CLASS

the Cosmological Linear Anisotropy Solving System[1]

Julien Lesgourgues
TTK, RWTH Aachen University

CCA, New York, 15-16.07.2019

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

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?

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 JL'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-2.7.2/), you should see:

```
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`:

```
# huge reference file containing
# all possible input parameters with comments
explanatory.ini

# slim file matching Planck 2015 "baseline model" bestfit
base_2015_plikHM_TT_lowTEB_lensing.ini

# slim file matching Planck 2018 "baseline model" bestfit
base_2018_plikHM_TTTEEE_lowl_lowE_lensing.ini
```

- All possible input parameters and details on the syntax explained in `explanatory.ini`
- For *basic* usage: `explanatory.ini` ≡ full documentation of the code
- This is only a reference file; we advise you to *never* modify it:
  - either copy it and reduce it to a shorter and more friendly file,
  - or start from a slim file,
  - or write your own from scratch with only needed input lines.

Running in terminal with input file (old fashioned)

Try for instance:

```
> ./class base_2018_plikHM_TTTEEE_lowl_lowE_lensing.ini
```

It gives some output:

```
Reading input parameters
 -> matched budget equations by adjusting Omega_Lambda =
     6.840972e-01
Running CLASS version v2.7.2
# selected lines from the output:
  -> age = 13.797336 Gyr
  -> radiation/matter equality at z = 3406.907947
  -> recombination at z = 1088.798382
  -> reionization  at z = 7.681290
  -> [WARNING:] Halofit non-linear corrections could not be
      computed at redshift z= 3.25 and higher.
  -> sigma8=0.811718 (computed till k = 8.44246 h/Mpc)
  -> sigma8 (ONLY CDM+BARYON)=0.8152 # Paco's special
(...)
Writing output files in output/
    base_2018_plikHM_TTTEEE_lowl_lowE_lensing_...
```

Running in terminal with input file (old fashioned)

Try for instance:

```
> ./class base_2018_plikHM_TTTEEE_lowl_lowE_lensing.ini
```

It gives some output:

```
Reading input parameters
 -> matched budget equations by adjusting Omega_Lambda =
     6.840972e-01
Running CLASS version v2.7.2
# selected lines from the output:
  -> age = 13.797336 Gyr
  -> radiation/matter equality at z = 3406.907947
  -> recombination at z = 1088.798382
  -> reionization  at z = 7.681290
  -> [WARNING:] Halofit non-linear corrections could not be
      computed at redshift z= 3.25 and higher.
  -> sigma8=0.811718 (computed till k = 8.44246 h/Mpc)
  -> sigma8 (ONLY CDM+BARYON)=0.8152 # Paco's special
(...)
Writing output files in output/
    base_2018_plikHM_TTTEEE_lowl_lowE_lensing_...
```

- Chatty behavior comes from 10 verbose parameters fixed to 1 in `base_2018_plikHM_TTTEEE_lowl_lowE_lensing.ini` (see them with `> tail base_2018_plikHM_TTTEEE_lowl_lowE_lensing.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.

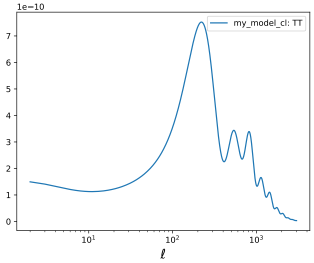
Running in terminal with input file (old fashioned)

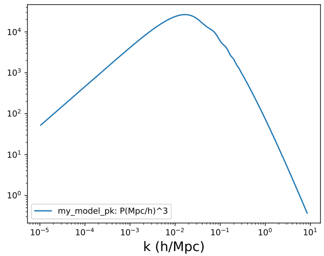
Results are in several files `output/my_model_*.dat`
Can be quickly plotted with provided python script `CPU.py` (Class Plotting Unit):

```
> 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
```





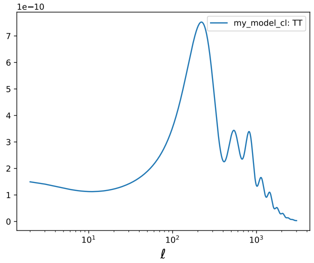
Running in terminal with input file (old fashioned)

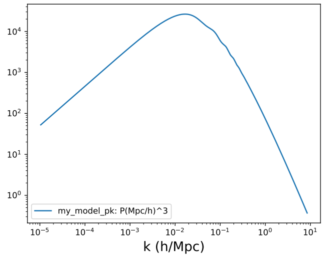
Results are in several files `output/my_model_*.dat`
Can be quickly plotted with provided python script `CPU.py` (Class Plotting Unit):

```
> 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
```

Running class from python

class as a Python module

- based on wrapper located in python/classy.pyx (developed initially by B. Audren and extended by many others)
- the compilation produces a python module classy.py and installs it on your computer (can be called from anywhere)
- wrapper written in Cython, encapsulates most useful class variables/functions, contains extra functions (e.g. MontePython-motivated)
- (project: 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 which are available since v2.7.0 in the folders scripts/ and notebooks/
- with jupyter installed, open the notebooks with e.g.

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

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

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

Python wrapper

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

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

```
# create instance of the class "Class"
LambdaCDM = Class()
# pass input parameters
LambdaCDM.set({'omega_b':0.022032,'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:]
```

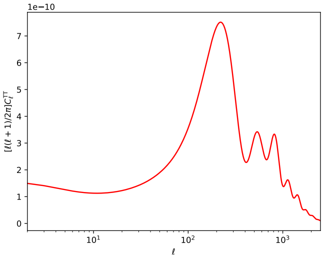
Python wrapper

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

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

(some systems prefer %matplotlib notebook to %matplotlib inline)

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



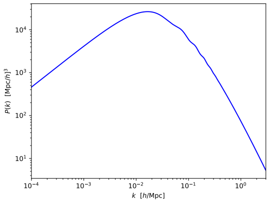
```
plt.savefig('warmup_cltt.pdf')
```

Python wrapper

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

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

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



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

Python wrapper

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

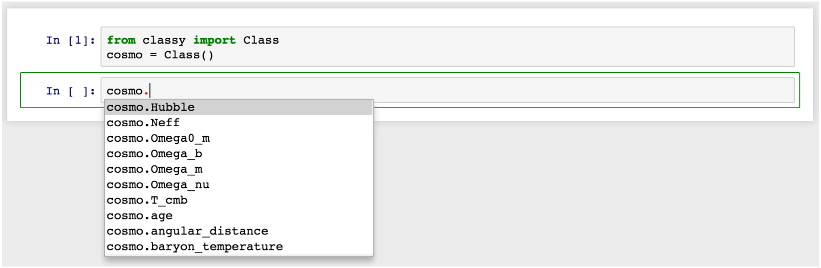
Obsolete statements:

```
# optional: clear content of LambdaCDM (to reuse it for another model)
LambdaCDM.struct_cleanup()
# optional: reset parameters to default
LambdaCDM.empty()
```

(still found in some scripts but useless: handled automatically by wrapper since `v2.7.2`)

Python wrapper: IPython notebooks

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

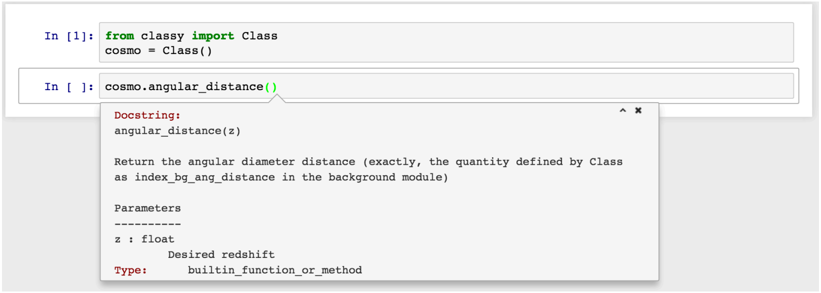


```
In [1]: from classy import Class
        cosmo = Class()

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

Python wrapper: IPython notebooks

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



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

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

```
Docstring:
angular_distance(z)

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

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

Species in public `class`

- compulsory: photons: `T_cmb` or `Omega_g` or `omega_g`
- compulsory: baryons: `Omega_b` or `omega_b`
- ultra-relativistic species (massless neutrinos): `N_ur` or `Omega_ur` or `omega_ur`
- cold dark matter: `Omega_cdm` or `omega_cdm` (possibly annihilating: `annihilation`, etc.)
- `N_ncdm` distinct non-cold dark matter species (massive neutrinos, warm dark matter...): `m_ncdm` or `Omega_ncdm` or `omega_ncdm` plus lots of options
- cold dark matter decaying into dark radiation: `Omega_dcdmdr` or `omega_dcdmdr` plus `Gamma_dcdm`
- 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`

Species in public class

Budget equation:

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

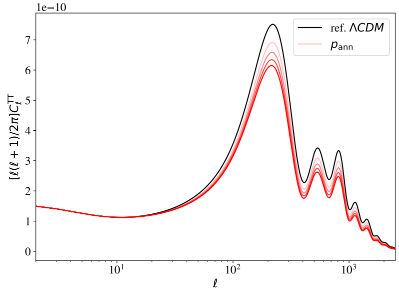
To avoid over-constraining the input, one of the last three (Omega_Lambda, Omega_fld, Omega_scf) must be left unspecified and class will assign it using budget equation.

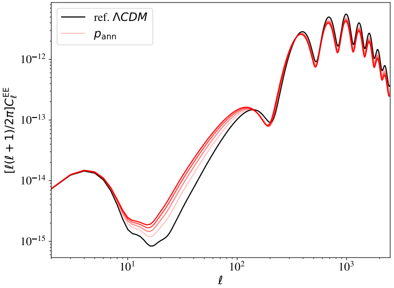
- default: Omega_Lambda is automatically adjusted, assuming Omega_fld $=$ Omega_scf $= 0$.
- if you pass Omega_Lambda $= 0$: Omega_fld is automatically adjusted, assuming Omega_scf $= 0$.
- if you pass Omega_Lambda $= 0$ and Omega_fld $= 0$: Omega_scf is automatically adjusted.

Allows whatever combination.
E.g. to get Λ plus a DE fluid:
Omega_Lambda=0.2, Omega_scf=0 or Omega_fld=0.3, Omega_scf=0

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

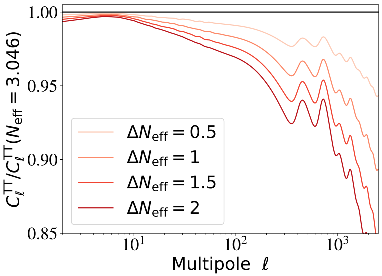
With `notebooks/varying_pann.ipynb` or `scripts/varying_pann.py`:

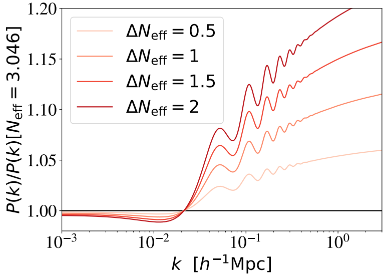
Main steps:

```
var_name = 'annihilation'
var_array = np.linspace(0,1.e-5,5)
common_settings = {'output':'tCl,pCl,lCl,mPk', ...}
# loop over varying parameter values
for i,var in enumerate(var_array):
  M = Class()
  M.set(common_settings)
  M.set({var_name:var})
  M.compute()
  clM = M.lensed_cl(2500)
  # ... plotting ...
  M.struct_cleanup() # clear all class output
  M.empty()          # clear input previously set by .set()
```

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:
(given $z_{\rm eq} = \omega_m/\omega_r, \quad \omega_r \propto (1 + cN_{\rm eff}), \quad \omega_m = \omega_b + \omega_{cdm}, \quad h = \sqrt{\omega_m/(1 - \Omega_\Lambda)}$)

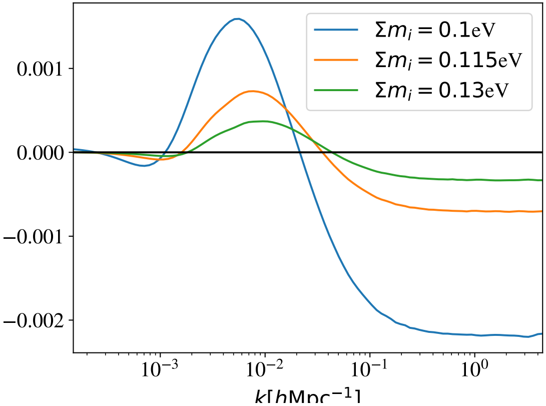
```
M = {}
for i, N_ur in enumerate(var_array):
    # The goal is to vary
    # - omega_cdm by a factor alpha = (1 + coeff*Neff)/(1 +
        coeff*3.046)
    # - h by a factor sqrt*(alpha)
    # in order to keep a fixed z_equality(R/M) and
        z_equality(M/Lambda)
    alpha = (1.+coeff*N_ur)/(1.+coeff*3.046)
    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:

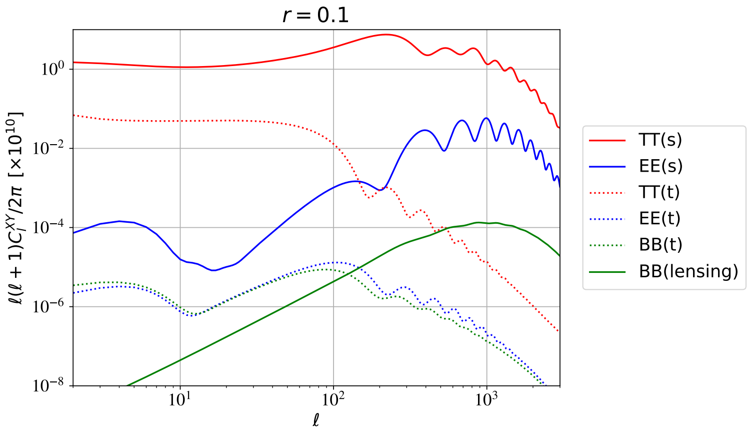
Main steps:

```
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

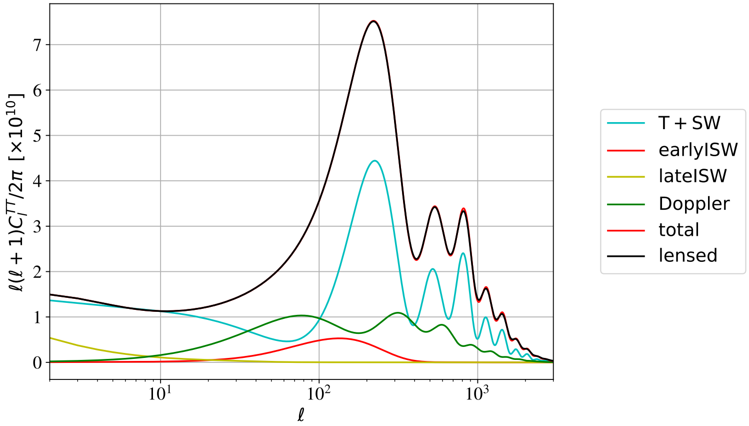
With `notebooks/cl_ST.ipynb` or `scripts/cl_ST.py`:

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

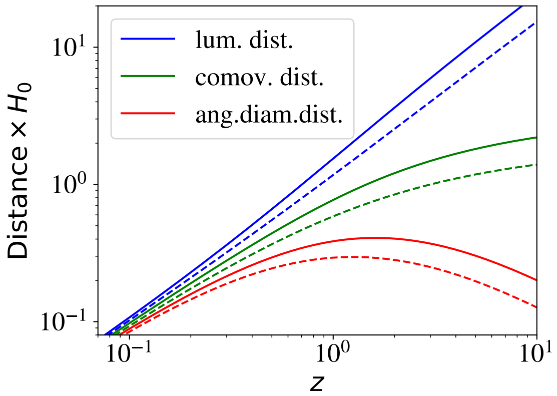
With `notebooks/cltt_terms.ipynb` or `scripts/cltt_terms.py`:

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 quantities

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



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

Background quantities

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()
```

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 quantities

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

Main steps:

```
# Just to cross-check that Omega_Lambda is negligible
# (but not exactly zero because we neglected radiation)
der = CDM.get_current_derived_parameters(['Omega0_lambda'])
print der
print "Omega_Lambda =",der['Omega0_lambda']
```

```
{'Omega0_lambda':  -9.167135654530867e-05}
Omega_Lambda = -9.16713565453e-05
```

Background quantities

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',
# --> quantities 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
```

Background quantities

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]'`.

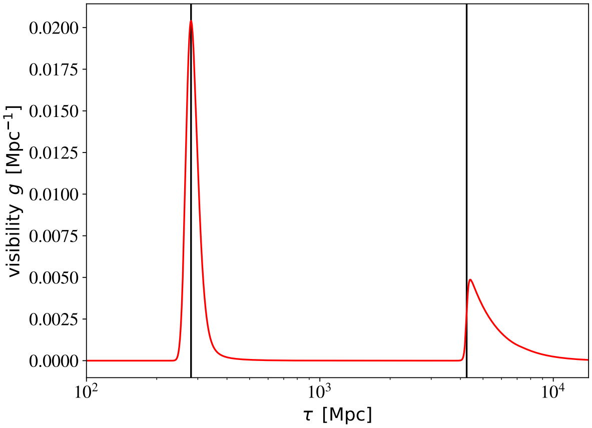
There are also many functions directly giving interpolated values of background quantities at a given redshift:

```
.Hubble(z), .angular_distance(z), .luminosity_distance(z),
.scale_independent_growth_factor(z),
.scale_independent_growth_factor_f(z),
.sigma(R,z),
```

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

Thermodynamics quantities

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 quantities

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)`

Primordial spectra

We don't have an example of notebook here, but there are lots of options for the primordial spectra, depending what P_k_ini type is set to (see explanatory.ini or https://lesgourg.github.io/class-tour/Tokyo2014/lecture12_primordial.pdf for more details):

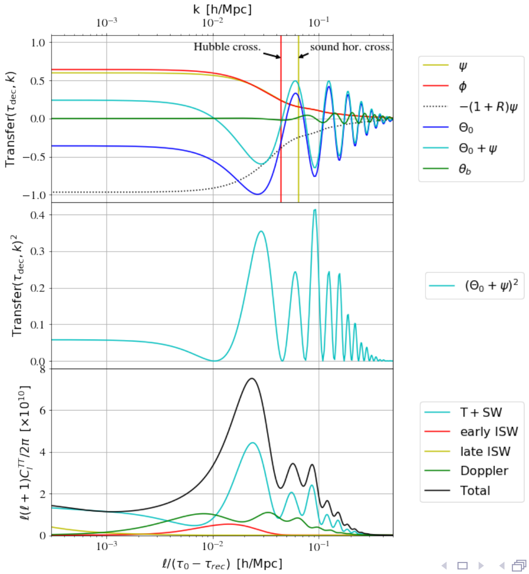
- analytic_Pk: traditional input (A_s, n_s, alpha_s, k_pivot, r, n_t plus many others, in particular for isocurvature modes)
- two_scales: an alternative used in Planck inflation papers for isocurvature modes
- external_Pk: primordial spectra read on-the-fly from external code
- inflation_V: full inflation simulator for given function $V(\phi - \phi_{\rm pivot})$
- inflation_H: full inflation simulator for given function $H(\phi - \phi_{\rm pivot})$
- inflation_V_end: full inflation simulator for given function $V(\phi)$ and N_*

In python notebook/script: the scalar and tensor primordial spectra $\mathcal{P}_{\mathcal{R}}(k)$, $\mathcal{P}_h(k)$ can be extracted with the function .get_primordial()

With the inflation simulator: the parameters A_s, ln10^{10} A_s, n_s, alpha_s, beta_s, r, r_0002, n_t, alpha_t are then *derived* parameter accessible with .get_derived_parameters(...)

Transfer functions at given time

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



Transfer functions 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',...,
                   'gauge':'newtonian'}
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()
```

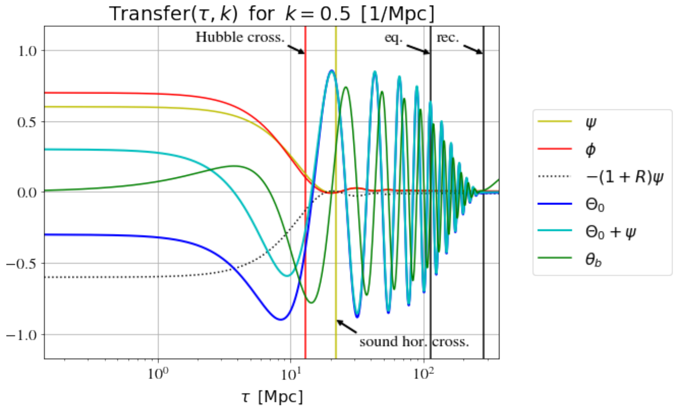
```
dict_keys(['phi', 'psi', 't_cdm', 't_b', 'd_tot', 't_g', 'd_ur', 'd_cdm'
, 'd_b', 't_tot', 't_ur', 'd_g', 'k (h/Mpc)'])
```

```
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).

Transfer functions for a given wavenumber

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



Transfer functions 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()
```

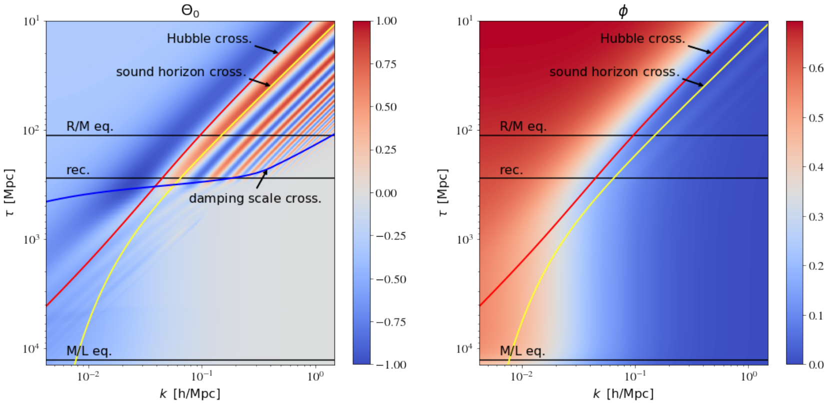
```
dict_keys(['a', 'theta_g', 'phi', 'pol0_g', 'theta_b', 'theta_ur', '
shear_ur', 'shear_g', 'tau [Mpc]', 'theta_cdm', 'delta_ur', 'psi', '
pol2_g', 'delta_g', 'delta_cdm', 'pol1_g', 'delta_b'])
```

```
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']`

Transfer functions in (k, τ) space

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



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