CLASS

the Cosmological Linear Anisotropy Solving System[1]

Julien Lesgourgues
TTK, RWTH Aachen University

Darmouth-TRIUMF-U. of Washington,HEP/Cosmology Tools Bootcamp,
26-27.10.2017

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

Context

CLASS is the 5th public Boltzmann code 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`, see Antony's lectures)
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

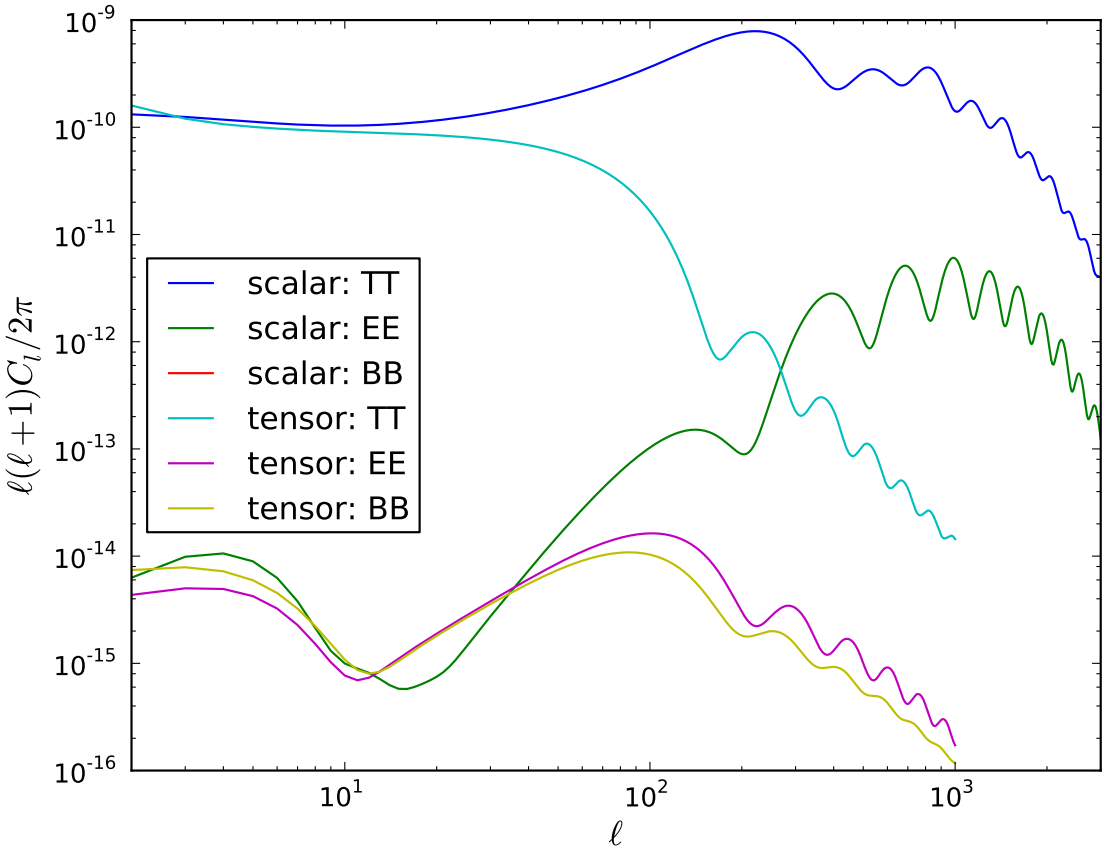
Project started on request of Planck science team, in order to have a tool independent from CAMB, and check for possible Boltzmann-code-induced bias in parameter extraction. The CLASS-CAMB comparison has triggered progress in the accuracy of both codes. Agreement established at 10^{-4} (0.01%) level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

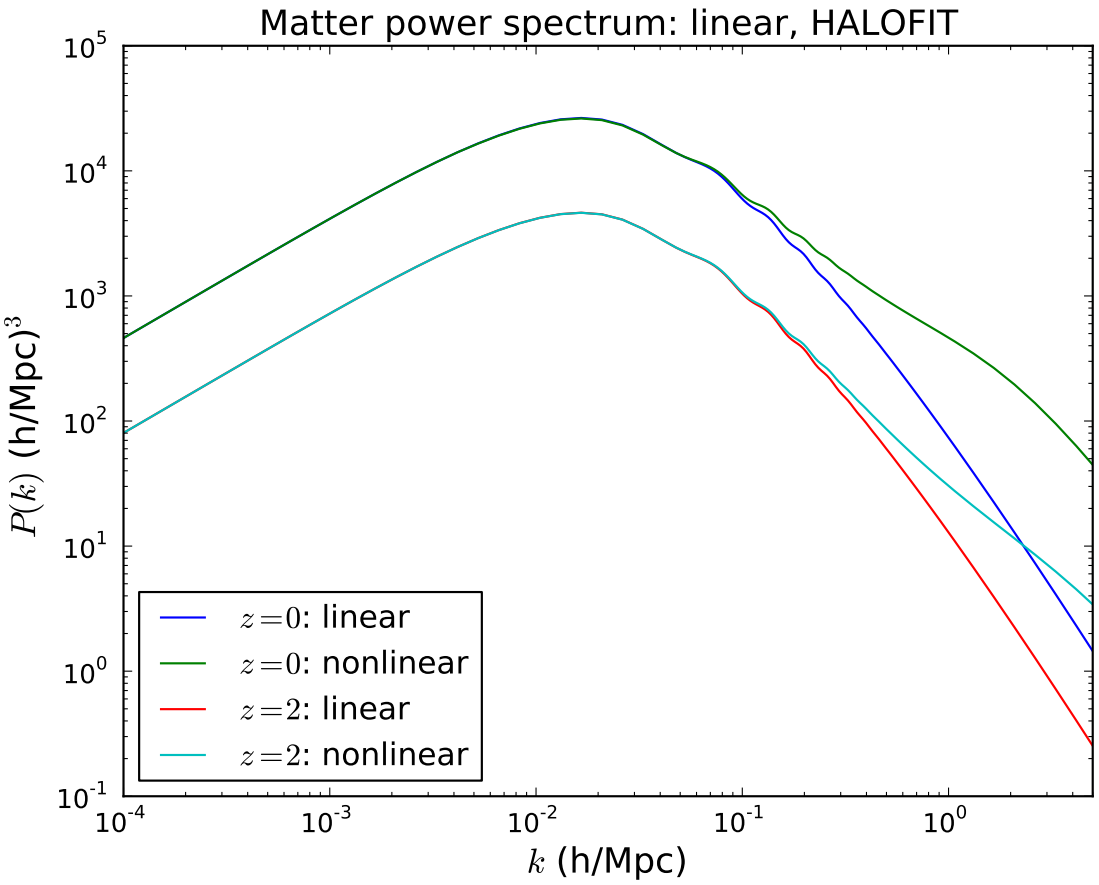
With CLASS you can get:

The CMB anisotropy spectra:



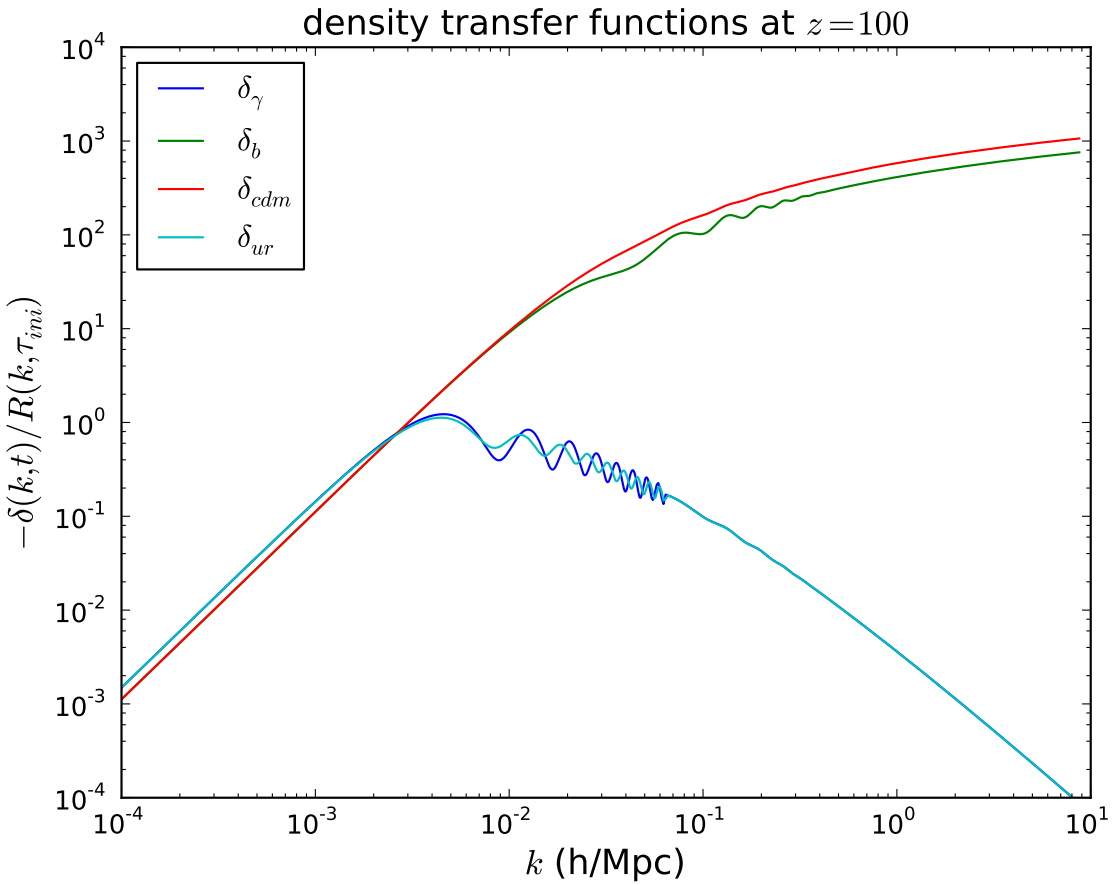
With CLASS you can get:

The matter power spectrum:



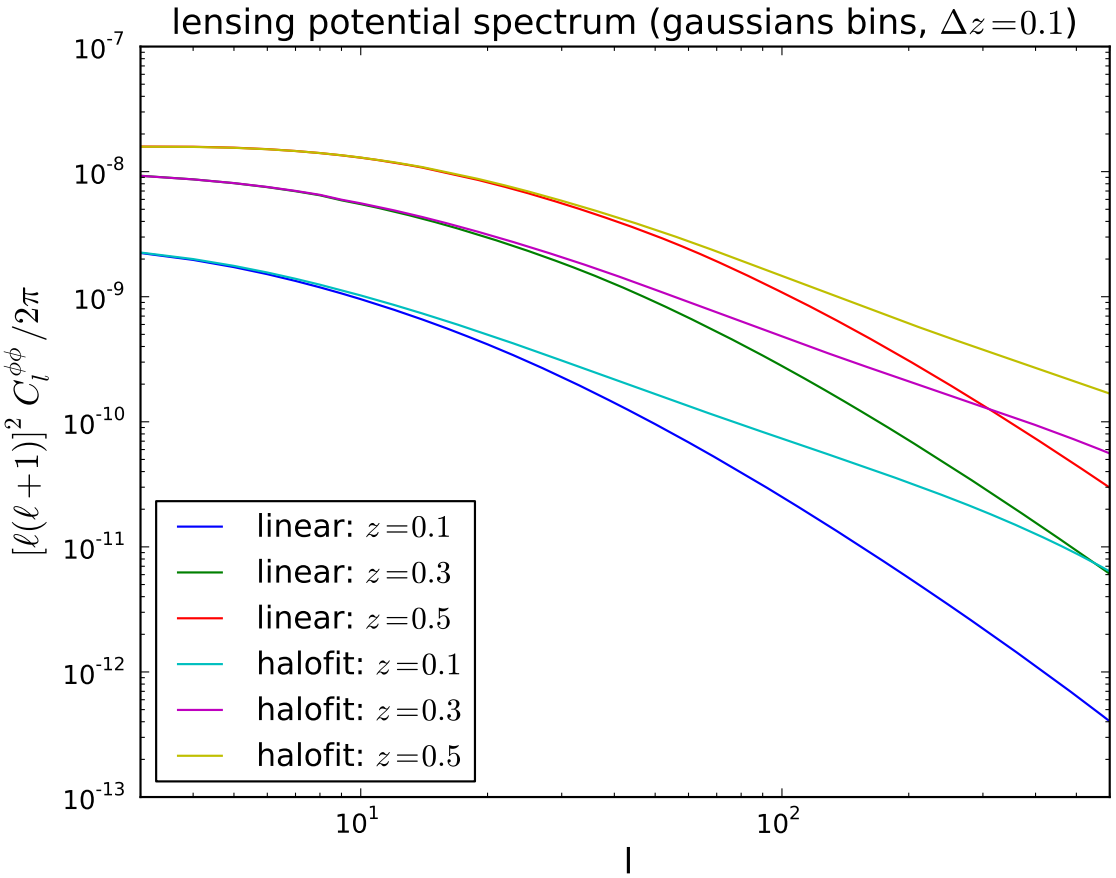
With CLASS you can get:

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



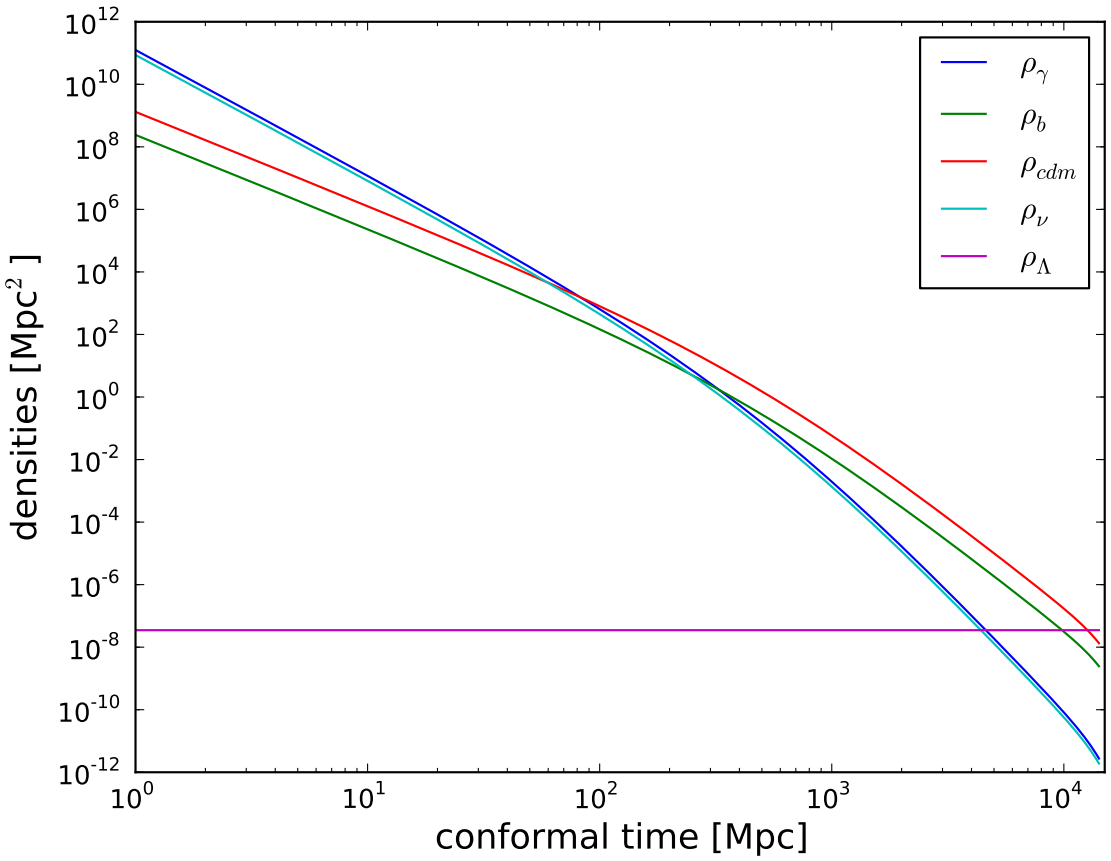
With CLASS you can get:

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



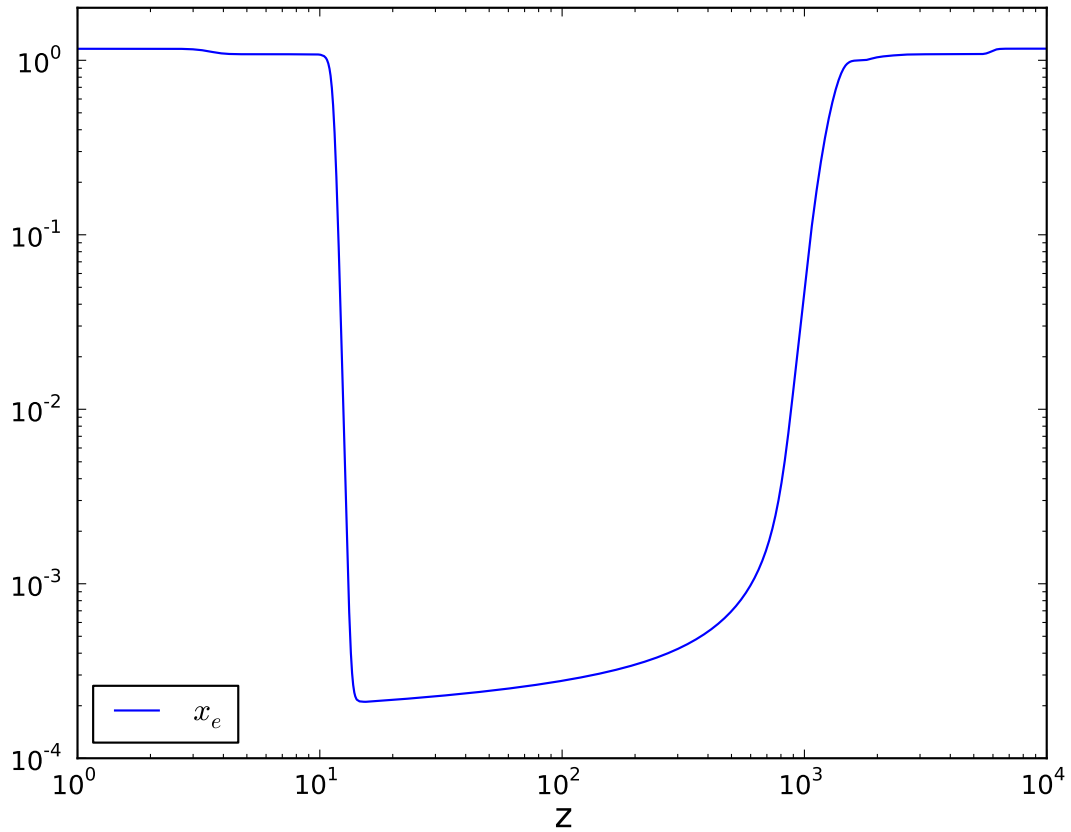
With CLASS you can get:

The background evolution in a given cosmological model:



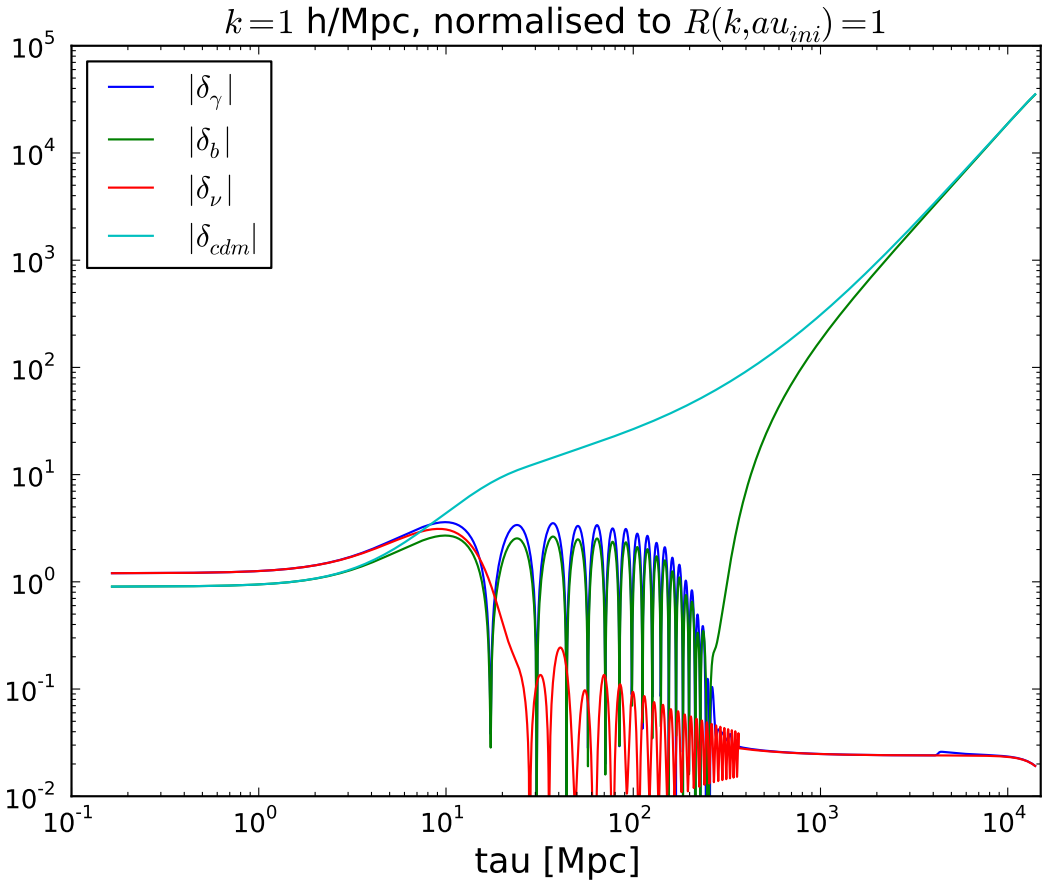
With CLASS you can get:

The thermal history in a given cosmological model:



With CLASS you can get:

The time evolution of perturbations for individual Fourier modes:



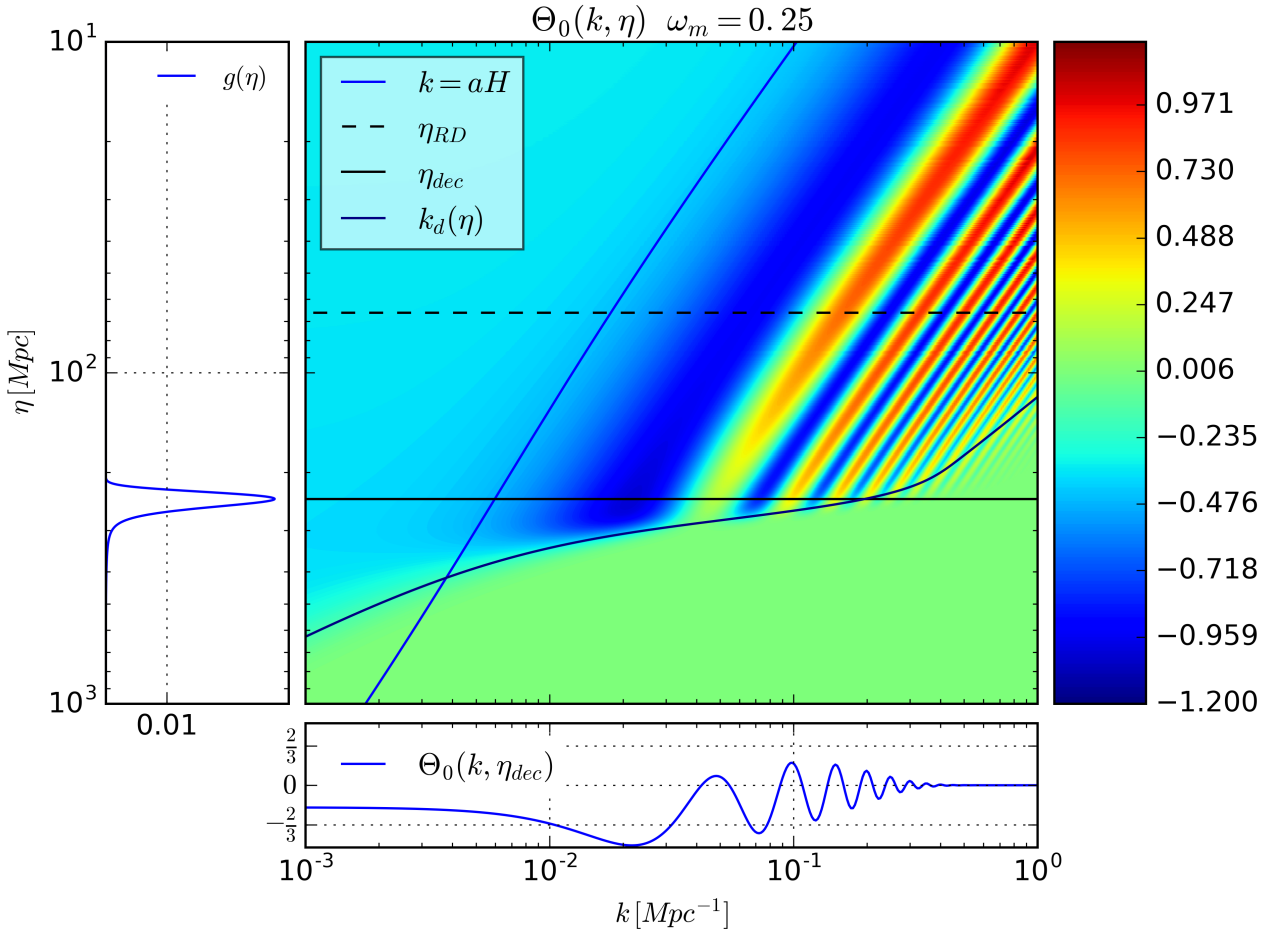
With CLASS you can get:

... 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.;
- ...

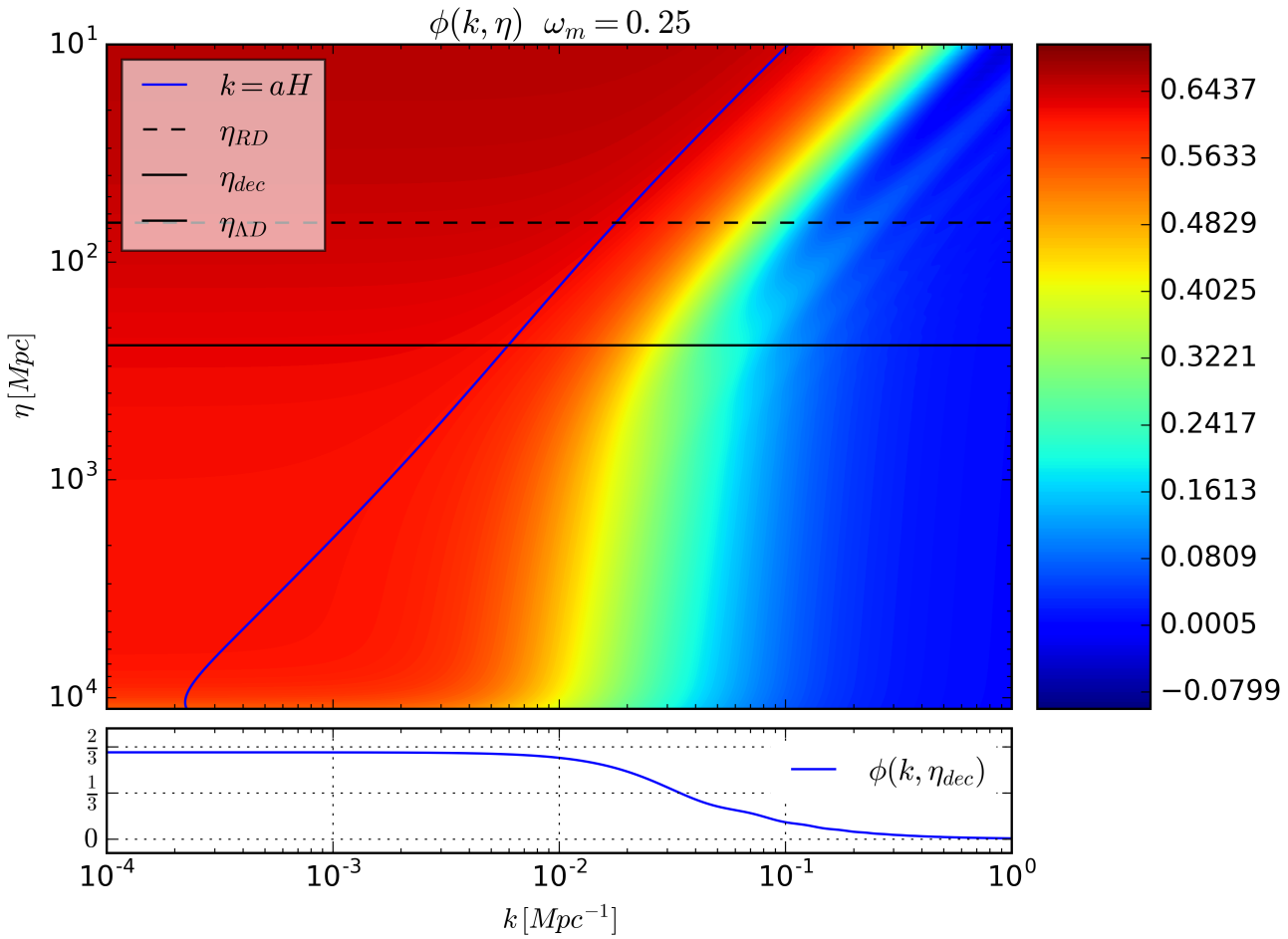
With CLASS you can get:

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



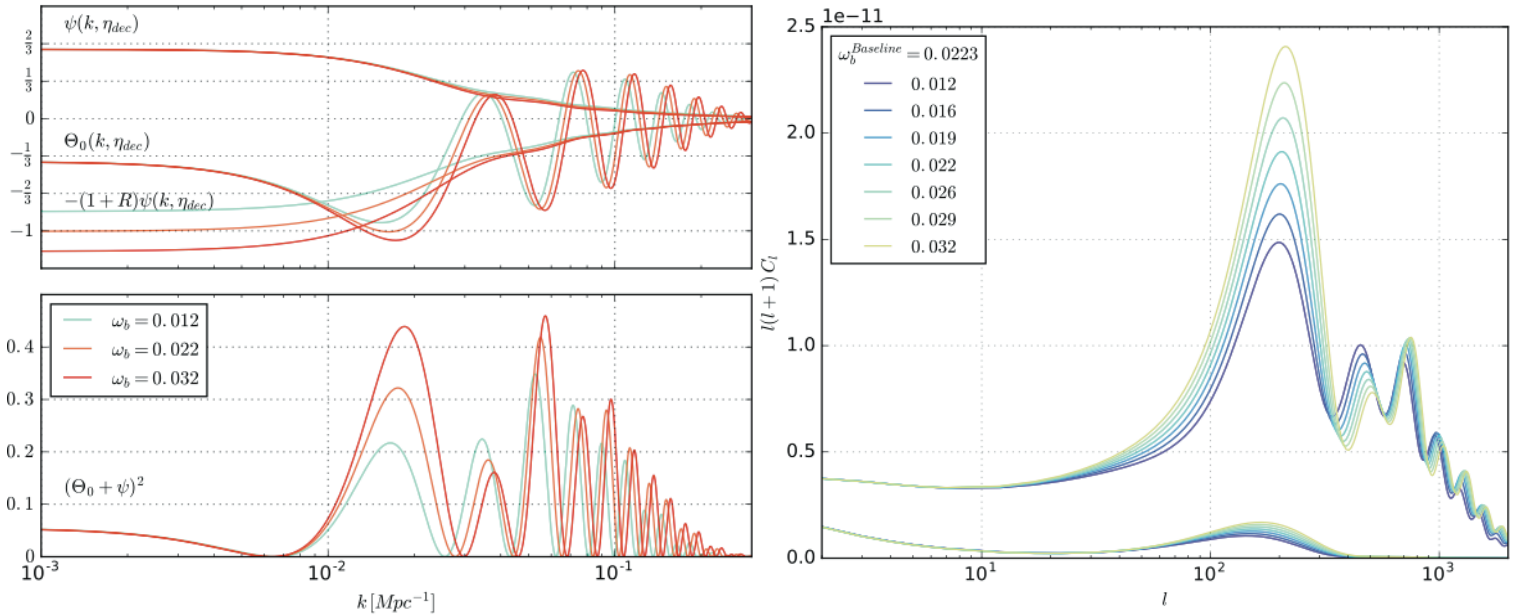
With CLASS you can get:

... 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:



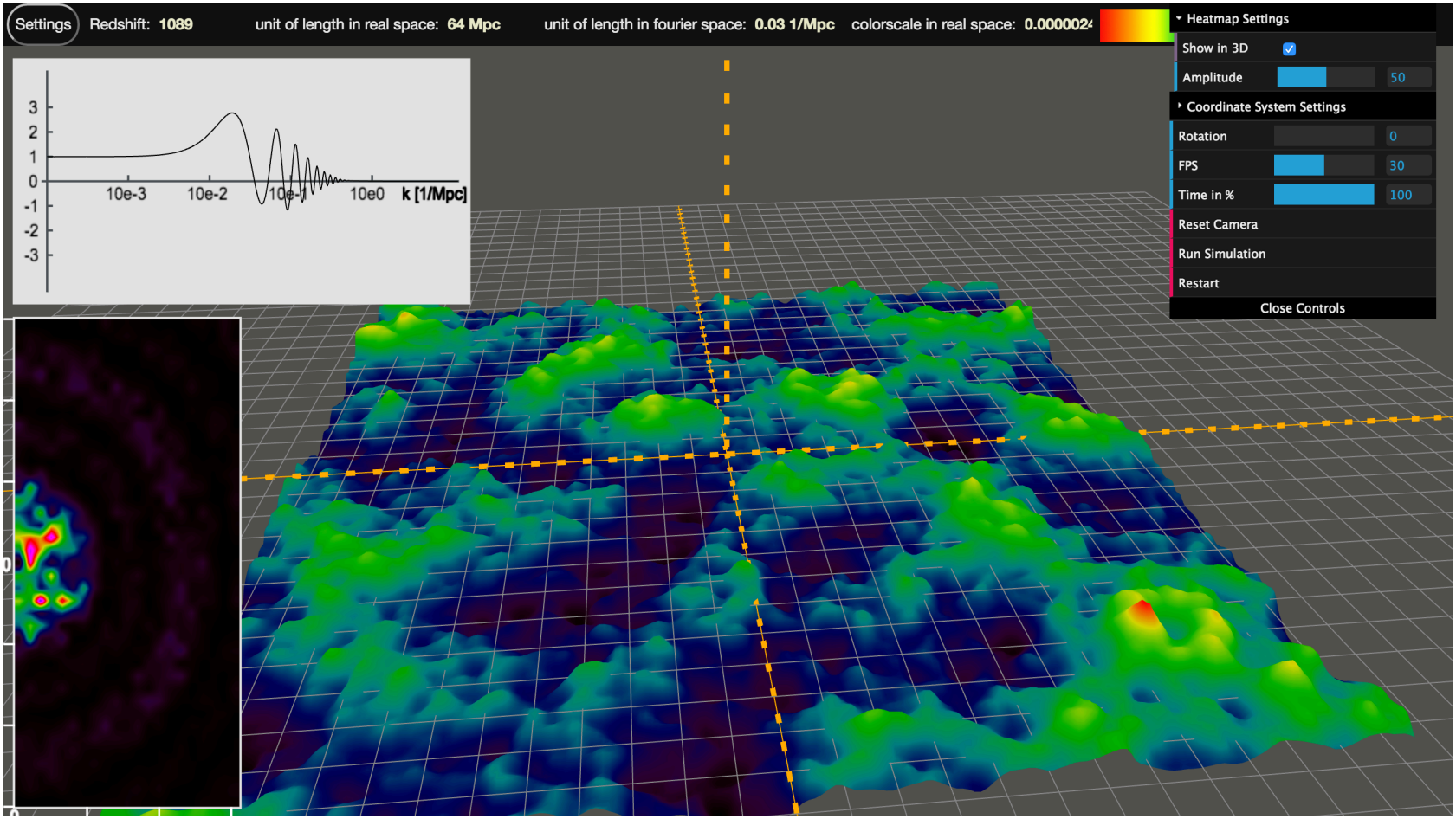
With CLASS you can get:

... 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, 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`)

In these two lectures...

... we will...

1. mention a few aspects of the CLASS coding style
2. show how to run on a terminal with an input file (old-fashioned)
3. give example of python scripts / jupyter notebooks for testing and plotting

Installation

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

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

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

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

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

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

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

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

1. Basic information and links:
   - in the historical CLASS webpage `http://class-code.net`
   - in the online documentation page (from the previous page, or from `https://github.com/lesgourg/class_public/wiki`, clik on the link `online html documentation`), in the first two subsections:
     - CLASS: Cosmic Linear Anisotropy Solving System
     - Where to find information and documentation on CLASS?
     - CLASS overview (architecture, input/output, general principles)
2. More advanced:
   - several detailed courses at different levels on my course webpage `https://lesgourg.github.io/courses.html`, especially the courses from Narbonne and Tokyo; this Bootcamp 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-2.6.3/), 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
output/   # directory for output files
include/  # header files (*.h) containing declarations
test/     # other main functions for testing part of the code
python/   # python wrapper of CLASS
cpp/      # C++ wrapper of CLASS
```

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

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 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 function!

class coding spirit

- equations follow literally notations of most famous papers (like Ma & Bertschinger 1996)
- lots of comments in the code + automatic `doxygen` documentation
- plethoric accumulation of extended models/observables/features without making the code slower or less readable, thanks to homogeneous style and strict rules (example: all array index values are allocated dynamically, with explicit names following homogeneous conventions: `index_pt_delta_cdm` means "index of the perturbation $\delta_{\rm cdm}$", etc.)
- no harded-coded precision parameters ("hidden magic numbers"), all precision-related numbers/flags gathered in single structure `precision`
- rigorous error management: when `class` fails, it returns an error message with tree-like informations (like e.g. python)
- not a single global variables
- development without breaking compatibility with old versions
- ...

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 v2.6.3
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 v2.6.3
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 *never* to modify it, but rather to copy it and reduce it to a shorter and more friendly file.
- For *basic* usage: `explanatory.ini` $\equiv$ 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 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.

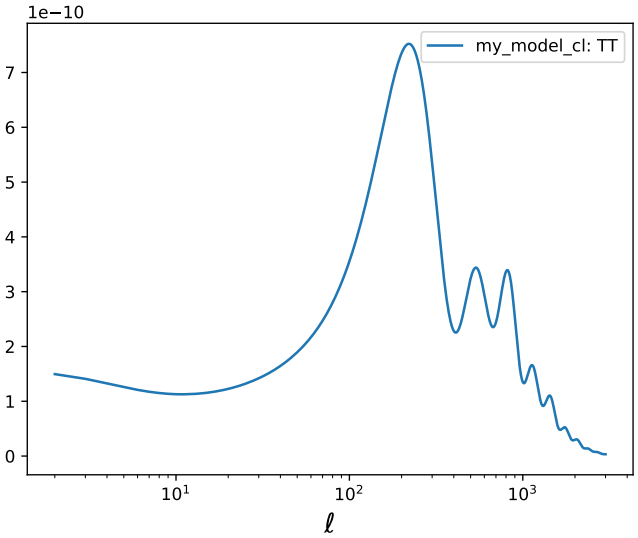
Running in terminal with input file (old fashioned)

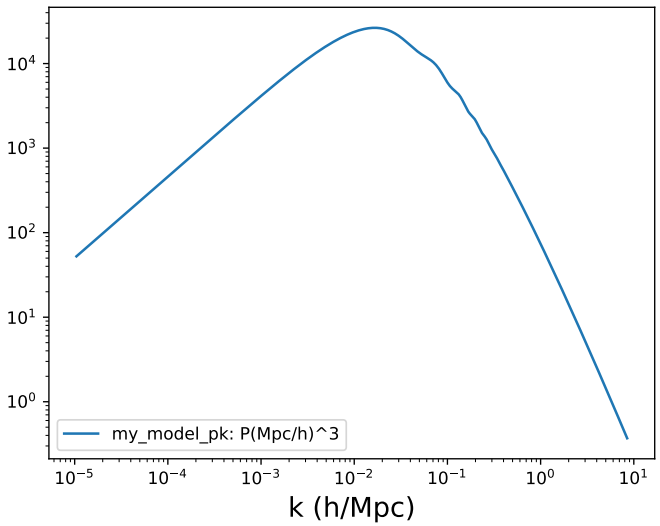
Results are in several files `output/my_model_*.dat`
Can be quickly plotted with provided python script `CPU.py` (Class Plotting Unit):

```
> 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
```





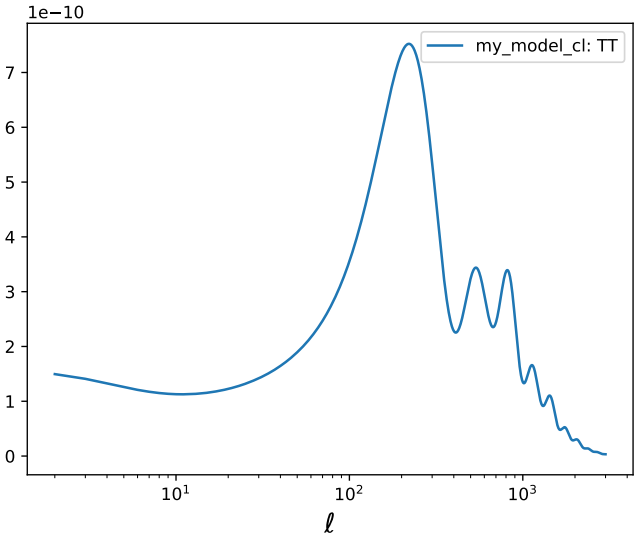
Running in terminal with input file (old fashioned)

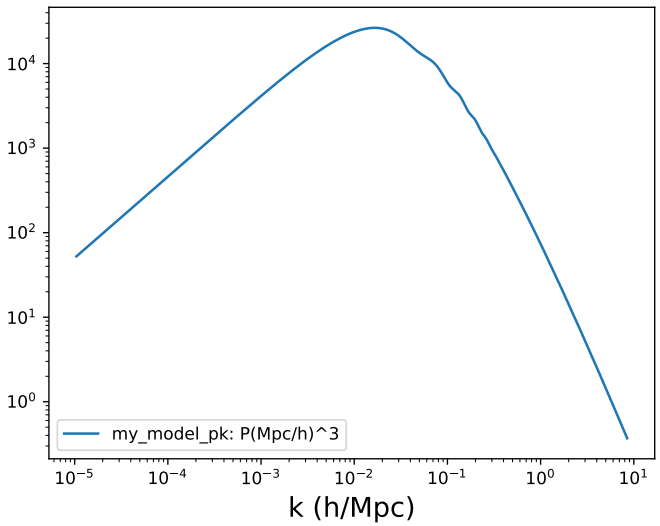
Results are in several files `output/my_model_*.dat`
Can be quickly plotted with provided python script `CPU.py` (Class Plotting Unit):

```
> 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
```

Running class from python

class as a Python module

- based on wrapper located in `python/classy.pyx`
- the compilation produced a python module `classy.py` and installed it on your computer
- wrapper written in Cython, encapsulates most useful class variables/functions, contains extra functions
- (project to get most of the wrapper generated automatically from C code at compilation)
- goal: obtain, manipulate and plot the results directly within (i)python scripts or notebooks (recommended)

- we will now discuss several examples of scrips/notebooks
- we provide an archive `class_ipynb_py.tar.gz` containing them in script (*.py) and notebook (*.ipynb) version
- ideally you would have `jupyter` installed, and you would try to open and execute them with

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

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

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

Python wrapper

First basic example (download `class_ipynb_py.tar.gz` and see `notebooks/warmup.ipynb` or `scripts/warmup.py`)

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

```
# create instance of the class "Class"
LambdaCDM = Class()
# pass input parameters
LambdaCDM.set({'omega_b':0.022032,'omega_cdm':0.12038,'h':0.67556,'A_s':2.215e
    -9,'n_s':0.9619,'tau_reio':0.0925})
LambdaCDM.set({'output':'tCl,pCl,lCl,mPk','lensing':'yes','P_k_max_1/Mpc':3.0})
# run class
LambdaCDM.compute()
```

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

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

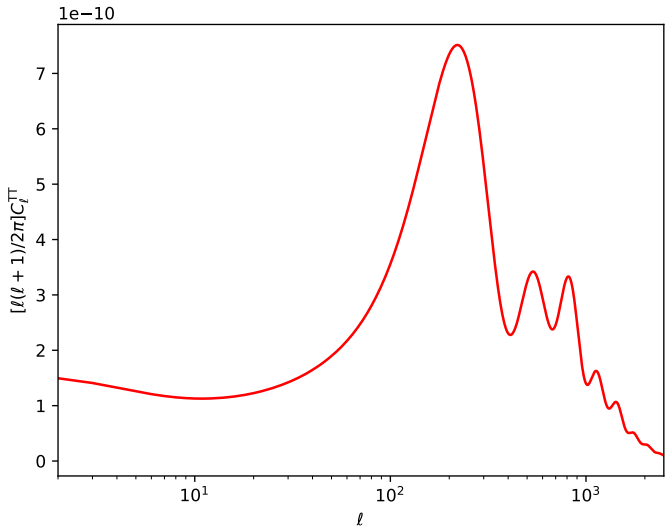
```
ll = cls['ell'][2:]
clTT = cls['tt'][2:]
clEE = cls['ee'][2:]
clPP = cls['pp'][2:]
```

Python wrapper

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

```
# to get plots displayed in notebook
%matplotlib notebook
import matplotlib.pyplot as plt
from math import pi
```

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



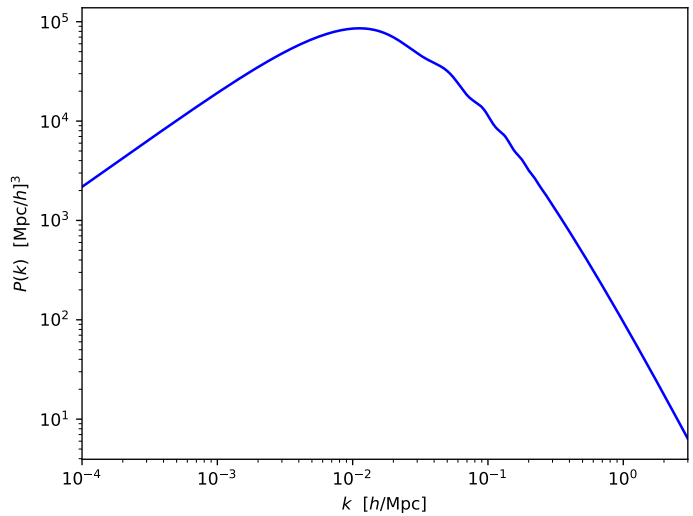

```
plt.savefig('warmup_cltt.pdf')
```

Python wrapper

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

```
# get P(k) at redhsift z=0
import numpy as np
kk = np.logspace(-4,np.log10(3),1000)
Pk = []
for k in kk:
    Pk.append(LambdaCDM.pk(k,0.)) # function .pk(k,z)
```

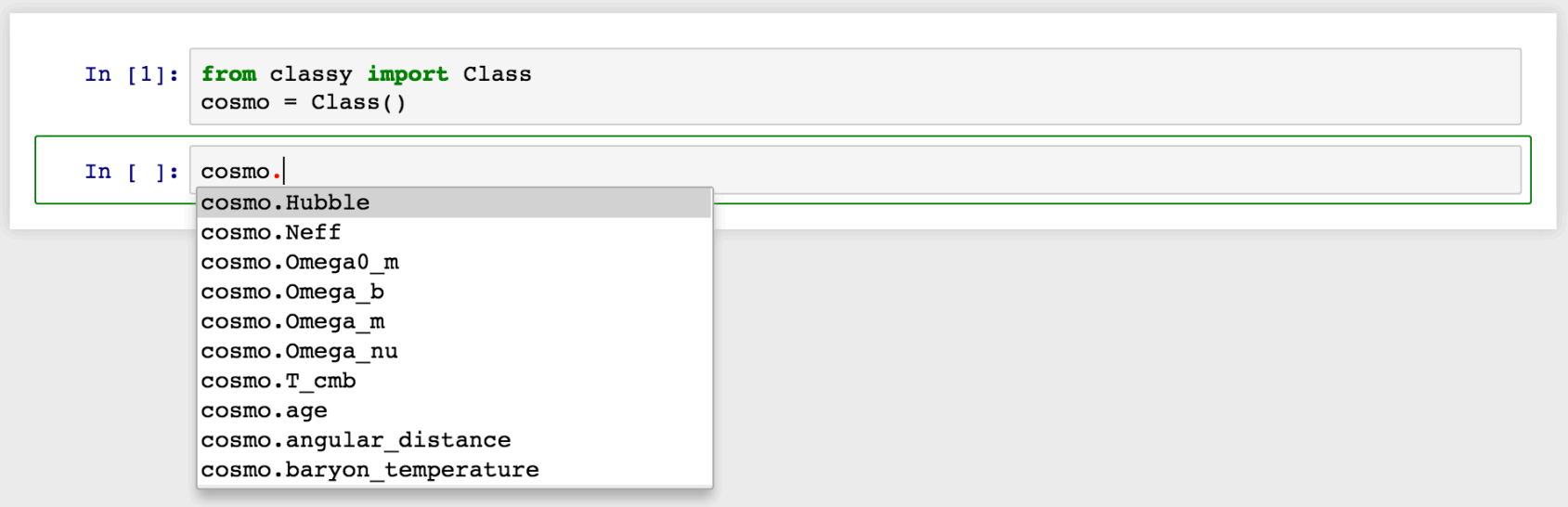
```
# plot P(k)
plt.figure(2)
plt.xscale('log');plt.yscale('log');plt.xlim(kk[0],kk[-1])
plt.xlabel(r'$k \,\,\,\,\, [h/\mathrm{Mpc}]$')
plt.ylabel(r'$P(k) \,\,\,\,\, [\mathrm{Mpc}/h]^3$')
plt.plot(kk,Pk,'b-')
```




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

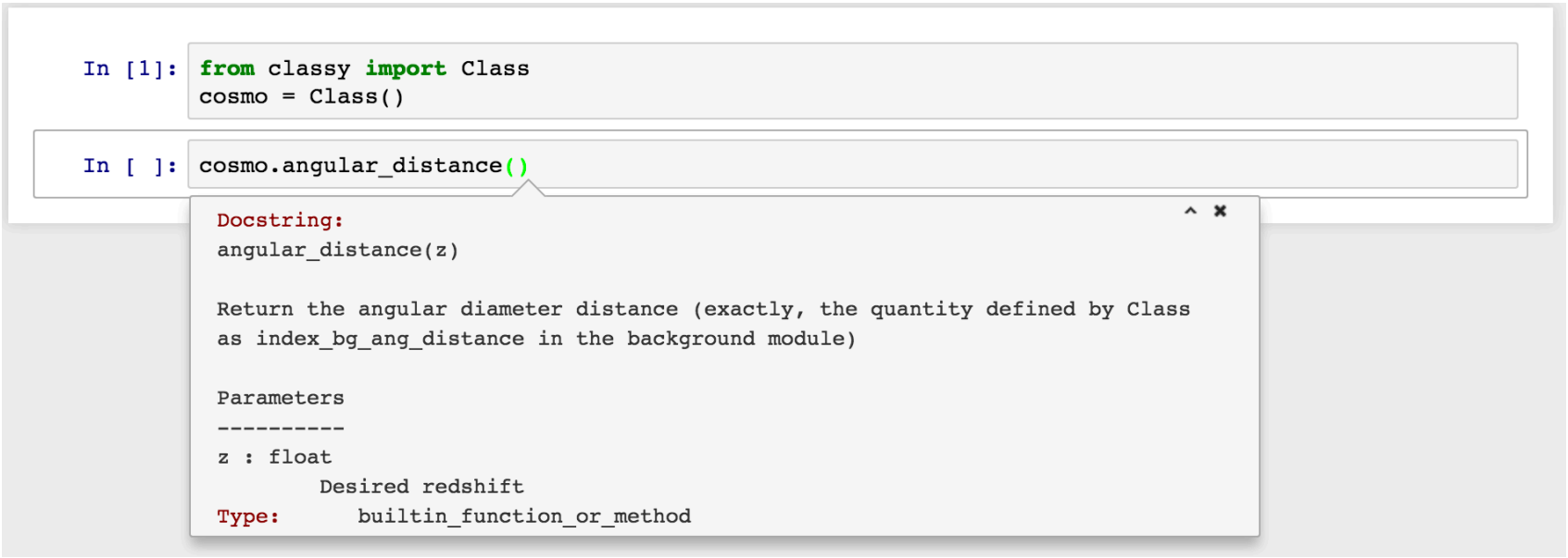
Python wrapper: IPython notebooks

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



Python wrapper: IPython notebooks

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



Species in public class

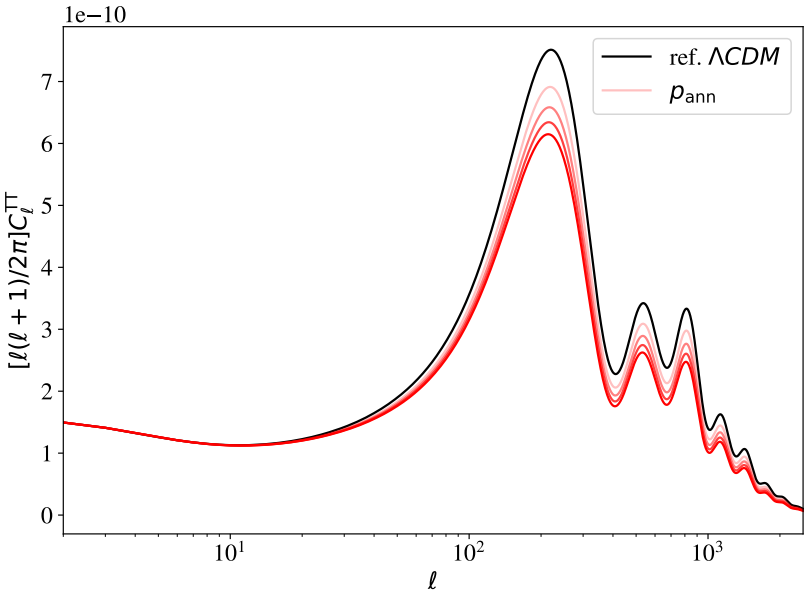
- photons: T_cmb or Omega_g or omega_g
- baryons: Omega_b or omega_b
- ultra-relativistic species (massless neutrinos): N_ur or Omega_ur or omega_ur
- cold dark matter: Omega_cdm or omega_cdm (eventually annihilating: annihilation, etc.)
- N_ncdm distinct non-cold dark matter species (massive neutrinos,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

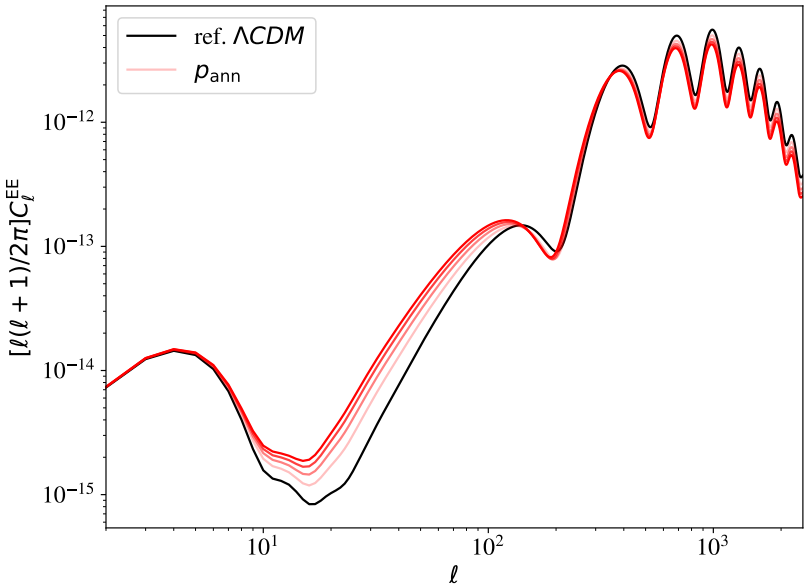
All details are in explanatory.ini

To avoid over-constraining the input, one of the last three (Omega_Lambda, Omega_fld, Omega_scf) must be left unspecified and class will assign it using budget equation. Defaut: Omega_fld = Omega_scf = 0 so Omega_Lambda is automatically adjusted.

Plots with varying parameters

With `notebooks/varying_pann.ipynb` or `scripts/varying_pann.py`:





We called `class` within a loop with different values of the DM annihilation parameter $p_{\rm ann} = \frac{\langle \sigma v \rangle}{m}$.

Plots with varying parameters

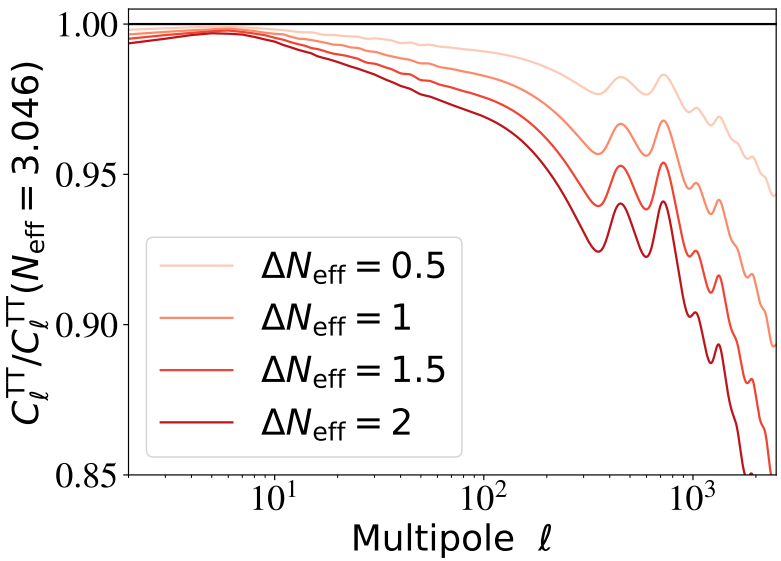
With `notebooks/varying_pann.ipynb` or `scripts/varying_pann.py`:

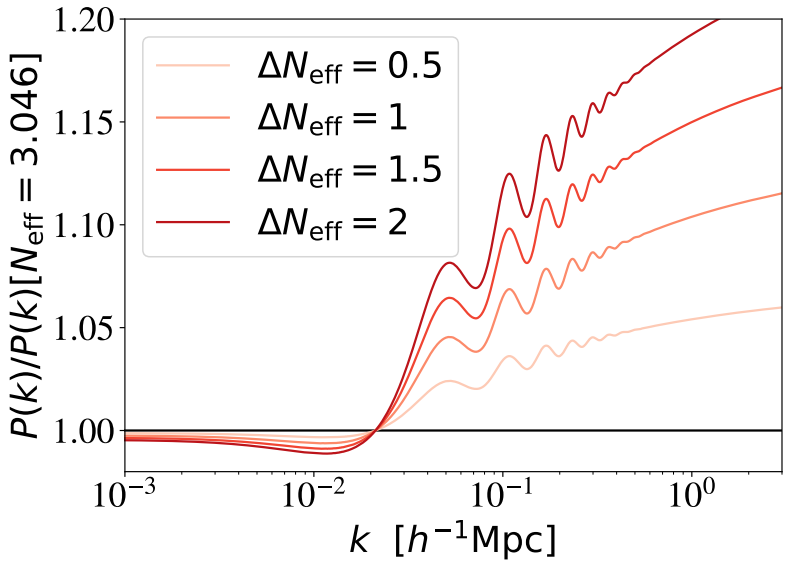
Main steps:

```
var_name = 'annihilation'
common_settings = {'output':'tCl,pCl,lCl,mPk', ...}
# loop over varying parameter values
for i in range(var_num):
  var = var_min + (var_max-var_min)*i/(var_num-1.)
  M = Class()
  M.set(common_settings)
  M.set({var_name:var})
  M.compute()
  clM = M.lensed_cl(2500)
  # ... plotting ...
  M.empty()           # clear input previously set by .set()
  M.struct_cleanup() # clear all class output
```

Plots with varying parameters

With `notebooks/varying_neff.ipynb` or `scripts/varying_neff.py`:





Slightly more elaborate: we had to call `class` with different values of $N_{\rm eff}$ for massless neutrinos (in fact `N_ur`) while keeping $z_{\rm eq}$ and z_Λ fixed, which implies to adjust `h` and `omega_cdm` in a non-trivial way. We also wanted a separate cell for calling `class` for each model, and then for plotting.

Plots with varying parameters

With `notebooks/varying_neff.ipynb` or `scripts/varying_neff.py`:
Main steps:

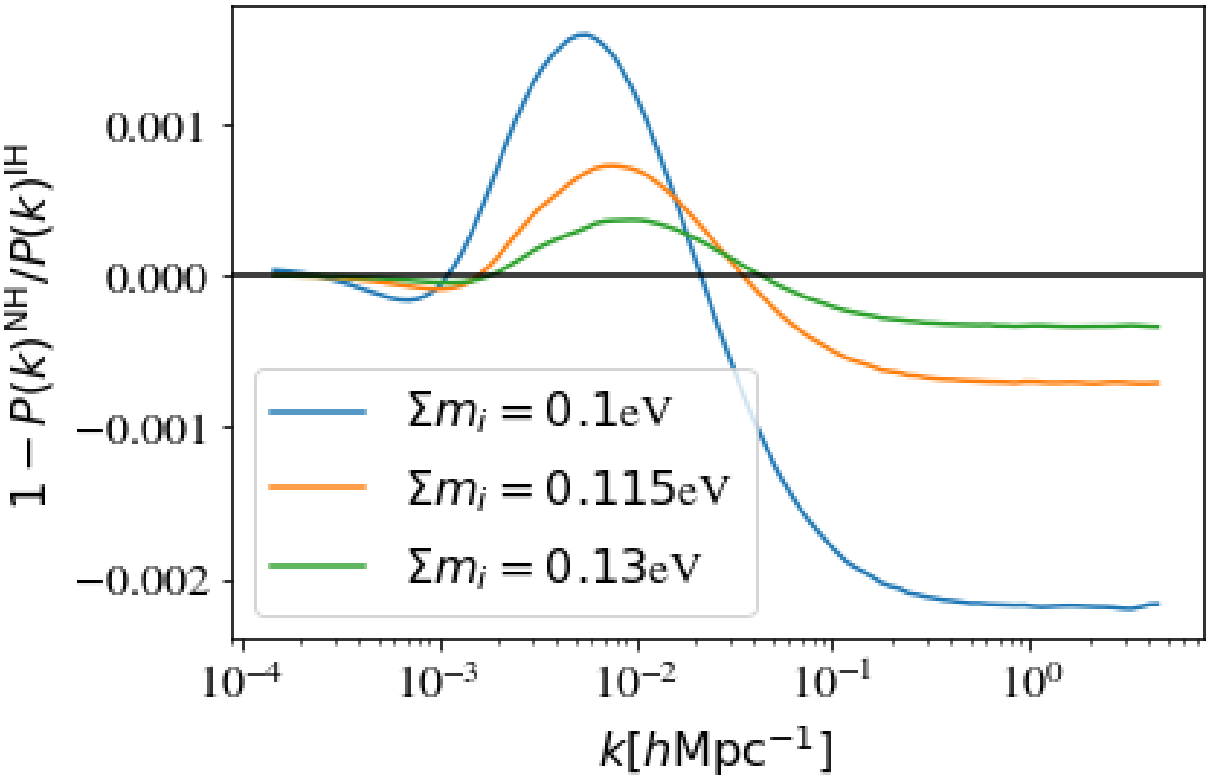
```
M = {}
for i in range(var_num):
    # The goal is to vary
    # - omega_cdm by a factor alpha = (1 + coeff*Neff)/(1 +
        coeff*3.046)
    # - h by a factor sqrt*(alpha)
    # in order to keep a fixed z_equality(R/M) and
        z_equality(M/Lambda)
    N_ur = var_min + (var_max-var_min)*i/(var_num-1.)
    alpha = (1.+coeff*N_ur)/(1.+coeff*3.046)
    omega_cdm = (0.022032 + 0.12038)*alpha - 0.022032
    h = 0.67556*math.sqrt(alpha)
    M[i] = Class()
    M[i].set(common_settings)
    M[i].set({'N_ur':N_ur})
    M[i].set({'omega_cdm':omega_cdm})
    M[i].set({'h':h})
    M[i].compute()
```

In the next cell there is another loop for plotting the data from
`clM[i] = M[i].lensed_cl(2500)` and `M[i].pk(k,0.)`

Plots with varying parameters

With `notebooks/neutrinohierarchy.ipynb` or `scripts/neutrinohierarchy.py`:

The goal here is to plot the ratio of $P(k)$ with 3 massive neutrinos obeying to Normal Hierarchy over $P(k)$ with 3 massive neutrinos obeying to Inverted Hierarchy, both with the same total mass $\sum_i m_i$.



Plots with varying parameters

With `notebooks/neutrinohierarchy.ipynb` or `scripts/neutrinohierarchy.py`:

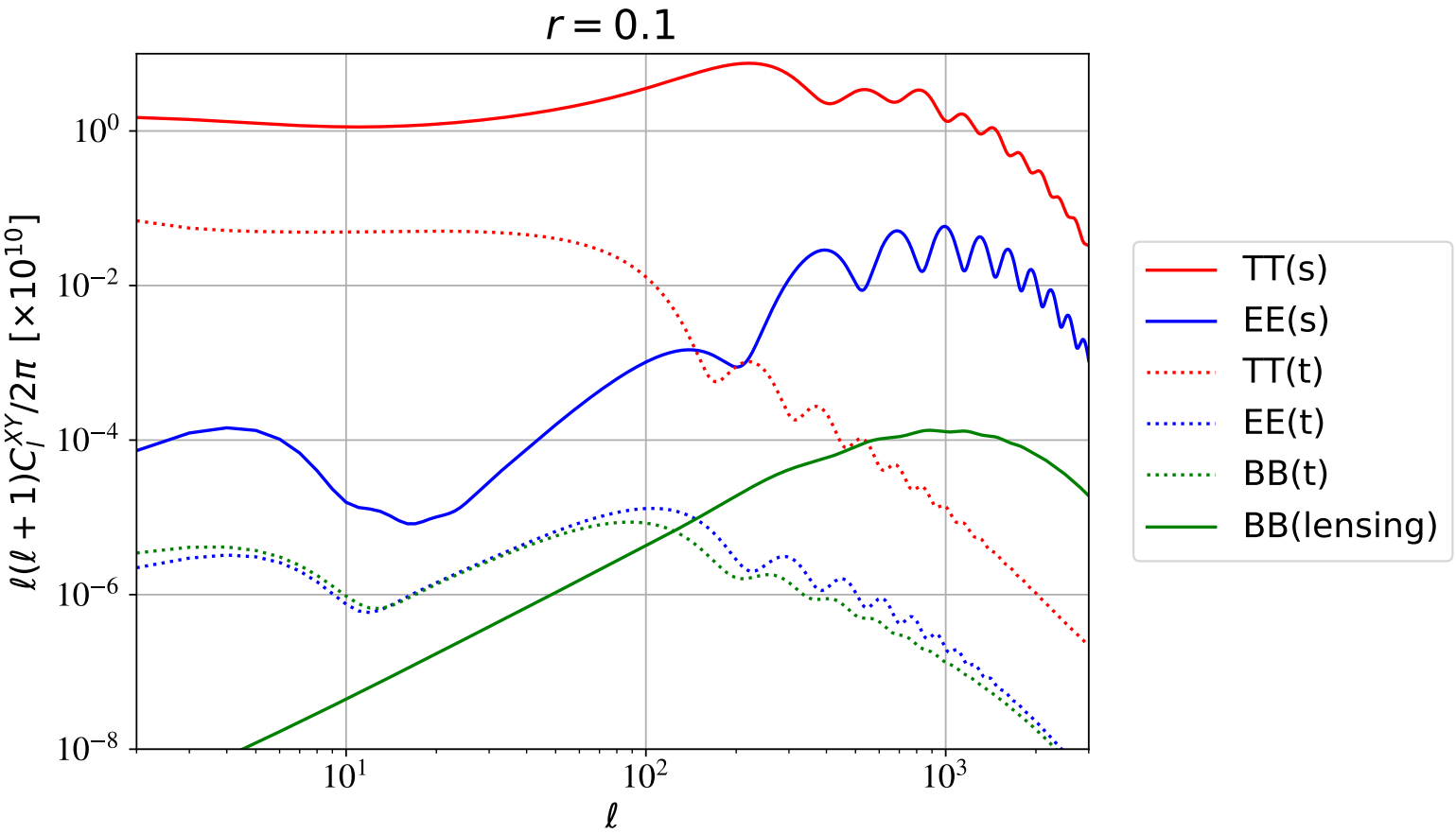
Main steps:

```
def get_masses(delta_m_squared_atm, delta_m_squared_sol,
    sum_masses, hierarchy):
    # function returning individual masses for given sum
```

```
# loop over total mass values
for sum_masses in [0.1, 0.115, 0.13]:
    # normal hierarchy
    [m1,m2,m3] = get_masses(2.45e-3,7.50e-5,sum_masses,'NH')
    NH = Class()
    NH.set(commonsettings)
    NH.set({'m_ncdm':str(m1)+','+str(m2)+','+str(m3)})
    NH.compute()
    # inverted hierarchy
    [m1,m2,m3] = get_masses(2.45e-3,7.50e-5,sum_masses,'IH')
    IH = Class()
    IH.set(commonsettings)
    IH.set({'m_ncdm':str(m1)+','+str(m2)+','+str(m3)})
    IH.compute()
    ...
```

Contributions to CMB C_l's

With `notebooks/cl_ST.ipynb` or `scripts/cl_ST.py`:



Contributions to CMB C_l's

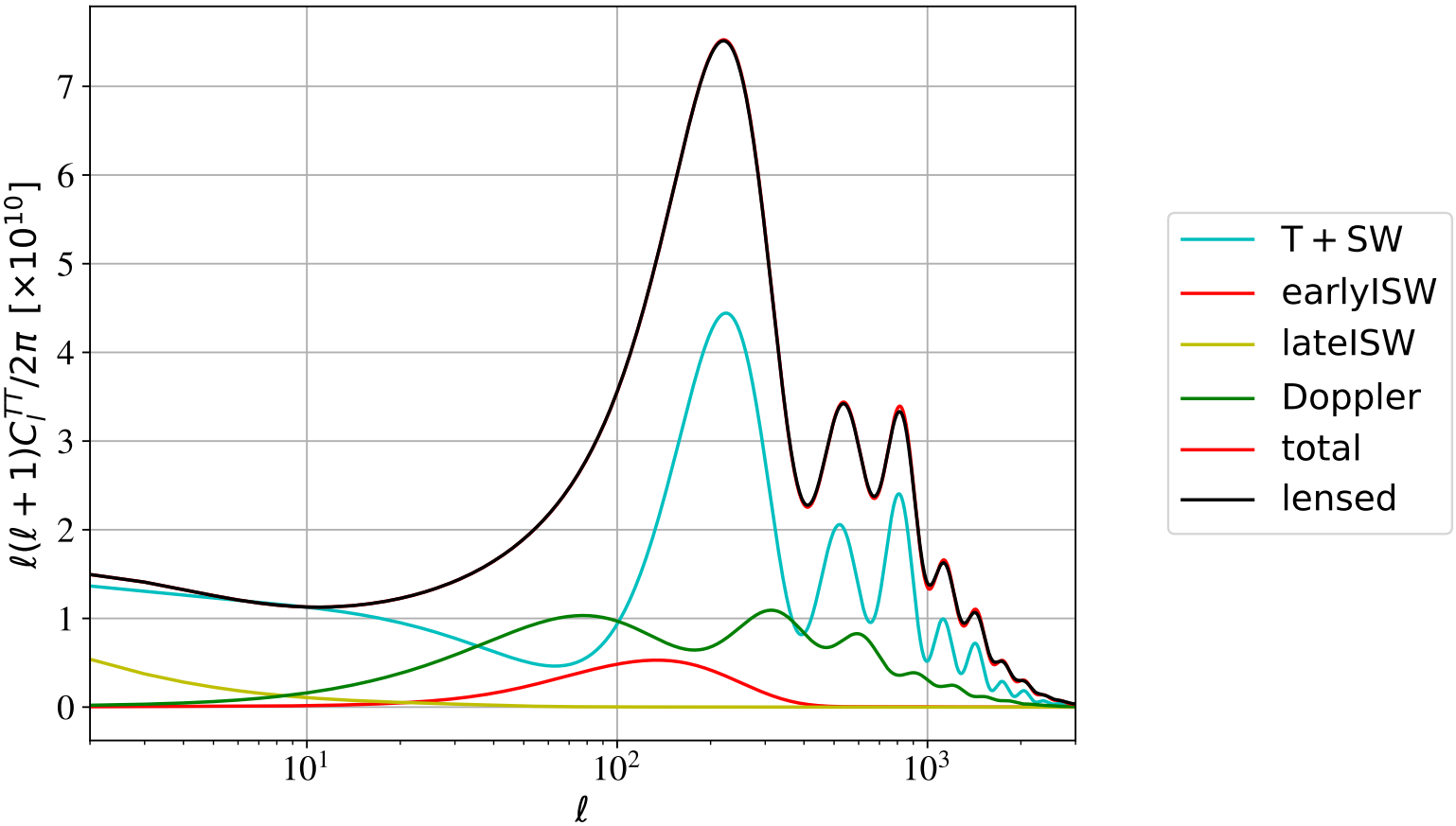
With `notebooks/cl_ST.ipynb` or `scripts/cl_ST.py`:

Main steps:

```
# scalars only
M = Class()
M.set({'output':'tCl,pCl','modes':'s','lensing':'no','n_s'
    :0.9619,'l_max_scalars':3000})
cls = M.raw_cl(3000)
...
# tensors only
M.set({'output':'tCl,pCl','modes':'t','lensing':'no','r'
    :0.1,'n_t':0,'l_max_tensors':l_max_tensors})
clt = M.raw_cl(l_max_tensors)
...
# scalars + tensors (only in this case we can get the
    correct lensed ClBB)
M.set({'output':'tCl,pCl,lCl','modes':'s,t','lensing':'yes',
    'r':0.1,'n_s':0.9619,'n_t':0,'l_max_scalars':3000,'
    l_max_tensors':l_max_tensors})
M.compute()
cl_tot = M.raw_cl(3000)
cl_lensed = M.lensed_cl(3000)
...
```

Contributions to CMB C_l's

With `notebooks/cltt_terms.ipynb` or `scripts/cltt_terms.py`:



Contributions to CMB C_l's

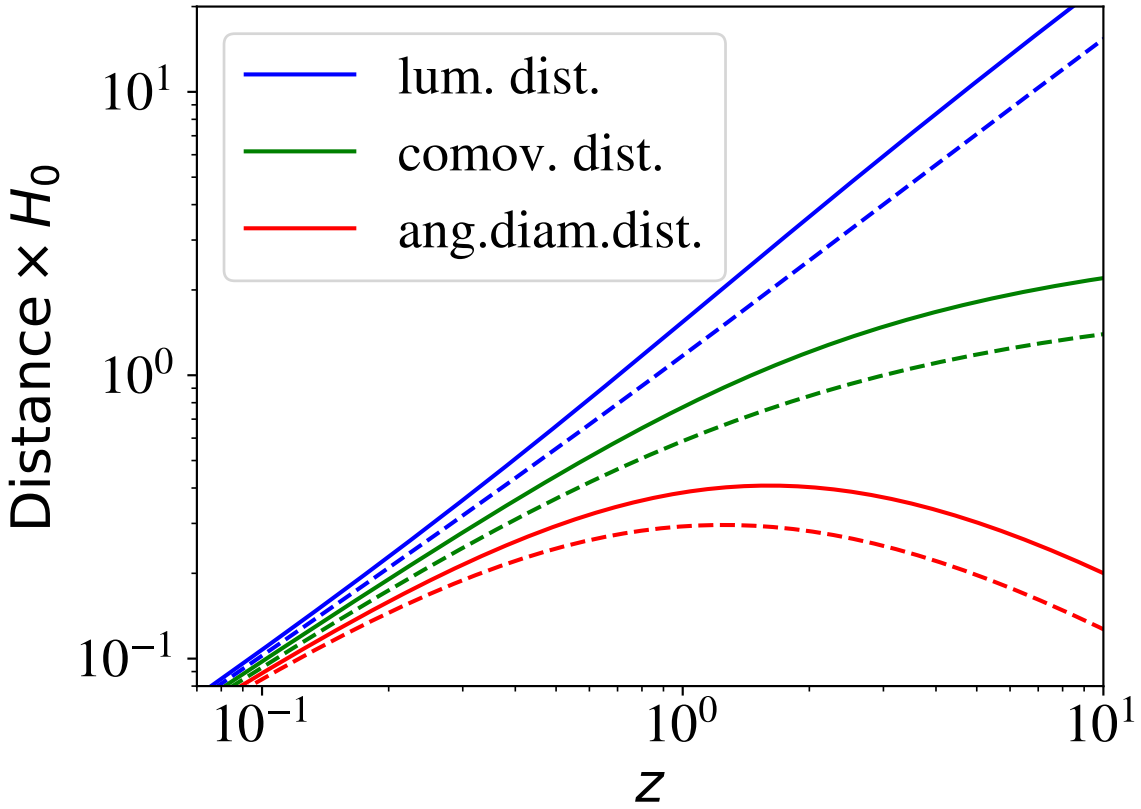
With `notebooks/cltt_terms.ipynb` or `scripts/cltt_terms.py`:

Main steps:

```
M = Class()
M.set(common_settings)
M.compute()
cl_tot = M.raw_cl(3000)
cl_lensed = M.lensed_cl(3000)
M.struct_cleanup()  # clean output
M.empty()           # clean input
...
M.set({'temperature contributions':'tsw'})
M.compute()
cl_tsw = M.raw_cl(3000)
...
M.set({'temperature contributions':'eisw'})
...
M.set({'temperature contributions':'lisw'})
...
M.set({'temperature contributions':'dop'})
```

Background quantitites

With `notebooks/distances.ipynb` or `scripts/distances.py`:



Similar to plot in Scott Dodelson's book.
Solid $=$ ΛCDM, dashed $=$ Einstein-De-Sitter ($\Omega_m = 1$).

Background quantitites

With `notebooks/distances.ipynb` or `scripts/distances.py`:

Main steps:

```
#Lambda CDM
LCDM = Class()
LCDM.set({'Omega_cdm':0.25,'Omega_b':0.05})
LCDM.compute()
#Einstein-de Sitter
CDM = Class()
CDM.set({'Omega_cdm':0.95,'Omega_b':0.05})
CDM.compute()
# Just to cross-check that Omega_Lambda is negligible
# (but not exactly zero because we neglected radiation)
Omega_lambda = CDM.get_current_derived_parameters(['
    Omega0_lambda'])
print Omega_lambda
```

Remark: we did not pass anything to `'output'` field. Seeing that no spectra need to be computed, `class` will only call its background and thermodynamics modules.

Background quantitites

With `notebooks/distances.ipynb` or `scripts/distances.py`:

Main steps:

```
#Get background quantities and recover their names:
baLCDM = LCDM.get_background()
baCDM = CDM.get_background()
baCDM.viewkeys()
```

```
dict_keys(['(.)rho_crit', 'lum. dist.', '(.)rho_b', 'H [1/Mpc]', 'conf.
time [Mpc]', 'comov.snd.hrz.', '(.)rho_g', '(.)rho_lambda', 'comov. dist
.', '(.)rho_cdm', 'ang.diam.dist.', 'proper time [Gyr]', 'gr.fac. D', '
gr.fac. f', 'z', '(.)rho_ur'])
```

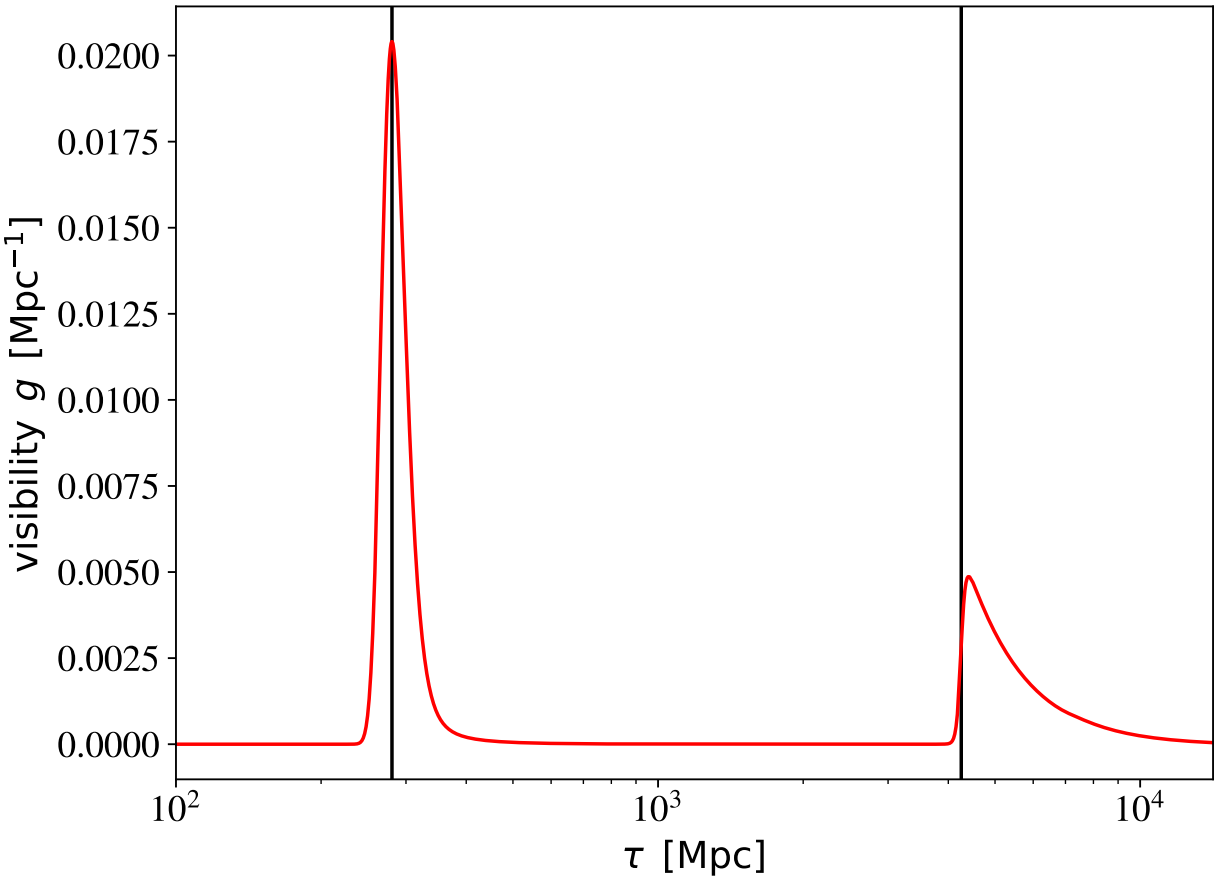
So this big array contains all background quantities for each value of `'z'` (redshift) or `'proper time [Gyr]'`.

There are also many functions directly giving interpolated values of background quantities at a given redshift:
`.Hubble(z)`, `.angular_distance(z)`, `.luminosity_distance(z)`,
`.scale_independent_growth_factor(z)`,
`.scale_independent_growth_factor_f(z)`,
`.sigma8(z)`, `.sigma(R,z)`,

(Also `.z_of_r([z_1, z_n])` which returns r and dz/dr).

Thermodynamics quantitites

With `notebooks/thermo.ipynb` or `scripts/thermo.py`:



Visibility function = probability of last interaction of a photon. Rescaled by factor 100 at late times to make reionisation peak visible on the same scale.

Thermodynamics quantitites

With `notebooks/thermo.ipynb` or `scripts/thermo.py`:

Main steps:

```
M = Class()
M.set(common_settings)
M.compute()
derived = M.get_current_derived_parameters(['tau_rec','
    conformal_age'])
thermo = M.get_thermodynamics()
print thermo.viewkeys()
```

```
dict_keys(['x_e', 'g [Mpc^-1]', 'conf. time [Mpc]', "kappa' [Mpc^-1]", '
tau_d', 'Tb [K]', 'c_b^2', 'exp(-kappa)', 'z'])
```

So this big array contains all background quantities for each value of `'z'` (redshift). (Note: `x_e` is the ionisation fraction, `kappa` is the optical depth, `kappa'` is the scattering rate, `g` is the visibility function, `tau_d` is the baryon optical depth).

There are also two functions directly giving interpolated values of thermodynamical quantities at a given redshift:
`.ionisation_fraction(z)`, `.baryon_temperature(z)`

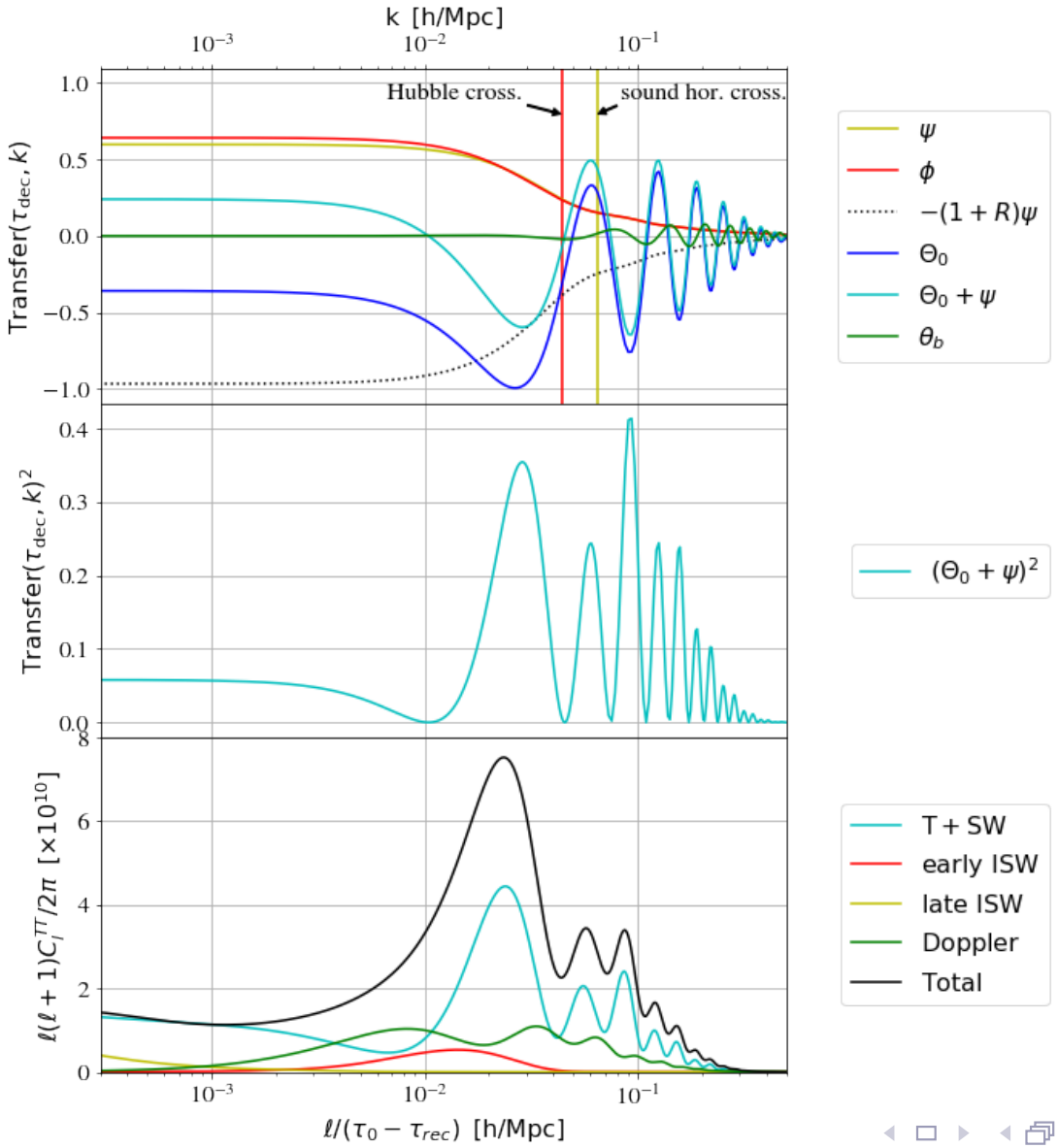
Thermodynamics quantitites

List of derived parameters that can be passed as arguments of `.get_current_derived_parameters([..,..,...])`:

```
# background:
'h', 'H0', 'Omega_Lambda', 'Omega0_fld',
'age', 'conformal_age',  'm_ncdm_in_eV',
'm_ncdm_tot', 'Neff', 'Omega_m', 'omega_m',
# thermodynamics:
'tau_reio', 'z_reio', '100*theta_s', 'YHe', 'n_e',
# quantitites at recombination:
'z_rec', 'tau_rec', 'rs_rec', 'rs_rec_h', 'ds_rec',
'ds_rec_h', 'ra_rec', 'ra_rec_h', 'da_rec', 'da_rec_h',
# quantities at baryon drag:
'z_d', 'tau_d', 'ds_d', 'ds_d_h', 'rs_d', 'rs_d_h',
# primordial perturbations:
'A_s', 'ln10^{10} A_s', 'n_s', 'sigma8', 'exp_m_2_tau_As',
'alpha_s', 'beta_s', 'r', 'r_0002', 'n_t', 'alpha_t',  '
    exp_m_2_tau_As',
+ others related to inflation/isocurvature
```

Perturbations at given time

With `notebooks/one_time.ipynb` or `scripts/one_time.py`:



Perturbations at given time

With `notebooks/one_time.ipynb` or `scripts/one_time.py`:

Main steps:

```
M = Class()
M.set(common_settings)
common_settings = {'output':'tCl,mTk,vTk',...}
M.set({'z_pk':z_rec}) # for transfer functions at z<z_rec
M.compute()
one_time = M.get_transfer(z_rec)
print one_time.viewkeys()
```

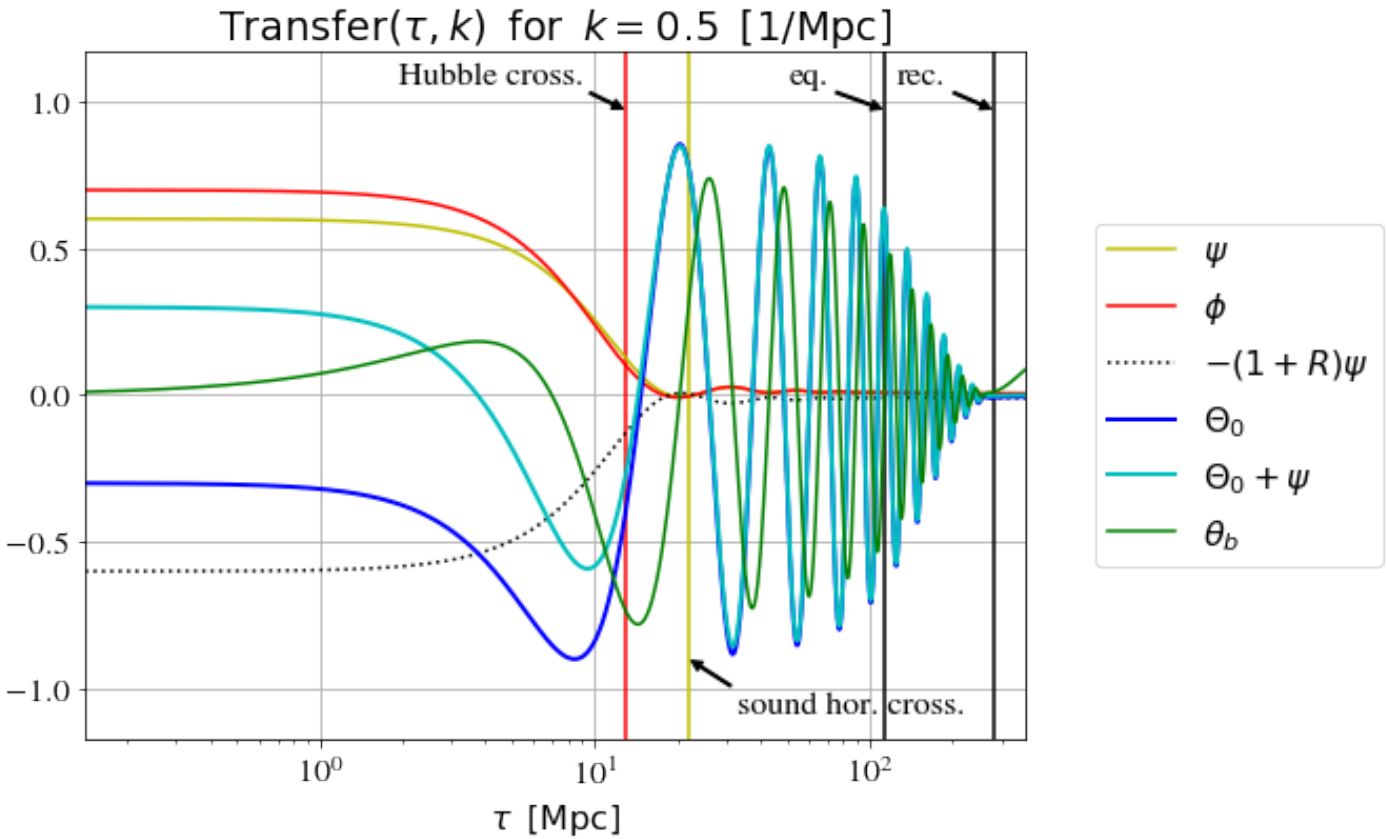
```
dict_keys(['phi', 'psi', 't_cdm', 't_b', 'd_tot', 't_g', 'd_ur', 'd_cdm'
, 'd_b', 't_tot', 't_ur', 'd_g', 'k (h/Mpc)'])
```

```
k = one_time['k (h/Mpc)']
Theta0 = 0.25*one_time['d_g']
phi = one_time['phi']
...
```

The key step was to include `'mTk'` in the output. Setting `'z_pk'` was also crucial to get transfer functions at high redshift (default: `'z_pk'`=0 and we would only be able to get the perturbations today).

Perturbations for a given wavenumber

With `notebooks/one_k.ipynb` or `scripts/one_k.py`:



Perturbations for a given wavenumber

With `notebooks/one_k.ipynb` or `scripts/one_k.py`:

Main steps:

```
k = 0.5  # 1/Mpc
common_settings = {'output':'mPk','k_output_values':k,...}
M = Class()
M.set(common_settings)
M.compute()
all_k = M.get_perturbations()
one_k = all_k['scalar'][0]
print one_k.viewkeys()
```

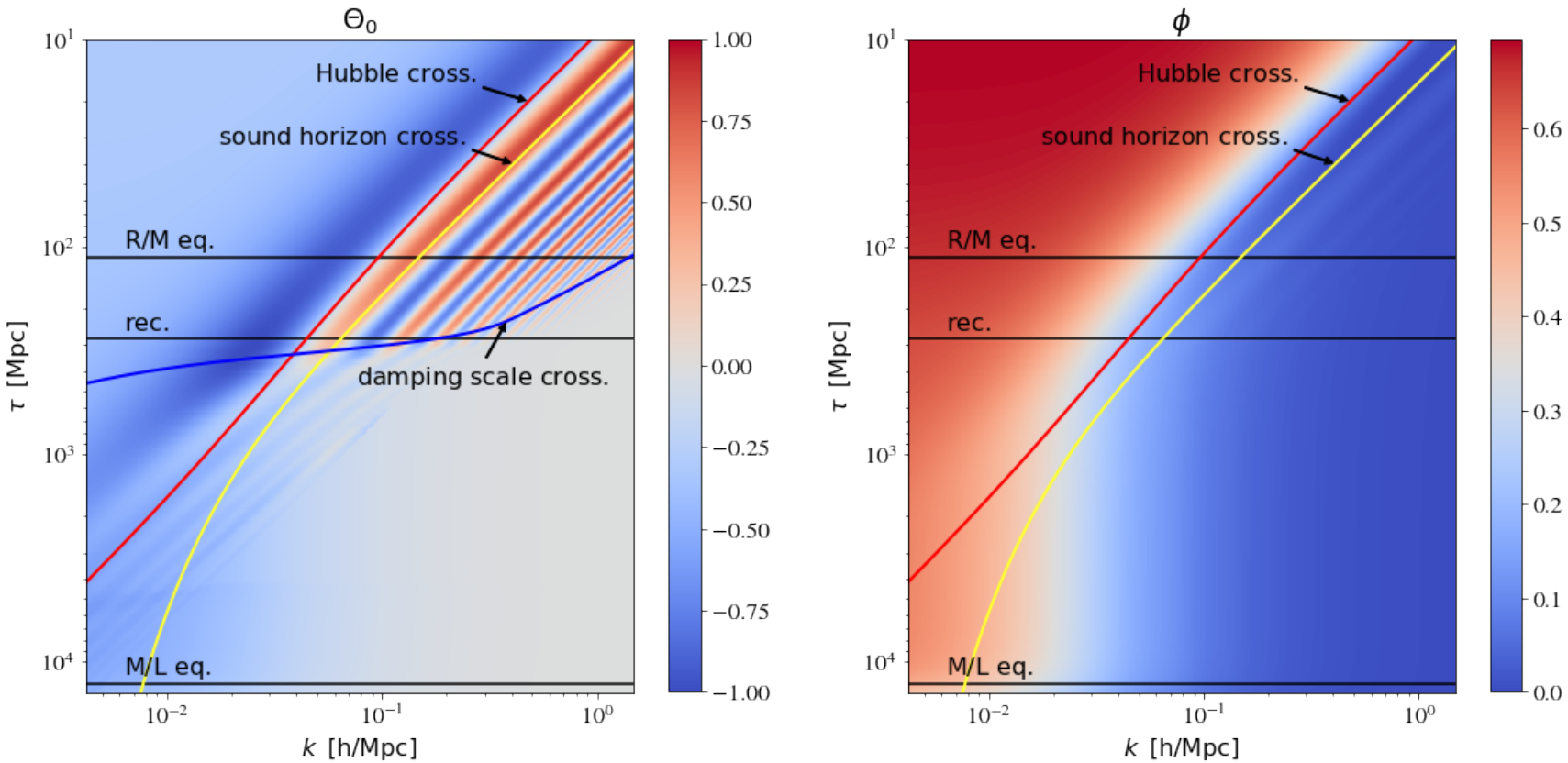
```
dict_keys(['a', 'theta_g', 'phi', 'pol0_g', 'theta_b', 'theta_ur', '
shear_ur', 'shear_g', 'tau [Mpc]', 'theta_cdm', 'delta_ur', 'psi', '
pol2_g', 'delta_g', 'delta_cdm', 'pol1_g', 'delta_b'])
```

```
tau = one_k['tau [Mpc]']
Theta0 = 0.25*one_k['delta_g']
phi = one_k['phi']
...
```

Remark: `'k_output_values'` can be set to a list: `'k_output_values'='0.05,0.1,0.4'`. Each is labelled by i starting from zero and the perturbations are in `M.get_perturbations()['scalars'][i]['key']`

Perturbations in (k, τ) space

With `notebooks/many_k.ipynb` or `scripts/many_k.py`:



Sophisticated script (and long to execute) but no new command with respect to previous cases.