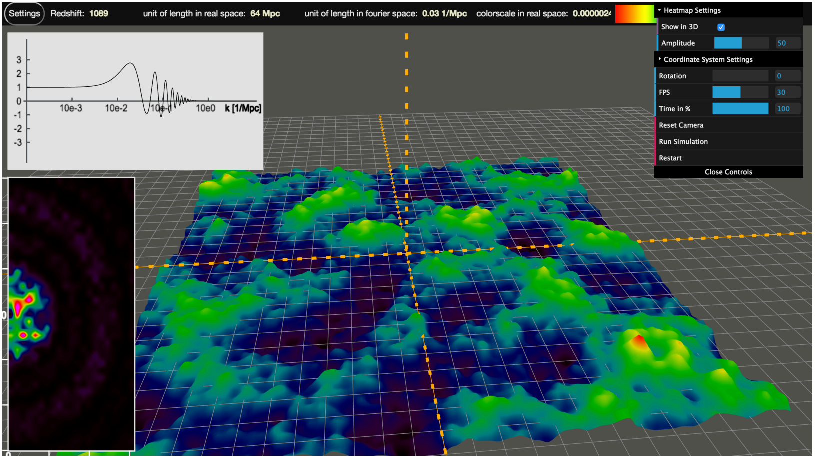
... if you use CLASS as a python module you can extract all kind of output or intermediate quantities, manipulate them in various way and make all kinds of computations or nice plots:





# With CLASS you can get:

... and movies of CMB perturbations in 2D slices of early universe with our Real space graphical interface (not yet included in public distribution, please be patient or ask for it by email); here is a snapshot:



## With CLASS you can get:

... all this for a wide range of cosmological models: all those implemented in the public CAMB code, plus several other ingredients, especially in the sectors of:

- primordial perturbations (internal inflationary perturbation module with given $V(\phi)$, takes arbitrary BSI spectra, correlated isocurvature modes),
- neutrinos (chemical potentials, arbitrary phase-space distributions, flavor mixing...),
- Dark Matter (warm, decaying, annihilating, interacting...),
- Dark Energy (fluid with CLP+ sound speed, quintessence with given $V(\phi)$)
- also Modified Gravity if you try the recently released HiCLASS branch (Bellini, Sawicki, Zumalacarregui, http://www.hiclass-code.net)

## Our program...

This is what we can fit in $\sim 3.5$ hours:

1. 14 general coding principles of the code
2. Basic running
3. Plotting
4. Python wrapper and notebook

with short hands-on throughout the lecture.

That should be sufficient if you plan some efficient and even advanced use of the code, but not if you plan to modify the code significantly by yourself.

## ... and beyond!

If you do plan to modify the code significantly, on top of the official documentation:

`http://www.class-code.net` $\Longrightarrow$ click: `online html documentation`

you may have a look at extra material and corrected exercises e.g. from the Dropbox folder of our lectures at IPMU Tokyo (2014):

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

1. In `Tokyo_Tools/CLASS_Lecture_Slides`: topics not covered here:

   lecture2_structure.pdf
   lecture5_index_and_error.pdf
   lecture6_input_module.pdf
   lecture8_background.pdf
   lecture9_thermodynamics.pdf
   lecture10_perturbations.pdf
   lecture11_non_cold_dark_matter.pdf
   lecture12_primordial.pdf
   lecture13_other_modules.pdf
   lecture14a_testing.pdf
   lecture14b_adding_species.pdf

2. In `Tokyo_Tools/CLASS_Exercises`: exercises.pdf, exercises_correction.pdf (Exercises 0, 1a, 1b, 1c, 1d, 2a can be done directly after this course; 1e, 1f, 2b, 3 require reading at least lectures 2, 5, 6, 14b above)
3. In `Tokyo_Tools/IPython_Notebooks`: solution to the same exercises entirely in python
4. + slides/exercises on MontePython

## Hand-on during the lecture

When you see the logo followed by some instructions, you can check things by yourself on your terminal. For that, you should be in the class directory (e.g. `class_public-2.5.0/`), with the code installed, and seeing among others:

```
class     # C executable
explanatory.ini    # reference input file
output/  # directory for output files
include/ # header files (*.h) containing declarations
source/  # the 10 important modules of CLASS
main/    # main CLASS function: short, just calls 10 modules
test/    # other main functions for testing part of the code
tools/   # auxiliary pieces of code (numerical methods)
python/  # python wrapper of CLASS
cpp/     # C++ wrapper of CLASS
```

plus a few other directories containing ancillary data (`bbn/`) or external code (`hyrec/`)

## 14 general coding principles of `class`

Our efforts for ensuring flexibility and friendliness in CLASS, summarised in 14 key points

## 14 general coding principles of `class`

**1. Written in plain C with no external libraries**

C is free, diffuse, easy, fast. Self-contained and ready to install, straightforward to compile. (However parallelisation requires OpenMP, and the python wrapper requires the installation of python modules compatible with C compiler: both are more delicate on Mac OS $\geq$ 10.9).

## 14 general coding principles of `class`

**1. Written in plain C with no external libraries**

C is free, diffuse, easy, fast. Self-contained and ready to install, straightforward to compile. (However parallelisation requires OpenMP, and the python wrapper requires the installation of python modules compatible with C compiler: both are more delicate on Mac OS $\geq$ 10.9).

**2. Input parameters are "interpreted"**

Some basic logic has been incorporated in the code. Easy to elaborate further.

Examples:
- expects only one out of $\{H_0,\ h,\ 100 \times \theta_s\}$, otherwise complains;
- missing ones inferred from given one
- same with $\{T_{\rm cmb},\ \Omega_\gamma,\ \omega_\gamma\}$, $\{\Omega_{\rm ncdm},\ \omega_{\rm ncdm},\ m_\nu\}$, $\{\Omega_{\rm ur},\ \omega_{\rm ur},\ N_{\rm ur}\}$,...

Look at beginning of explanatory.ini!!!

```
class$ more explanatory.ini
```

`explanatory.ini` = monster file, with all possible input parameters, plays role of an end-user documentation. Keep it as a reference! Don't modify it! Copy it first, or paste relevant parts to your own file!

## 14 general coding principles of `class`

**1. Written in plain C with no external libraries**

C is free, diffuse, easy, fast. Self-contained and ready to install, straightforward to compile. (However parallelisation requires OpenMP, and the python wrapper requires the installation of python modules compatible with C compiler: both are more delicate on Mac OS $\geq$ 10.9).

**2. Input parameters are "interpreted"**

Some basic logic has been incorporated in the code. Easy to elaborate further.

Examples:
- expects only one out of $\{H_0,\ h,\ 100 \times \theta_s\}$, otherwise complains;
- missing ones inferred from given one
- same with $\{T_{\rm cmb},\ \Omega_\gamma,\ \omega_\gamma\}$, $\{\Omega_{\rm ncdm},\ \omega_{\rm ncdm},\ m_\nu\}$, $\{\Omega_{\rm ur},\ \omega_{\rm ur},\ N_{\rm ur}\}$,...

**3. Perturbation equations and notations taken literally from well-known Ma & Bertschinger (`astro-ph/9506072`) paper ...**

... rather than specific notations of one given group, or mixed notations from various origins.

For non-flat universes we found and published the simplest possible generalisation of Ma & Bertschinger notations, (`arXiv:1305.3261`).

## 14 general coding principles of `class`

**4. Code intensively documented**

Inside `.h` and `.c` files: as many comment lines as C lines. On web page and in `doc/` folder: extensive PDF and html documentation generated automatically with `doxygen`.

# 14 general coding principles of `class`

### 4. Code intensively documented

Inside `.h` and `.c` files: as many comment lines as C lines. On web page and in `doc/` folder: extensive PDF and html documentation generated automatically with `doxygen`.

### 5. Easy units

Inside the code, all important variables are either dimensionless or in Mpc$^n$
(excepted inside recombination part, developped externally: `recfast` or `hyrec`)

Of course the output may have different units if more convenient/traditional (e.g. $k$ in $h$/Mpc, etc.)

# 14 general coding principles of `class`

**4. Code intensively documented**

Inside `.h` and `.c` files: as many comment lines as C lines. On web page and in `doc/` folder: extensive PDF and html documentation generated automatically with `doxygen`.

**5. Easy units**

Inside the code, all important variables are either dimensionless or in Mpc$^n$
(excepted inside recombination part, developped externally: `recfast` or `hyrec`)
Of course the output may have different units if more convenient/traditional (e.g. $k$ in $h$/Mpc, etc.)

**6. No hard coding, for example:**

• Never write a sampling step scheme in physical units; code infers sampling as given fraction of dimensionless physical quantities;
• Never write the index of an array as an integer; indexing done automatically and internally by the code; use symbolic index names; e.g. `index_pt_delta_cdm`

# 14 general coding principles of `class`

**4. Code intensively documented**

Inside `.h` and `.c` files: as many comment lines as C lines. On web page and in `doc/` folder: extensive PDF and html documentation generated automatically with `doxygen`.

**5. Easy units**

Inside the code, all important variables are either dimensionless or in Mpc$^n$
(excepted inside recombination part, developped externally: `recfast` or `hyrec`)
Of course the output may have different units if more convenient/traditional (e.g. $k$ in $h$/Mpc, etc.)

**6. No hard coding, for example:**

• Never write a sampling step scheme in physical units; code infers sampling as given fraction of dimensionless physical quantities;
• Never write the index of an array as an integer; indexing done automatically and internally by the code; use symbolic index names; e.g. `index_pt_delta_cdm`

**7. No global variables**

All variables passed as arguments of functions. Important for readability and parallelisation.

## 14 general coding principles of class

8. Clear modular structure

Dinstinct modules with separate physical tasks. No duplicate equations.

```
 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
```

E.g.: Friedmann equation appears in one single place. Same for linearised Einstein equations. Ideal for implementing modified gravity theories.

Search `Friedmann` in source/background.c

## 14 general coding principles of `class`

9. All precision variables grouped in one single place (`input.c`), and even inside a single structure ‘`precision`’

There are... many. True for any code, but they are usually hidden and spread!

## 14 general coding principles of class

9. All precision variables grouped in one single place (input.c), and even inside a single structure 'precision'

There are... many. True for any code, but they are usually hidden and spread!

10. Given "ingredient" always implemented between brackets, in zone switched by a flag

- adding new physics does not slow down the code or compromise its readability.
- incentive to add lots of new things even if rarely used, with no drawback.
- with a search, one can localise all the parts of the code related to a given ingredient.

Examples:
```
if (has_fld == TRUE) {...}
if (has_cmb_lensing == TRUE) {...}
```

Search has_fld in source/background.c

# 14 general coding principles of `class`

**9. All precision variables grouped in one single place (`input.c`), and even inside a single structure '`precision`'**

There are... many. True for any code, but they are usually hidden and spread!

**10. Given "ingredient" always implemented between brackets, in zone switched by a flag**

- adding new physics does not slow down the code or compromise its readability.
- incentive to add lots of new things even if rarely used, with no drawback.
- with a search, one can localise all the parts of the code related to a given ingredient.

Examples:

```
if (has_fld == TRUE) {...}
if (has_cmb_lensing == TRUE) {...}
```

**11. Adding new ingredient...**

... can be done by searching for occurrence of another similar ingredient, copy/pasting, and adapting the new lines.

Example: if you want to add a new Dark Energy component, you may search for '_fld', duplicate all corresponding lines, change '_fld' into e.g. '_myde', and adapt the physical equations.

## 14 general coding principles of class

### 12. Error management

Our goal: CLASS should never crash. In case of problem, it returns an error message, with a well-documented error (line, function, what caused the crash, how to avoid it). Most of this message is generated automatically by the code.

If you get a crash: probably due to your own modifications. If you make the public CLASS crash, please write to us!

Run `./class input.ini` with an input file containing only

```
omega_b = 0.07
```

## 14 general coding principles of `class`

### 12. Error management

Our goal: CLASS should never crash. In case of problem, it returns an error message, with a well-documented error (line, function, what caused the crash, how to avoid it). Most of this message is generated automatically by the code.

Run `./class input.ini` with only `omega_b = 0.07`
You get an informative error message:

```
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 = 7.350000e-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
```

In the relevant part of the code, we only wrote the piece starting with "`You have asked...`". All the rest was generated automatically by the error management system. This follows from following everywhere some systematic rules and using specif macros for calling functions; see e.g.:
`Tokyo_Tools/CLASS_Lecture_Slides/lecture5_index_and_error.pdf`

## 14 general coding principles of class

**13. Version history**

Old versions can always be downloaded (from both http://class-code.net and https://github.com/lesgourg/class_public/).
In most cases, new versions feature new ingredients and avoid (whenever possible) to modify or erase the old ones, in such a way that modifications to an old version can still be pasted in a new version (as much as possible).

## 14 general coding principles of `class`

**13. Version history**

Old versions can always be downloaded (from both `http://class-code.net` and `https://github.com/lesgourg/class_public/`).
In most cases, new versions feature new ingredients and avoid (whenever possible) to modify or erase the old ones, in such a way that modifications to an old version can still be pasted in a new version (as much as possible).

**14. Git repository and GitHub website.**

The code can be downloaded as a `.tar.gz`, or as a git repository. Then, user can develop his own modification with the advantages of git (branching, memory of changes...); or merge his changes with a newer version almost automatically; or submit his modifications to the CLASS team in view of an easy merging with the public version.
You can also raise issues there, like in a discussion forum (will be propagated to the main CLASS developers, answering time variable)

More details on GitHub in `Tokyo_Tools/General_Lecture_Slides/git.pdf`

## Basic running

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

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

Syntax inside input file:

```
h = 0.7
T_cmb = 2.726 # comment
output = tCl, pCl
more comments, ignored because there is no equal sign
# comment with an =, still ignored thanks to the sharp
```

## Basic running

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

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

Syntax inside input file:

```
h = 0.7
T_cmb = 2.726 # comment
output = tCl, pCl
more comments, ignored because there is no equal sign
# comment with an =, still ignored thanks to the sharp
```

- Order of lines doesn't matter at all.
- All parameters not passed fixed to default, i.e. the most reasonable or minimalistic choice
- All possible input parameters and details on the syntax explained in `explanatory.ini`
- This is only a reference file; we advise you *never* to modify it, but rather to copy it and reduce it to a shorter and more friendly file.
- For *basic* usage: `explanatory.ini` ≡ full documentation of the code

## Basic running

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

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

Syntax inside input file:

```
h = 0.7
T_cmb = 2.726 # comment
output = tCl, pCl
more comments, ignored because there is no equal sign
# comment with an =, still ignored thanks to the sharp
```

- Order of lines doesn't matter at all.
- All parameters not passed fixed to default, i.e. the most reasonable or minimalistic choice
- All possible input parameters and details on the syntax explained in `explanatory.ini`
- This is only a reference file; we advise you *never* to modify it, but rather to copy it and reduce it to a shorter and more friendly file.
- For *basic* usage: `explanatory.ini` ≡ full documentation of the code
- `./class` can take two input files `*.ini` and `*.pre`:

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

  But one is enough.

## Basic running

For instance, we can create a very short file lcdm.ini:

```
****************************
* CLASS input parameter file *
****************************
----> background parameters:
H0 = 72.     # km/s/Mpc
omega_b = 0.0266691
omega_cdm = 0.110616
----> thermodynamics parameters:
z_reio = 10.
----> define primordial perturbation spectra:
A_s = 2.3e-9
n_s = 1.
----> define which perturbations should be computed:
output = tCl, pCl, lCl     # temp., polar., CMB lensing Cl's
lensing = yes
----> parameters for the output spectra:
l_scalar_max = 2500
```

## Basic running

Try to run the code with an even smaller input file **nut.ini**:

```
output = tCl
output_verbose=1
```

Run with

```
./class nut.ini
```

Check that $C_l$'s have been written in `output/nut00_cl.dat`

```
more output/nut00_cl.dat
```

Check if you run once more, the output goes automatically to `output/nut01_cl.dat` and so on...

## Basic running

Essential input parameters controlling the output (1/2); details in explanatory.ini:

```
modes = s,t
ic = ad, cdi, bi, nid, niv
lensing = yes
non linear = halofit
output = tCl, pCl, lCl, mPk, mTk, vTk, nCl, sCl

l_max_scalars=2500
l_max_tensors=500
l_max_lss = 1000
P_k_max_h/Mpc = 0.2
#P_k_max_1/Mpc =
z_pk = 0 #or 1,2,10

root = output/test_ #default: output/<ini_file>##_

headers = [yes/no]
format = [class/camb]
                                                       ...
```

## Basic running

Essential input parameters controlling the output (2/2); details in `explanatory.ini`:

```
write background = [yes/no]
write thermodynamics = [yes/no]
k_output_values = 0.01, 0.1, 0.0001  # in 1/Mpc
write primordial = [yes/no]

write parameters = [yes/no]
write warnings = [yes/no]

input_verbose = 1   # or 0, 2, 3,...
background_verbose = 1
thermodynamics_verbose = 1
perturbations_verbose = 1
transfer_verbose = 1
primordial_verbose = 1
spectra_verbose = 1
nonlinear_verbose = 1
lensing_verbose = 1
output_verbose = 1
```

## Basic running

Following files created (or not) automatically (here we assume that root=test_), depending on the content of the input fields:

```
output= [tCl,pCl,lCl,nCl,sCl,mPk,dTk,vTk]
lensing= [yes, no]
modes = [s,v,t]
ic = [ad, bi, cdi, nid, niv]
# ... and more, e.g. for nCl, sCl redshift bins
```

For all output in harmonic space ($C_l$'s), including CMB, density, lensing potential/cosmic shear:

- test_cl.dat total unlensed $C_l$'s

## Basic running

Following files created (or not) automatically (here we assume that root=test_), depending on the content of the input fields:

```
output= [tCl,pCl,lCl,nCl,sCl,mPk,dTk,vTk]
lensing= [yes, no]
modes = [s,v,t]
ic = [ad, bi, cdi, nid, niv]
# ... and more, e.g. for nCl, sCl redshift bins
```

For all output in harmonic space ($C_l$'s), including CMB, density, lensing potential/cosmic shear:

- test_cl.dat total unlensed $C_l$'s
- test_cl_lensed.dat total lensed $C_l$'s

## Basic running

Following files created (or not) automatically (here we assume that `root=test_`), depending on the content of the input fields:

```
output= [tCl,pCl,lCl,nCl,sCl,mPk,dTk,vTk]
lensing= [yes, no]
modes = [s,v,t]
ic = [ad, bi, cdi, nid, niv]
# ... and more, e.g. for nCl, sCl redshift bins
```

For all output in harmonic space ($C_l$'s), including CMB, density, lensing potential/cosmic shear:

- `test_cl.dat` total unlensed $C_l$'s
- `test_cl_lensed.dat` total lensed $C_l$'s
- `test_cls.dat` scalar $C_l$'s when two modes
- `test_clt.dat` tensor $C_l$'s when two modes

## Basic running

Following files created (or not) automatically (here we assume that root=test_), depending on the content of the input fields:

```
output= [tCl,pCl,lCl,nCl,sCl,mPk,dTk,vTk]
lensing= [yes, no]
modes = [s,v,t]
ic = [ad, bi, cdi, nid, niv]
# ... and more, e.g. for nCl, sCl redshift bins
```

For all output in harmonic space ($C_l$'s), including CMB, density, lensing potential/cosmic shear:

- test_cl.dat total unlensed $C_l$'s
- test_cl_lensed.dat total lensed $C_l$'s
- test_cls.dat scalar $C_l$'s when two modes
- test_clt.dat tensor $C_l$'s when two modes
- test_cl_ad.dat, test_cl_cdi.dat, test_cl_ad_cdi.dat etc. when different i.c. requested

## Basic running

Following files created (or not) automatically (here we assume that `root=test_`), depending on the content of the input fields:

```
output= [tCl,pCl,lCl,nCl,sCl,mPk,dTk,vTk]
lensing= [yes, no]
modes = [s,v,t]
ic = [ad, bi, cdi, nid, niv]
# ... and more, e.g. for nCl, sCl redshift bins
```

For all output in harmonic space ($C_l$'s), including CMB, density, lensing potential/cosmic shear:

- `test_cl.dat` total unlensed $C_l$'s
- `test_cl_lensed.dat` total lensed $C_l$'s
- `test_cls.dat` scalar $C_l$'s when two modes
- `test_clt.dat` tensor $C_l$'s when two modes
- `test_cl_ad.dat`, `test_cl_cdi.dat`, `test_cl_ad_cdi.dat` etc. when different i.c. requested

Number of columns in these files can vary a lot depending on input parameters. Always indicated in the header.

# Basic running

- Create an input file `test.ini` containing just
  ```
  output = tCl,lCl
  lensing = yes
  root = output/test1_
  ```
  Remember to go back to a new line at the end.
- Trick: add all the verbose parameters to it with:
  `tail explanatory.ini >> test.ini`
- Run with `./class test.ini`
- Look at the header of `output/test1_cl_lensed.dat`
- Repeat the test after changing to
  `root=output/test2_`, `output=tCl,pCl,lCl`
- Repeat the test after changing to
  `root=output/test3_` and adding `format = camb`
- Remove `format = camb` (or equivalently, set `format = class`) for the next exercises

## Basic running

Following files created (or not) automatically (here we assume that `root=test_`), depending on the content of the input fields:

```
output= [tCl,pCl,lCl,nCl,sCl,mPk,dTk,vTk]
non linear= [none, halofit]
modes = [s,v,t]
ic = [ad, bi, cdi, nid, niv]
z_pk = [list of values]
```

For all output in Fourier space:

- `test_pk.dat` matter power spectrum

## Basic running

Following files created (or not) automatically (here we assume that root=test_), depending on the content of the input fields:

```
output= [tCl,pCl,lCl,nCl,sCl,mPk,dTk,vTk]
non linear= [none, halofit]
modes = [s,v,t]
ic = [ad, bi, cdi, nid, niv]
z_pk = [list of values]
```

For all output in Fourier space:

- test_pk.dat matter power spectrum
- test_pk_nl.dat non-linear matter power spectrum

## Basic running

Following files created (or not) automatically (here we assume that root=test_), depending on the content of the input fields:

```
output= [tCl,pCl,lCl,nCl,sCl,mPk,dTk,vTk]
non linear= [none, halofit]
modes = [s,v,t]
ic = [ad, bi, cdi, nid, niv]
z_pk = [list of values]
```

For all output in Fourier space:

- test_pk.dat matter power spectrum
- test_pk_nl.dat non-linear matter power spectrum
- test_pk_ad.dat, test_pk_cdi.dat, test_pk_ad_cdi.dat etc. when different i.c. requested

## Basic running

Following files created (or not) automatically (here we assume that `root=test_`), depending on the content of the input fields:

```
output= [tCl,pCl,lCl,nCl,sCl,mPk,dTk,vTk]
non linear= [none, halofit]
modes = [s,v,t]
ic = [ad, bi, cdi, nid, niv]
z_pk = [list of values]
```

For all output in Fourier space:

- `test_pk.dat` matter power spectrum
- `test_pk_nl.dat` non-linear matter power spectrum
- `test_pk_ad.dat`, `test_pk_cdi.dat`, `test_pk_ad_cdi.dat` etc. when different i.c. requested
- `test_tk.dat` density and/or velocity transfer functions

## Basic running

Following files created (or not) automatically (here we assume that root=test_), depending on the content of the input fields:

```
output= [tCl,pCl,lCl,nCl,sCl,mPk,dTk,vTk]
non linear= [none, halofit]
modes = [s,v,t]
ic = [ad, bi, cdi, nid, niv]
z_pk = [list of values]
```

For all output in Fourier space:

- test_pk.dat matter power spectrum
- test_pk_nl.dat non-linear matter power spectrum
- test_pk_ad.dat, test_pk_cdi.dat, test_pk_ad_cdi.dat etc. when different i.c. requested
- test_tk.dat density and/or velocity transfer functions
- test_tk_ad.dat, test_tk_cdi.dat, test_tk_ad_cdi.dat etc. when different i.c. requested

## Basic running

Following files created (or not) automatically (here we assume that `root=test_`), depending on the content of the input fields:

```
output= [tCl,pCl,lCl,nCl,sCl,mPk,dTk,vTk]
non linear= [none, halofit]
modes = [s,v,t]
ic = [ad, bi, cdi, nid, niv]
z_pk = [list of values]
```

For all output in Fourier space:

- `test_pk.dat` matter power spectrum
- `test_pk_nl.dat` non-linear matter power spectrum
- `test_pk_ad.dat`, `test_pk_cdi.dat`, `test_pk_ad_cdi.dat` etc. when different i.c. requested
- `test_tk.dat` density and/or velocity transfer functions
- `test_tk_ad.dat`, `test_tk_cdi.dat`, `test_tk_ad_cdi.dat` etc. when different i.c. requested
- if `pk` or `tk` requested at different redshift, several files, with extra suffix `_z0`, `_z1`, etc.

## Basic running

Following files created (or not) automatically (here we assume that root=test_), depending on the content of the input fields:

```
output= [tCl,pCl,lCl,nCl,sCl,mPk,dTk,vTk]
non linear= [none, halofit]
modes = [s,v,t]
ic = [ad, bi, cdi, nid, niv]
z_pk = [list of values]
```

For all output in Fourier space:

- test_pk.dat matter power spectrum
- test_pk_nl.dat non-linear matter power spectrum
- test_pk_ad.dat, test_pk_cdi.dat, test_pk_ad_cdi.dat etc. when different i.c. requested
- test_tk.dat density and/or velocity transfer functions
- test_tk_ad.dat, test_tk_cdi.dat, test_tk_ad_cdi.dat etc. when different i.c. requested
- if pk or tk requested at different redshift, several files, with extra suffix _z0, _z1, etc.

Run with root=output/test4_, output=mPk,
and without or with the extra line z_pk=0,0.4,0.8.
In each case, look at output file names and headers.

## Basic running

Following files created (or not) automatically (here we assume that root=test_), depending on the content of the input fields:

```
write background = [yes,no]
write thermodynamics = [yes,no]
write primordial = [yes,no]
k_output_values = [list of values]
modes = [s,v,t]
```

- test_background.dat background quantities versus time and redshift

## Basic running

Following files created (or not) automatically (here we assume that `root=test_`), depending on the content of the input fields:

```
write background = [yes,no]
write thermodynamics = [yes,no]
write primordial = [yes,no]
k_output_values = [list of values]
modes = [s,v,t]
```

- `test_background.dat` background quantities versus time and redshift
- `test_thermodynamics.dat` thermodynamical quantities versus redshift

## Basic running

Following files created (or not) automatically (here we assume that `root=test_`), depending on the content of the input fields:

```
write background = [yes,no]
write thermodynamics = [yes,no]
write primordial = [yes,no]
k_output_values = [list of values]
modes = [s,v,t]
```

- `test_background.dat` background quantities versus time and redshift
- `test_thermodynamics.dat` thermodynamical quantities versus redshift
- `test_primordial.dat` primordial spectra (may follow from inflation simulation)

## Basic running

Following files created (or not) automatically (here we assume that root=test_), depending on the content of the input fields:

```
write background = [yes,no]
write thermodynamics = [yes,no]
write primordial = [yes,no]
k_output_values = [list of values]
modes = [s,v,t]
```

- test_background.dat background quantities versus time and redshift
- test_thermodynamics.dat thermodynamical quantities versus redshift
- test_primordial.dat primordial spectra (may follow from inflation simulation)
- test_perturbations_k*_s[vt].dat evolution of perturbations versus time

## Basic running

Following files created (or not) automatically (here we assume that root=test_), depending on the content of the input fields:

```
write background = [yes,no]
write thermodynamics = [yes,no]
write primordial = [yes,no]
k_output_values = [list of values]
modes = [s,v,t]
```

- test_background.dat background quantities versus time and redshift
- test_thermodynamics.dat thermodynamical quantities versus redshift
- test_primordial.dat primordial spectra (may follow from inflation simulation)
- test_perturbations_k*_s[vt].dat evolution of perturbations versus time

Run with

```
root=output/test5_
output=tCl
k_output_values = 0.001,0.01
write background = yes
```

Look at output file names and headers.

## Plotting

## Plotting

**You can get plots**

1. Manually: using e.g. `gnuplot`, `IDL`, `python`, `Mathematic`, `GNU Octave`...
2. Automatically: using `python` and script `CPU.py`, or `MATLAB` and script `plot_CLASS_output.m`
3. Interactively: using CLASS as a `python` module, within a `python` session or an `iPython Notebook`

## Plotting: Manual mode

**Using Gnuplot**

```
$> cd output/
$> gnuplot
gnuplot> plot 'test1_cl.dat' using 1:2 with lines title 'Cl'
```

or equivalently

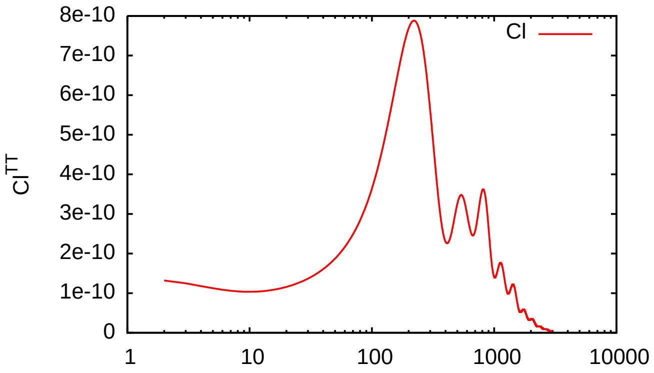
```
gnuplot> p 'test1_cl_lensing.dat' u 1:2 w l t 'Cl'
```

Some additional Gnuplot commands:

- Modifying scale and adding labels:
  ```
  gnuplot> set logscale x
  gnuplot> set ylabel 'Cl^{TT}'
  ```
- Writing to a file:
  ```
  gnuplot> set terminal pdf
  gnuplot> set out 'my plot.pdf'
  ```

## Plotting: Manual mode

Example of `gnuplot` output:



## Plotting: Manual mode

**Which column number?**

Easy! Column number and title written automatically in file.

Example for `test2_cl_lensed.dat` file:

```
1:l    2:TT    3:EE    4:TE    5:BB    6:phiphi    7:TPhi    8:Ephi
```

Example for a `*_background.dat` file:

```
1:z    2:proper time [Gyr]    3:conf. time [Mpc] ...
  4:H [1/Mpc]    5:comov. dist.    6:ang.diam.dist. ...
     7:lum. dist.    8:comov.snd.hrz.    9:(.)rho_g ...
```

Example for a `*_thermodynamics.dat` file:

```
1:z    2:conf. time [Mpc]    3:x_e    4:kappa' [Mpc^-1] ...
  5:exp(-kappa)    6:g [Mpc^-1]    7:Tb [K]    8:c_b^2    9:tau_d
```

etc.

**Other tools**

Alternatives include MATLAB, Python, IDL, Mathematica, GNU Octave, . . .

## Plotting: Automatic mode

**CPU.py**

- CPU: CLASS Plotting Utility
- Written in Python by Benjamin Audren

**plot_CLASS_output.m**

- Writen in MATLAB by Thomas Tram
- Compatible* with GNU Octave

## Plotting: Automatic mode

CPU.py

Getting help:

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

Plot certain quantities:

```
python CPU.py output/test5_background.dat -y rho_g rho_cdm
```

Plot quantities with a common string across several files:

```
python CPU.py output/test1_cl.dat output/test2_cl.dat -y T
```

Set scale: {`lin,loglog,loglin,george`}

```
python CPU.py ... --scale loglog
```

Set axis limits:

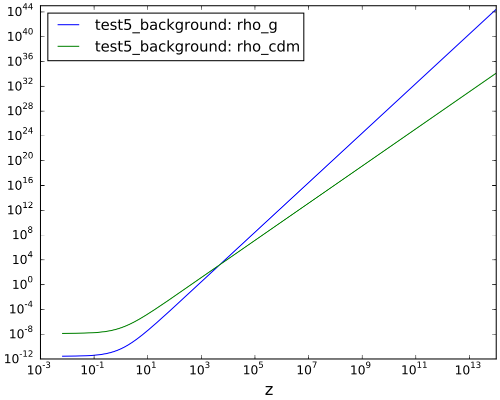
```
python CPU.py ... --xlim 0.1 10 --ylim 1e2 1e5
```

Save plot to PDF file:

```
python CPU.py ... -p
```

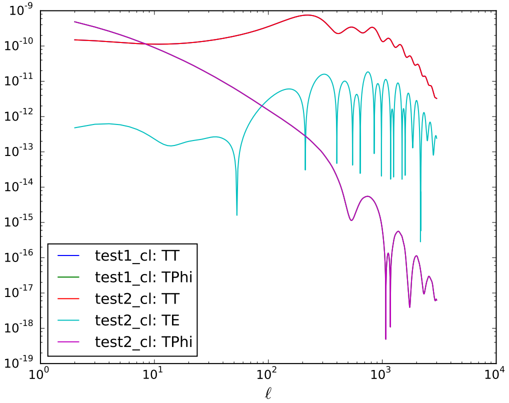
## Plotting: Automatic mode

```
$> python CPU.py output/test5_background.dat
  -y rho_g rho_cdm --scale loglog -p example_background.pdf
```



## Plotting: Automatic mode

```
$> python CPU.py output/test1_cl.dat output/test2_cl.dat
  -y T --scale loglog -p example_T.pdf
```



## Plotting: Automatic mode

plot_CLASS_output.m

Getting help:

```
  help plot_CLASS_output
```

Plot certain quantities:

```
  plot_C('test5_background.dat',{'rho_g','rho_cdm'})
```

Plot quantities with a common string across several files:

```
  plot_C({'test1_cl.dat', 'test2_cl.dat'},'T')
```

Set xscale and yscale:

```
  plot_C(...,...,'xscale','log',yscale','log')
```

Set axis limits:

```
  plot_C(...,...,'xlim',[0.1, 10])
```

Specify name of EPS file:

```
  plot_C(...,...,'EpsFilename','myplot.eps')
```

## Plotting: Interactive mode

Let's look first at the python wrapper! After, you'll see that you can

- use CLASS as a Python module
- call most useful CLASS functions directly from Python
- get the output stored directly in Python variables
- plot whatever you want with the usual functions of matplotlib, pyplot ... (and also perform algebra on the output with numpy, etc)

## Python wrapper

**What happens when you type `make` (and not just `make class`)?**

The complier:

- creates the C executable `class`
- creates a C library `libclass.a`
- executes a code written in **Cython**, `python/classy.pyx`, which reads `libclass.a`, produces a python module called `classy.py`, and installs it on your computer

## Python wrapper

**What happens when you type `make` (and not just `make class`)?**

The complier:

- creates the C executable `class`
- creates a C library `libclass.a`
- executes a code written in **Cython**, `python/classy.pyx`, which reads `libclass.a`, produces a python module called `classy.py`, and installs it on your computer

**classy, the CLASS wrapper**

- Written in **Cython**.
- Started by Karim Benabed, mainly developed by Benjamin Audren
- Needed for Monte Python and when using CLASS from **Python**.
- the developers need to manually implement in `class.pyx` some lines for any variable or function of CLASS that should be known by the python module.
- ideas for making this step automatic and systematic in the future...

## Python wrapper

**classy, the CLASS wrapper**

- **classy** is the name of the **Python module** "containing the code CLASS"
- **Class** is the name of the **Python class** "containing the code CLASS" (it is defined by **classy**)
- so, before doing anything, we need to start from:
  `python> from classy import Class`

Running CLASS from Python:

```
from classy import Class
import numpy as np
import matplotlib.pyplot as plt
```

```
cosmo = Class()
cosmo.set({'output':'tCl,pCl,lCl','lensing':'yes'})
cosmo.compute()
```

The name cosmo in this exemple is optional. It is just one instance of the class `Class` containing a given cosmology (= a set of input parameters and of output quantitites). You can work with several at the same time, e.g.: `lcdm = Class()` and `wcdm = Class()`

## Python wrapper

Let's do a simple plot of the lensed $C_l^{TT}$, $C_l^{EE}$ for $\Lambda$CDM:

## Python wrapper

Let's do a simple plot of the lensed $C_l^{TT}$, $C_l^{EE}$ for $\Lambda$CDM:

```
from classy import Class
import numpy as np
import matplotlib.pyplot as plt
cosmo = Class()
cosmo.set({'output':'tCl,pCl,lCl','lensing':'yes'})
cosmo.compute()
l = np.array(range(2,2501))
factor = l*(l+1)/(2*np.pi)
lensed_cl = cosmo.lensed_cl(2500)
```

## Python wrapper

Let's do a simple plot of the lensed $C_l^{TT}$, $C_l^{EE}$ for $\Lambda$CDM:

```
from classy import Class
import numpy as np
import matplotlib.pyplot as plt
cosmo = Class()
cosmo.set({'output':'tCl,pCl,lCl','lensing':'yes'})
cosmo.compute()
l = np.array(range(2,2501))
factor = l*(l+1)/(2*np.pi)
lensed_cl = cosmo.lensed_cl(2500)
```

We may want to check which "columns" are stored in the array `lensed_cl` (= in fact a python dictionnary):

```
lensed_cl.viewkeys()
```

## Python wrapper

Let's do a simple plot of the lensed $C_l^{TT}$, $C_l^{EE}$ for $\Lambda$CDM:

```
from classy import Class
import numpy as np
import matplotlib.pyplot as plt
cosmo = Class()
cosmo.set({'output':'tCl,pCl,lCl','lensing':'yes'})
cosmo.compute()
l = np.array(range(2,2501))
factor = l*(l+1)/(2*np.pi)
lensed_cl = cosmo.lensed_cl(2500)
```

We may want to check which "columns" are stored in the array `lensed_cl` (= in fact a python dictionnary):

```
lensed_cl.viewkeys()
```

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

Here `t,e,b,p` mean respectively temp., E and B pol., CMB lensing potential "phi".

## Python wrapper

Let's do a simple plot of the lensed $C_l^{TT}$, $C_l^{EE}$ for $\Lambda$CDM:

```
from classy import Class
import numpy as np
import matplotlib.pyplot as plt
cosmo = Class()
cosmo.set({'output':'tCl,pCl,lCl','lensing':'yes'})
cosmo.compute()
l = np.array(range(2,2501))
factor = l*(l+1)/(2*np.pi)
lensed_cl = cosmo.lensed_cl(2500)
```

We may want to check which "columns" are stored in the array `lensed_cl` (= in fact a python dictionnary):

```
lensed_cl.viewkeys()
```

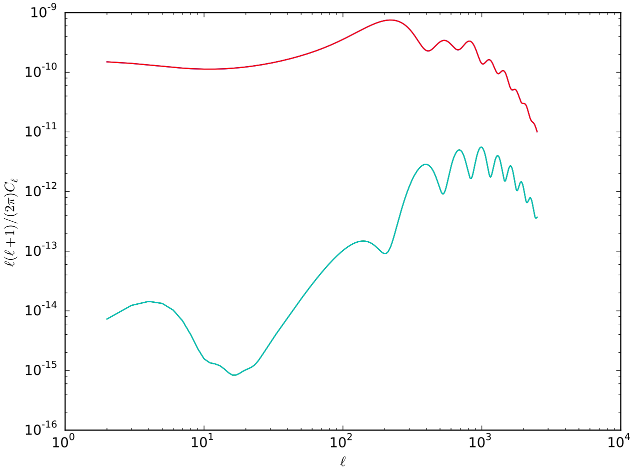
```
dict_keys(['pp', 'ell', 'bb', 'ee', 'tt', 'tp', 'te'])
```

Here `t`,`e`,`b`,`p` mean respectively temp., E and B pol., CMB lensing potential "phi".
Final plotting steps:

```
plt.loglog(l,factor*lensed_cl['tt'][2:],l,factor*lensed_cl['
    ee'][2:])
plt.xlabel(r"$\ell$")
plt.ylabel(r"$\ell(\ell+1)/(2\pi) C_\ell$")
plt.tight_layout()
plt.savefig("output/TT_EE_LambdaCDM.pdf")
```

## Python wrapper

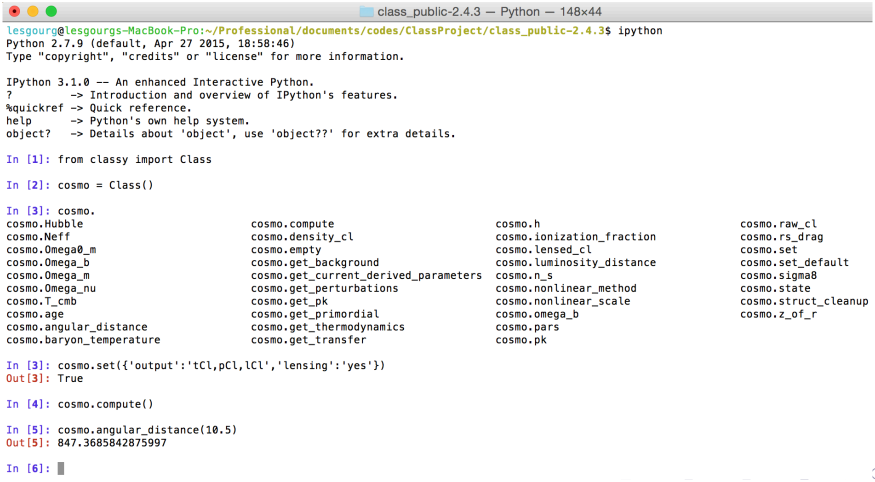
The following figure has been produced:



# Python wrapper: IPython

**IPython**

`ipython` is nicer than `python`, for instance because of automatic completion with TAB key!



```
lesgourg@lesgourgs-MacBook-Pro:~/Professional/documents/codes/ClassProject/class_public-2.4.3$ ipython
Python 2.7.9 (default, Apr 27 2015, 18:58:46)
Type "copyright", "credits" or "license" for more information.

IPython 3.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: from classy import Class

In [2]: cosmo = Class()

In [3]: cosmo.
cosmo.Hubble                      cosmo.compute                     cosmo.h                           cosmo.raw_cl
cosmo.Neff                        cosmo.density_cl                  cosmo.ionization_fraction         cosmo.rs_drag
cosmo.Omega0_m                    cosmo.empty                       cosmo.lensed_cl                   cosmo.set
cosmo.Omega_b                     cosmo.get_background              cosmo.luminosity_distance         cosmo.set_default
cosmo.Omega_m                     cosmo.get_current_derived_parameters cosmo.n_s                      cosmo.sigma8
cosmo.Omega_nu                    cosmo.get_perturbations           cosmo.nonlinear_method            cosmo.state
cosmo.T_cmb                       cosmo.get_pk                      cosmo.nonlinear_scale             cosmo.struct_cleanup
cosmo.age                         cosmo.get_primordial              cosmo.omega_b                     cosmo.z_of_r
cosmo.angular_distance            cosmo.get_thermodynamics          cosmo.pars
cosmo.baryon_temperature          cosmo.get_transfer                cosmo.pk

In [3]: cosmo.set({'output':'tCl,pCl,lCl','lensing':'yes'})
Out[3]: True

In [4]: cosmo.compute()

In [5]: cosmo.angular_distance(10.5)
Out[5]: 847.3685842875997

In [6]: 
```

## Python wrapper: IPython notebooks

### IPython Notebook

This is by far the most convenient way to work!

- IPython Notebook is a Mathematica style (cell) interface to IPython.
- Has TAB completion of variables and function names.
- Nicely presents the documentation of each function.
- Easy way to get started on Python and to use all functionalities of `classy` .

## Python wrapper: IPython notebooks

### IPython Notebook

This is by far the most convenient way to work!

- IPython Notebook is a Mathematica style (cell) interface to IPython.
- Has TAB completion of variables and function names.
- Nicely presents the documentation of each function.
- Easy way to get started on Python and to use all functionalities of `classy` .
- Your keep track of all your working session, which includes commands and plots.
- You can come back to it or modify it in the future.
- You can load sessions from other people.

# Python wrapper: IPython notebooks

## IPython Notebook

This is by far the most convenient way to work!

- IPython Notebook is a Mathematica style (cell) interface to IPython.
- Has TAB completion of variables and function names.
- Nicely presents the documentation of each function.
- Easy way to get started on Python and to use all functionalities of `classy` .
- Your keep track of all your working session, which includes commands and plots.
- You can come back to it or modify it in the future.
- You can load sessions from other people.
- We are setting an docker (on-line repositoryof session examples) thanks to the `mybinder.org` project.
- These example sessions can even be executed on-line, when this is more convenient than downloading the notebook files.

## Python wrapper: IPython notebooks

**IPython Notebook**

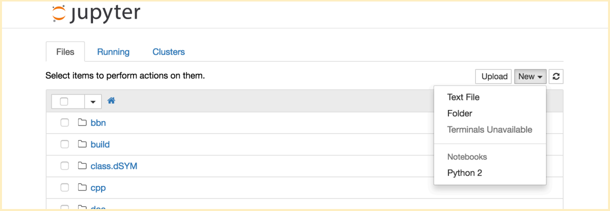
This is by far the most convenient way to work!

- IPython Notebook is a Mathematica style (cell) interface to IPython.
- Has TAB completion of variables and function names.
- Nicely presents the documentation of each function.
- Easy way to get started on Python and to use all functionalities of `classy` .
- Your keep track of all your working session, which includes commands and plots.
- You can come back to it or modify it in the future.
- You can load sessions from other people.
- We are setting an docker (on-line repositoryof session examples) thanks to the `mybinder.org` project.
- These example sessions can even be executed on-line, when this is more convenient than downloading the notebook files.
- Almost all your research (computing things, producing plots for papers) could be done within notebooks and calling `classy` functions ☺

# Python wrapper: IPython notebooks

## Launching IPython Notebook

- To launch a new notebook:
  `ipython notebook`
  This opens a new window on browser. Then click “new” and “Python 2”

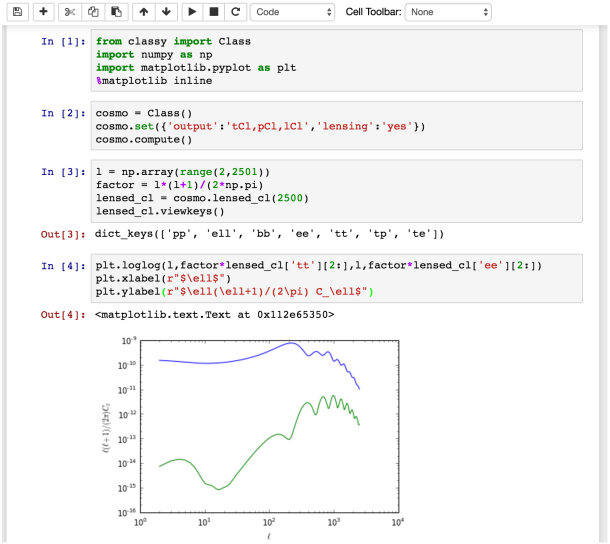


Select items to perform actions on them.

- To launch an existing one (your previous session, something taken from the net...):
  `ipython notebook my_notebook.ipynb`
- To use the docker (our online notebook server):
  https://github.com/ThomasTram/iCLASS
  Just choose one notebook. Then you can either run it directly on-line, or download it.

# Python wrapper: IPython notebooks

Same example as before, within a notebook:

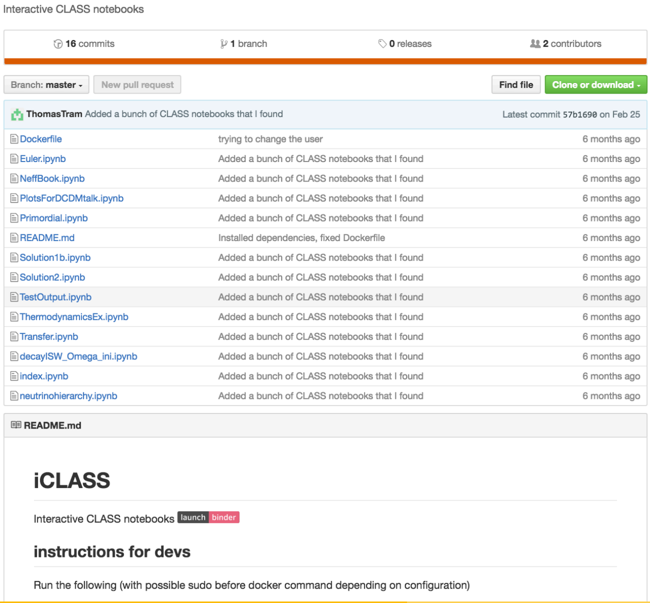


```
In [1]: from classy import Class
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

In [2]: cosmo = Class()
        cosmo.set({'output':'tCl,pCl,lCl','lensing':'yes'})
        cosmo.compute()

In [3]: l = np.array(range(2,2501))
        factor = l*(l+1)/(2*np.pi)
        lensed_cl = cosmo.lensed_cl(2500)
        lensed_cl.viewkeys()

Out[3]: dict_keys(['pp', 'ell', 'bb', 'ee', 'tt', 'tp', 'te'])

In [4]: plt.loglog(l,factor*lensed_cl['tt'][2:],l,factor*lensed_cl['ee'][2:])
        plt.xlabel(r"$\ell$")
        plt.ylabel(r"$\ell(\ell+1)/(2\pi) C_\ell$")

Out[4]: <matplotlib.text.Text at 0x112e65350>
```

- note the additional line `%matplotlib inline` (necessary to visualise figure inside notebook)
- you can save the figure separately in a file

# Python wrapper: IPython notebooks

Current aspect of the docker (we will improve the presentation!):



Interactive CLASS notebooks

ThomasTram Added a bunch of CLASS notebooks that I found — Latest commit 57b1690 on Feb 25

| | | |
|---|---|---|
| Dockerfile | trying to change the user | 6 months ago |
| Euler.ipynb | Added a bunch of CLASS notebooks that I found | 6 months ago |
| NeffBook.ipynb | Added a bunch of CLASS notebooks that I found | 6 months ago |
| PlotsForDCDMtalk.ipynb | Added a bunch of CLASS notebooks that I found | 6 months ago |
| Primordial.ipynb | Added a bunch of CLASS notebooks that I found | 6 months ago |
| README.md | Installed dependencies, fixed Dockerfile | 6 months ago |
| Solution1b.ipynb | Added a bunch of CLASS notebooks that I found | 6 months ago |
| Solution2.ipynb | Added a bunch of CLASS notebooks that I found | 6 months ago |
| TestOutput.ipynb | Added a bunch of CLASS notebooks that I found | 6 months ago |
| ThermodynamicsEx.ipynb | Added a bunch of CLASS notebooks that I found | 6 months ago |
| Transfer.ipynb | Added a bunch of CLASS notebooks that I found | 6 months ago |
| decayISW_Omega_ini.ipynb | Added a bunch of CLASS notebooks that I found | 6 months ago |
| index.ipynb | Added a bunch of CLASS notebooks that I found | 6 months ago |
| neutrinohierarchy.ipynb | Added a bunch of CLASS notebooks that I found | 6 months ago |

README.md

iCLASS

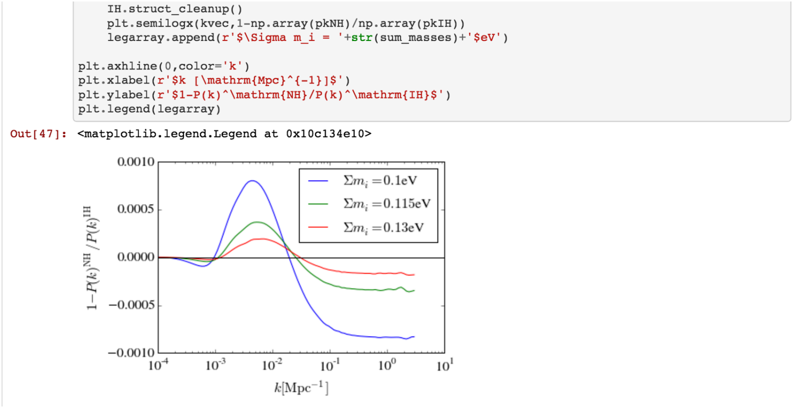
Interactive CLASS notebooks launch binder

instructions for devs

Run the following (with possible sudo before docker command depending on configuration)

## Python wrapper: IPython notebooks

For instance, `neutrinohierarchy.ipynb` shows how to plot the ratio of $P(k)$ in the normal/inverted hierarchy model for a fixed value of the total neutrino mass, given neutrino oscillation data:

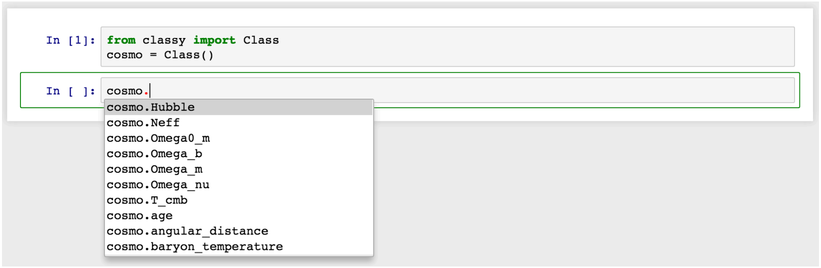


```
    IH.struct_cleanup()
    plt.semilogx(kvec,1-np.array(pkNH)/np.array(pkIH))
    legarray.append(r'$\Sigma m_i = '+str(sum_masses)+'$eV')

plt.axhline(0,color='k')
plt.xlabel(r'$k [\mathrm{Mpc}^{-1}]$')
plt.ylabel(r'$1-P(k)^\mathrm{NH}/P(k)^\mathrm{IH}$')
plt.legend(legarray)
```

```
Out[47]: <matplotlib.legend.Legend at 0x10c134e10>
```

## Python wrapper: IPython notebooks

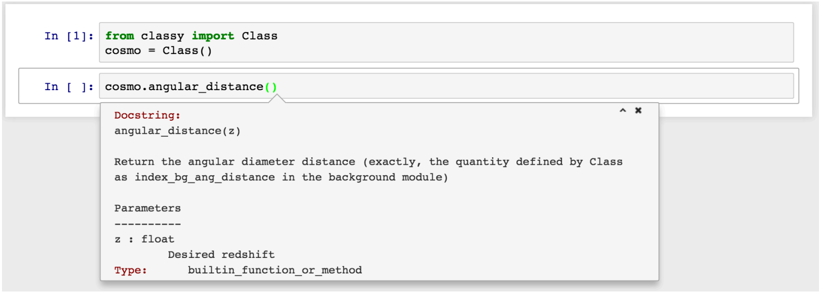
The TAB key after the dot gives you the list of available classy methods (= available functions and quantities) in a scrolling menu:



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

In [ ]: cosmo.
```

## Python wrapper: IPython notebooks

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

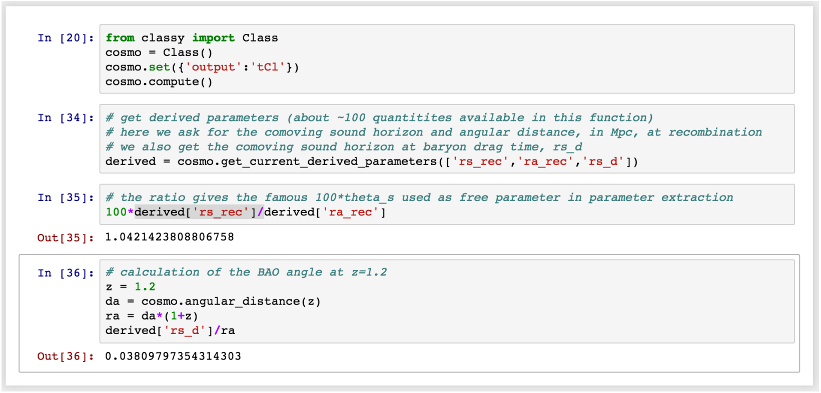


```
In [1]: from classy import Class
        cosmo = Class()

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
```

## Python wrapper: IPython notebooks

Example for getting derived quantities, such as the usual $100\,\theta_s$ (related to angular scale of the CMB peaks), and the BAO angle at redshift $z = 1.2$:



```
In [20]: from classy import Class
         cosmo = Class()
         cosmo.set({'output':'tCl'})
         cosmo.compute()

In [34]: # get derived parameters (about ~100 quantitites available in this function)
         # here we ask for the comoving sound horizon and angular distance, in Mpc, at recombination
         # we also get the comoving sound horizon at baryon drag time, rs_d
         derived = cosmo.get_current_derived_parameters(['rs_rec','ra_rec','rs_d'])

In [35]: # the ratio gives the famous 100*theta_s used as free parameter in parameter extraction
         100*derived['rs_rec']/derived['ra_rec']

Out[35]: 1.0421423808806758

In [36]: # calculation of the BAO angle at z=1.2
         z = 1.2
         da = cosmo.angular_distance(z)
         ra = da*(1+z)
         derived['rs_d']/ra

Out[36]: 0.03809797354314303
```

## This is the end...

Now you can train more with exercises 0, 1a, 1b, 1c, 1d of the Tokyo Dropbox:

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

They are in:
Tokyo_Tools/CLASS_Exercises/exercises.pdf

You can first try first the "old-school" way, using the terminal, output files, plotting scripts: detailed correction in:
Tokyo_Tools/CLASS_Exercises/exercises_correction.pdf

Or try directly with the notebook, partial correction in:
Tokyo_Tools/IPython_Notebooks
and also in the docker.

You can contact me at lesgourg@physik.rwth-aachen.de, or Thomas at thomas.tram@port.ac.uk, or raise issues on GitHub!