CLASS

the Cosmological Linear Anisotropy Solving System[1]

Julien Lesgourgues, Deanna C. Hooper
TTK, RWTH Aachen University

Kavli Institute for Cosmology, Cambridge, 11-13.09.2018

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

class in Cambridge

- Lecture 1: A few basics about class
- Lecture 2: How to run it from python scripts / jupyter notebooks
- Lecture 3: Guidelines for modifying / developing the code

Context

class is the 5th public Einstein-Boltzmann solver covering all basic cosmology:

1. COSMICS package in f77 (Bertschinger 1995)
   Basic equations, brute-force C_l^{TT}
2. CMBFAST in f77 (Seljak & Zaldarriaga 1996)
   Line-of-sight, $C_l^{EE,TE,BB}$, open universe, CMB lensing
3. CAMB in f90/2000 (Lewis & Challinor 1999)
   closed universe, better lensing, new algorithms, new approximations, new species, new observables... (http://camb.info)
4. CMBEASY in C++ (Doran 2003)
5. class in C (Lesgourgues & Tram 2011)
   simpler polarisation equations, new algorithms, new approximations, new species, new observables... (http://class-code.net)

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

Context

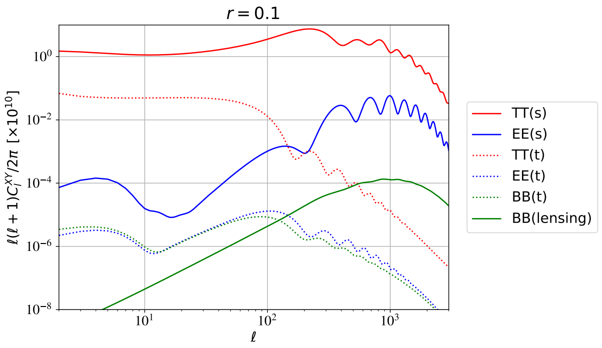
Project started on request of Planck science team, in order to have a tool independent from CAMB, and check for possible Boltzmann-code-induced bias in parameter extraction. The `class`-CAMB comparison has triggered progress in the accuracy of both codes. Agreement established at 10^{-4} (0.01%) level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

`class` aims at being:

- general (more models, more output/observables)
- modern (structured, modular, flexible, wrap-able: wrapper for python, C++, automatic precision test code)
- user friendly (documented, structured, easy to understand) and hence easier to modify (coding additional models/observables)
- accurate and fast (currently comparable to CAMB; in principle, clear structure offers potential for further optimisation/parallelisation/vectorisation/etc.)

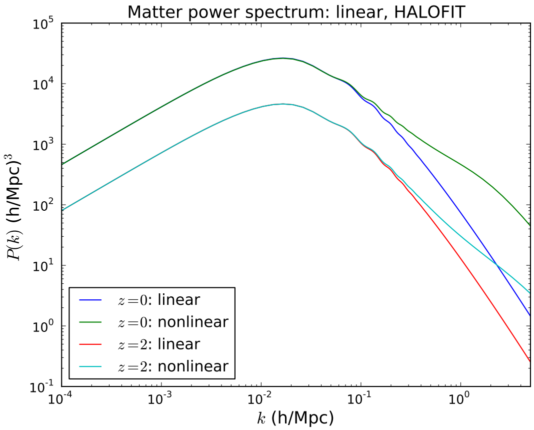
With class you can get:

The CMB anisotropy spectra:



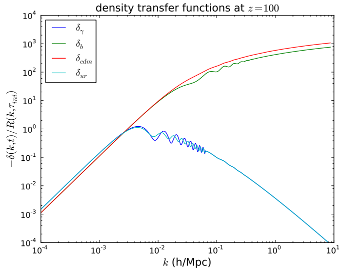
With class you can get:

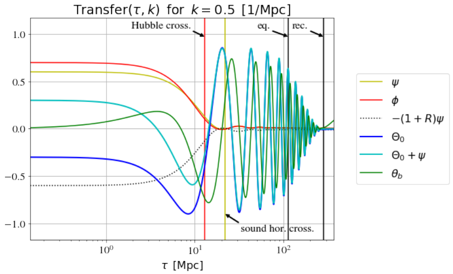
The matter power spectrum:



With class you can get:

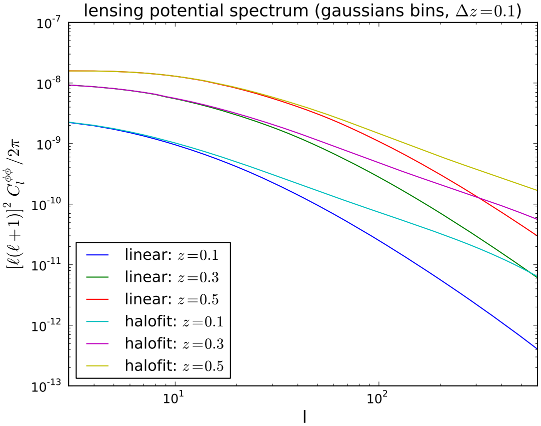
The transfer functions at a given time/redshift (e.g. initial conditions for N-body):





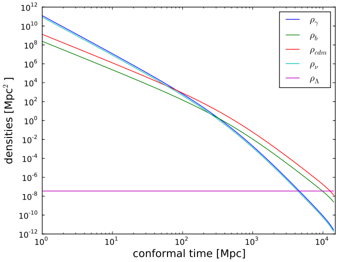
With `class` you can get:

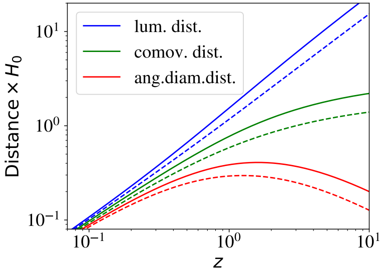
The matter density (number count) C_l's, or the lensing C_l's (with arbitrary selection/window functions):



With `class` you can get:

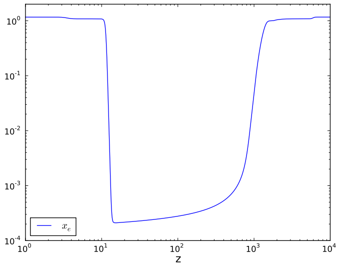
The background evolution in a given cosmological model:

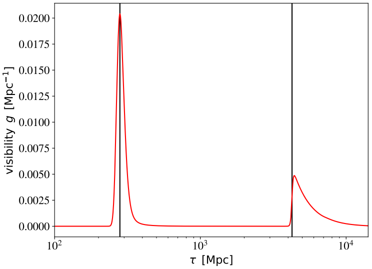




With `class` you can get:

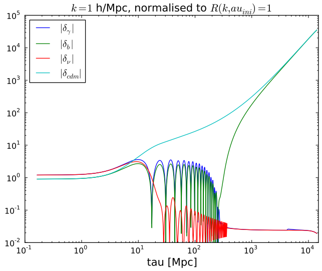
The thermal history in a given cosmological model:

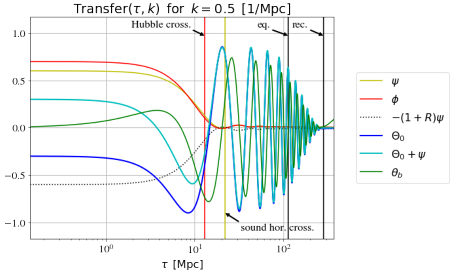




With class you can get:

The time evolution of perturbations for individual Fourier modes:





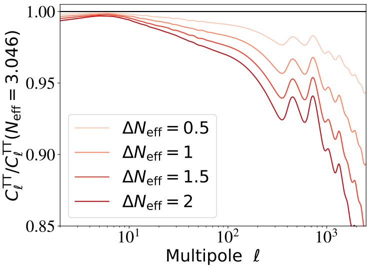
With `class` you can get:

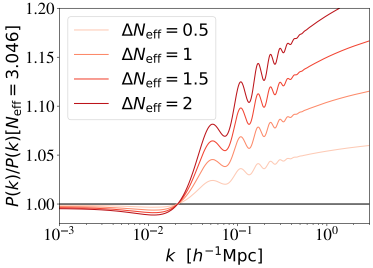
... and several other quantities, for instance:

- 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:

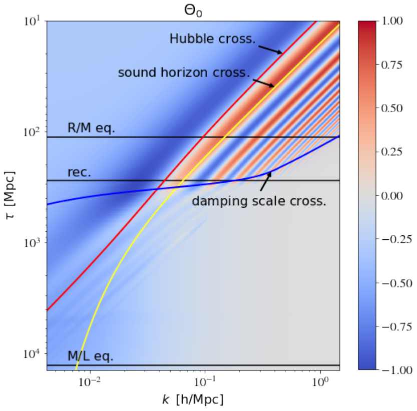
... if you use class as a python module you can extract all kind of output or intermediate quantities, manipulate them in various ways, and make all kinds of computations or nice plots:

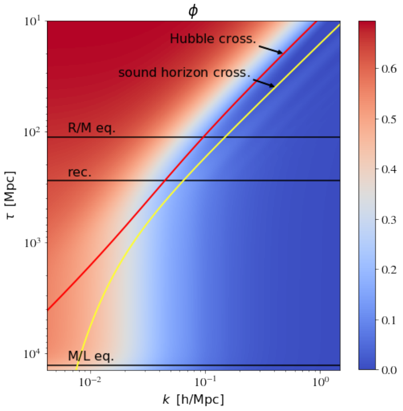




With class you can get:

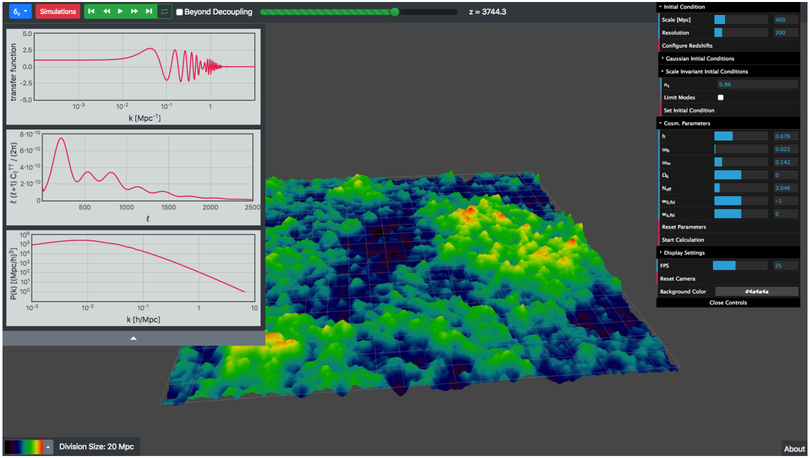
... if you use class as a python module you can extract all kind of output or intermediate quantities, manipulate them in various ways, and make all kinds of computations or nice plots:





With `class` you can get:

… and movies of CMB perturbations in 2D slices of early universe with our Real space graphical interface (just released in v2.7.0); here is a snapshot:



Initial Condition	
Scale [Mpc]	400
Resolution	200
Configure Redshifts	
Gaussian Initial Conditions	
Scale Invariant Initial Conditions	
n_s	0.96
Limit Modes	☐
Set Initial Condition	
Cosm. Parameters	
h	0.676
ω_b	0.022
ω_m	0.142
Ω_k	0
N_{eff}	3.046
$w_{0,fld}$	-1
$w_{a,fld}$	0
Reset Parameters	
Start Calculation	
Display Settings	
FPS	25
Reset Camera	
Background Color	#4a4a4a
Close Controls	

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, annihilating, *decaying, interacting*…),
- Dark Energy (fluid with flexible $w(a)$ + 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`)
- multi-gauge (synchronous, newtonian…)
- extension to second-order perturbation theory: SONG (Fidler, Pettinari, Tram, `https://github.com/coccoinomane/song`)
- interfacing with particle physics modules and codes for exotic energy injection available in ExoCLASS branch of `http://github.com/lesgourg/class_public.git` (Stöcker, Poulin)

class coding spirit

Equations follow literally notations of most famous papers

(in particular Ma & Bertschinger 1996, `astro-ph/9506072`).
Multi-gauge code: everything coded in newtonian and synchronous gauge, structure ready for more gauges.

class coding spirit

Equations follow literally notations of most famous papers

(in particular Ma & Bertschinger 1996, `astro-ph/9506072`).
Multi-gauge code: everything coded in newtonian and synchronous gauge, structure ready for more gauges.

Input parameters interpreted and processed into final form needed by the modules

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 $\{T_{\rm cmb},\ \Omega_\gamma,\ \omega_\gamma\}$, $\{\Omega_{\rm ncdm},\ \omega_{\rm ncdm},\ m_\nu\}$, $\{\Omega_{\rm ur},\ \omega_{\rm ur},\ N_{\rm ur}\}$,...

class coding spirit

Equations follow literally notations of most famous papers

(in particular Ma & Bertschinger 1996, astro-ph/9506072).
Multi-gauge code: everything coded in newtonian and synchronous gauge, structure ready for more gauges.

Input parameters interpreted and processed into final form needed by the modules

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 $\{T_{\rm cmb},\, \Omega_\gamma,\, \omega_\gamma\}$, $\{\Omega_{\rm ncdm},\, \omega_{\rm ncdm},\, m_\nu\}$, $\{\Omega_{\rm ur},\, \omega_{\rm ur},\, N_{\rm ur}\}$,...

Homogeneous units

Inside all modules except thermodynamics: everything in Mpc^n.
Examples:
- conformal time τ in Mpc, $H = \frac{a'}{a^2}$ in Mpc^{-1}
- $\rho_{\rm class} \equiv \frac{8\pi G}{3}\rho_{\rm physical}$ in Mpc^{-2}, such that $H^2 = \sum_i \rho_i$
- k in Mpc^{-1}, $P(k)$ in Mpc^3

class coding spirit

Accessible and self-contained

Plain C (for performance and readability) but mimicking features of C++ (see later).
No external libraries for a quick installation (but parallelisation requires OpenMP).
Lots of comments in the code, plus automatic `doxygen` documentation

class coding spirit

Accessible and self-contained

Plain C (for performance and readability) but mimicking features of C++ (see later).
No external libraries for a quick installation (but parallelisation requires OpenMP).
Lots of comments in the code, plus automatic `doxygen` documentation

Structured and flexible

Sequence of ten modules with distinct physical tasks, no duplicate equations.

`class` coding spirit

Plethoric accumulation of extended models/observables/features without making the code slower or less readable

Relies on homogeneous style and strict rules (e.g. anything related to given feature is inside an: `if (has_feature == _TRUE_){...}`)

class coding spirit

Plethoric accumulation of extended models/observables/features without making the code slower or less readable

Relies on homogeneous style and strict rules (e.g. anything related to given feature is inside an: `if (has_feature == _TRUE_){...}`)

No hard-coding

- All indices allocated dynamically (according to strict and homogeneous rules for more readability): see dedicated section in third lecture
- All arrays allocated dynamically
- Essentially no number found in the codes except factors in physical equations
- No hard-coded precision parameters, all precision-related numbers/flags gathered in single structure `precision`
- Not a single global variable: all variables passed as arguments of functions (for readability and parallelisation)
- Sampling steps inferred dynamically by the code for each model
- Time for switching approximations on/off inferred dynamically by the code for each model

class coding spirit

Rigorous error management

In principle, no segmentation faults when executing public `class`.
When `class` fails, it returns an error message with a tree-like information (like e.g. python)
We'll see how this works in third lecture...

class coding spirit

Rigorous error management

In principle, no segmentation faults when executing public class.
When class fails, it returns an error message with a tree-like information (like e.g. python)
We'll see how this works in third lecture...

Version history

All previous versions can be downloaded and compared on GitHub, changes documented in `class-code.net`
Always aim at developing without breaking compatibility with older versions.
Own changes can often be merged in newer version with `git merge`.

Installation

Installation should be straightforward on Linux, and slightly tricky but still easy on Mac. We suggest to not even try on Windows.
We really 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?

HELLO, 911? I JUST TRIED TO TOAST SOME BREAD, AND THE TOASTER GREW AN ARM AND STABBED ME IN THE FACE!
DID YOU READ THE TOASTER'S MAN PAGE FIRST?
WELL, NO, BUT ALL I WANTED WAS—
CLICK

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 pdf manual included in the doc folder, or the online documentation page (from the previous page, or from `https://github.com/lesgourg/class_public/wiki`, click on the link `online html documentation`), in the first three subsections:
        - `class`: Cosmic Linear Anisotropy Solving System
        - Where to find information and documentation on `class`? (includes references to many papers useful to understand the `class` equations and physics)
        - `class` overview (architecture, input/output, general principles)
2. More advanced:
    - several detailed courses at different levels on Julien's course webpage `https://lesgourg.github.io/courses.html`, especially the courses from Tokyo; 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-2.7.0/), you should see:

```
explanatory.ini   # reference input file
source/   # the 10 modules of class:
          # ALL THE PHYSICS
tools/    # auxiliary pieces of code (numerical methods):
          # ALL THE MATH (no external C library)
main/     # main class function: short, just calls 10 modules
test/     # other main functions for testing part of the code
output/   # output files (when running from terminal)
include/  # header files (*.h) containing declarations
doc/      # pdf version of the manual
python/   # python wrapper of class
cpp/      # C++ wrapper of class
notebooks/ # example of jupyter notebooks
scripts/ # same as plain python scripts
RealSpaceInterface/ # graphical interface
```

plus a few other directories containing ancillary data (bbn/) or interfaced codes (hyrec/, external_Pk/)

The 10 class modules

Executing class means going once through the sequence of modules:

```
 1. input.c             # parse/make sense of input parameters
                        # (advanced logic)
 2. background.c.       # homogeneous cosmology
 3. thermodynamics.c.   # ionisation history, scattering rate
 4. perturbations.c.    # linear Fourier perturbations
 5. primordial.c.       # primordial spectrum, inflation
 6. nonlinear.c         # recipes for non-linear corrections
                        # to 2-point statistics
 7. transfer.c.         # from Fourier to multipole space
 8. spectra.c.          # 2-point statistics (power spectra)
 9. lensing.c           # CMB lensing
10. output.c            # print out (not used from python)
```

Plain C (for performances and readability purposes) mimicking C++ and object-oriented programming:

- In C++: 10 "classes", each with a constructor/destructor and a few functions callable from outside.
- In class: each module (files *.c and *.h) is associated to one structure (with all its input/output data), one initialisation function, one freeing function, and a few functions callable from outside.
- main executable only consists in calling the 10 initialisation and ten freeing functions!

Running in terminal with input file (old fashioned)

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

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

It gives some output:

```
Reading input parameters
 -> matched budget equations by adjusting Omega_Lambda =
     6.878622e-01
Running CLASS version v2.7.0
Computing background
 -> age = 13.795359 Gyr
 -> conformal age = 14165.045412 Mpc
Computing thermodynamics with Y_He=0.2453
 -> recombination at z = 1089.184869
(...)
Writing output files in output/explanatory01_...
```

Running in terminal with input file (old fashioned)

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

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

It gives some output:

```
Reading input parameters
 -> matched budget equations by adjusting Omega_Lambda =
     6.878622e-01
Running CLASS version v2.7.0
Computing background
 -> age = 13.795359 Gyr
 -> conformal age = 14165.045412 Mpc
Computing thermodynamics with Y_He=0.2453
 -> recombination at z = 1089.184869
(...)
Writing output files in output/explanatory01_...
```

- All possible input parameters and details on the syntax explained in `explanatory.ini`
- This is only a reference file; we advise you to *never* 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
- output comes from 10 verbose parameters fixed to 1 in `explanatory.ini` (see them with `> tail explanatory.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
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
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)
- You can restore the online output with

  ```
  > tail explanatory.ini >> my_model.ini
  ```

  to append 10 verbose parameters at the end of `my_model.ini`
- `./class` can take two input files `*.ini` and `*.pre`:

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

  But one is enough.

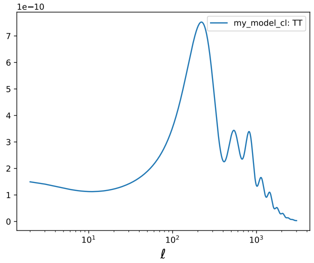
Running in terminal with input file (old fashioned)

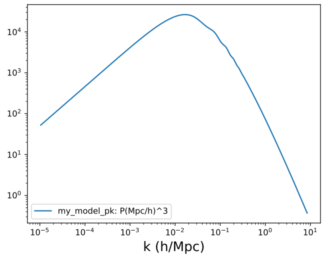
Results are in several files `output/my_model_*.dat`
Can be quickly plotted with provided python script `CPU.py` (Class Plotting Unit):

```
> 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
```





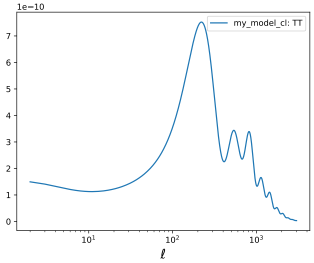
Running in terminal with input file (old fashioned)

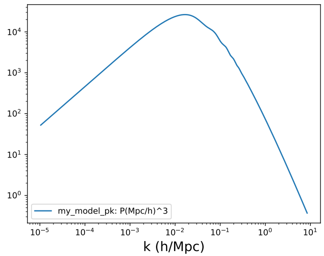
Results are in several files `output/my_model_*.dat`
Can be quickly plotted with provided python script `CPU.py` (Class Plotting Unit):

```
> 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
```