CLASS

the Cosmological Linear Anisotropy Solving System

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
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 scripts/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.ipynb

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

  > python scripts/warmup.py
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,lC1,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:]
c1TT = cls['tt'][2:]
c1EE = cls['ee'][2:]  
c1PP = cls['pp'][2:]
Python wrapper

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

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

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

(some systems prefer %matplotlib notebook to %matplotlib inline)
First basic example (notebooks/warmup.ipynb or scripts/warmup.py)

```python
# 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'$k , [h/\text{Mpc}]$')
plt.ylabel(r'$P(k) , [\text{Mpc}/h]^3 $')
plt.plot(kk, Pk, 'b-')
plt.savefig('warmup_pk.pdf')
```

![Plot of P(k) vs k (h/Mpc) with log-log scale.](image.png)
Python wrapper

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

# optional: clear content of LambdaCDM (to reuse it for another model)
LambdaCDM.struct_cleanup()
# optional: reset parameters to default
LambdaCDM.empty()
The TAB key after the dot gives you the list of available classy methods (= available functions and quantities) in a scrolling menu:
The TAB+SHIFT keys after the () gives you a short doc on each method (expand it by clicking +):
Species in public **class**

- **compulsory**: photons: $T_{\text{cmb}}$ or $\Omega_g$ or $\omega_g$
- **compulsory**: baryons: $\Omega_b$ or $\omega_b$
- **ultra-relativistic species (massless neutrinos)**: $N_{\text{ur}}$ or $\Omega_{\text{ur}}$ or $\omega_{\text{ur}}$
- **cold dark matter**: $\Omega_{\text{cdm}}$ or $\omega_{\text{cdm}}$ (possibly annihilating: annihilation, etc.)
- $N_{\text{ncdm}}$ distinct non-cold dark matter species (massive neutrinos, warm dark matter...): $m_{\text{ncdm}}$ or $\Omega_{\text{ncdm}}$ or $\omega_{\text{ncdm}}$ plus lots of options
- **cold dark matter decaying into dark radiation**: $\Omega_{\text{dcdm}}$ or $\omega_{\text{dcdm}}$ plus $\Gamma_{\text{dcdm}}$
- **spatial curvature** $\Omega_k$
- **cosmological constant** $\Omega_{\Lambda}$
- **fluid** $\Omega_{\text{fld}}$ plus $w_0_{\text{fld}}$, $w_a_{\text{fld}}$, $c_{s2_{\text{fld}}}$, etc.
- **scalar field (quintessence)** $\Omega_{\text{scf}}$ plus specifications

All details are in explanatory.ini

To avoid over-constraining the input, one of the last three ($\Omega_{\Lambda}$, $\Omega_{\text{fld}}$, $\Omega_{\text{scf}}$) must be left unspecified and **class** will assign it using budget equation. **Default**: $\Omega_{\text{fld}} = \Omega_{\text{scf}} = 0$ so $\Omega_{\Lambda}$ is automatically adjusted.
We called \texttt{class} within a loop with different values of the DM annihilation parameter 
\[ p_{\text{ann}} = \langle \sigma v \rangle / m. \]
Plots with varying parameters

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

Main steps:

```python
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 $N_{\text{eff}}$ for massless neutrinos (in fact $N_{\text{ur}}$) while keeping $z_{\text{eq}}$ and $z_{\Lambda}$ fixed, which implies to adjust $h$ and $\omega_{\text{cdm}}$ 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:

```python
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
```
c1M[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 $\sum_i m_i$. 

\[ \Sigma m_i = 0.1 \text{eV} \]
\[ \Sigma m_i = 0.115 \text{eV} \]
\[ \Sigma m_i = 0.13 \text{eV} \]
Plots with varying parameters

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

Main steps:

```python
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:

$r = 0.1$

$\ell(\ell+1)C_{XY}^l/2\pi \times 10^{10}$

- $C_{TT}(s)$
- $C_{EE}(s)$
- $C_{TT}(t)$
- $C_{EE}(t)$
- $C_{BB}(t)$
- $C_{BB(\text{lensing})}$
Contributions to CMB $C_{l}$'s

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

Main steps:

```python
# 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)
...```
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:

```python
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:

![Graph showing distances vs. redshift](graph.png)

Similar to plot in Scott Dodelson’s *Modern Cosmology* book. Solid = ΛCDM, dashed = Einstein-De-Sitter (Ω_m = 1).
Background quantities

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

Main steps:

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

Main steps:

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

Main steps:

```python
# 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).
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:

```python
M = Class()
M.set(common_settings)
M.compute()
derived = M.get_current_derived_parameters(['tau_rec', 'conformal_age'])
thermo = M.get_thermodynamics()
pdict = 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 $V(\phi - \phi_{pivot})$
- **inflation_H**: full inflation simulator for given function $H(\phi - \phi_{pivot})$
- **inflation_V_end**: full inflation simulator for given function $V(\phi)$ and $N_*$

In python notebook/script: the scalar and tensor primordial spectra $P_R(k)$, $P_h(k)$ 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(...)`
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',...,
                   '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).
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:

```python
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 \((k, \tau)\) 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.